Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Bendik Bogfjellmo

Two deep learning approaches to Sound Event Detection for bird sounds in the arctic biosphere

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Guillaume Dutilleux
July 2021

NTNU
Norwegian University of Science and Technology

Bendik Bogfjellmo

Two deep learning approaches to Sound Event Detection for bird sounds in the arctic biosphere

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Guillaume Dutilleux
July 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Preface

The work I've undergone during this thesis has again reiterated the age old adage: "keep it simple, stupid"; while it's fun and exciting to experiment with new methods, you'll most likely be left disappointed at the end of the day if your ideas don't pan out. However, these things are in some ways what univiersities are for, trying out lots of probably dumb things that don't necessarily have to be practically applicable.

I'm very pleased that I had the opportunity to find out whether my ideas could result in a new method for doing some stuff, but I'm also very content with "just making stuff that works", which is probably something you have to be to make it as an engineer.

I'd like to take the opportunity to thank my supervisor, Guillaume Dutilleux, for making this fun and inspiring task available to us students, in addition to *Norsk Institutt for Naturforskning*, especially John Atle Kålås, for allocating the funds to procure a high-memory, high-capacity GPU which made it feasible to research much more than otherwise would be possible. *Norsk Institutt for Naturforskning* also has to be thanked for providing huge amounts of recordings, making the rough work of data collection unnecessary.

A huge thanks has to be given to Håkon Hukkelås, for providing baseline codebases for both projects, making the development of the thesis work's codebase a figurative walk in the park compared to what it could have been.

Thanks to my sister, Silja Eggen Bogfjellmo, and my brother in law, Jonas Agentoft Eggen, for reading through this piece of garbage and finding some of the dumb mistakes I've made.

Lastly, I'd like to thank my girlfriend, Anna Lifen Tennfjord, for being understanding, and making room for a time with huge workloads.

Abstract

This master's thesis revolves around the work on creating a deep learning based sound event detection system for birds in the arctic biosphere. The thesis presents a workflow starting from raw audio data, to a fully functional state-of-the-art model for the task of sound event detection. The work undertaken and described includes approaches to annotation of audio data, dataset creation, augmentation & feature engineering, state-of-the-art deep learning model creation, inference algorithm development, and productivity focused experimental setups for training. Lastly, the work includes an extensive codebase which simplifies creation of models for solving sound event detection problems.

The training and validation datasets have been annotated from audio provided by Norsk Institutt for Naturforskning (NINA) , amounting to a total of 5740 sound events spanning 5 different sound event classes, annotated from a source of 450 hours of audio. A test dataset to verify generalization of the model has been sourced from user contributed recordings from Xeno Canto, amounting to 677 sound events, spanning the same 5 classes.

An attempt has been made to benefit from a state-of-the-art-image based object detection method, the Single Shot Multibox Detector (SSD), to create a custom multiline, equivalent architecture. This architecture performed at a classwise mean Average Precision (mAP) of 0.029, given a threshold value for Intersection over Union (IoU) of 0.5 implying a true positive prediction, this resulted in it being deemed unfit for practical applications.

A conventional spectrographic "sliding-window" (SED) classifier apporach to Sound Event Detection, as in the example of BirdNET has also been developed. The training results of hundreds of individual training sessions of this classifier are presented. The best performing classification model achieved a mAP of 0.989 on the classifier validation dataset and 0.971 on the classifier test dataset, with a window size of 2.5 seconds and positive ground truths implying 25% of the window containing the sound event. Due to its results, this classifier was deemed fit for practical application and a sliding-window moving average inference algorithm has been implemented, with ability to format output as csv-files or Audacity-compliant label-files. Some examples of the outputs of the inference algorithm are also presented.

The work in this thesis also includes an implementation of an extensive, extendable, reusable, and nearly fully configurable, sliding-window sound event detection codebase, which is mainly implemented through the pytorch framework, using torchvision and torchaudio as supplementary frameworks. The codebase should be suitable for application on most sound event detection problems with minimal efforts except annotation, allowing further work to be done more effectively in the future.

This thesis is written under the assumption that the reader has some background knowledge within state-of-the-art deep learning techniques, and has at least an introductory level familiarity with the pytorch-framework. A couple of key ideas behind state-of-the-art techniques are elaborated upon, but it does not elaborate on the core mathematical fundamentals of how ANNs work.

A recommendation for an introduction to ANNs is Michael Nielsens "Neural networks and deep learning" in addition to Grant Sandersons introductory video series on the topic,

published on his YouTube channel “3Blue1Brown”. To get both a practical and a theoretical grip on the pytorch framework and some of the more state-of-the-art deep learning techniques, reading papers on the techniques and pytorch-implementations of them is recommended. A couple of recommendations are the ResNet-paper and the SSD-paper. Additionally, pytorch can be described as one of the best documented frameworks that’s publically available, and it can be highly recommended as a learning source.

Sammendrag

Denne masteroppgaven dreier seg rundt et arbeid for å skape et dyp-lærings-basert system for lydhendelsesdeteksjon (”sound event detection”) for fugler i den arktiske biosfæren. Oppgaven presenterer en arbeidsflyt fra råe lydfiler, til en fullt fungerende, toppmoderne modell for oppgaven av lydhendelsesdeteksjon. Oppgaven inneholder beskrivelser av tilnærminger til annotering av lyddata, datasett-skaping, augmentering & egenskapsskaping for lyddata, toppmoderne dyplærings-modeller, utvikling av algoritme for postprossessering av prediksjoner, og produktivitetsfokusert eksperimentelt oppsett for modelltrening. Sist, men ikke minst omhandler oppgaven utvikling av en omfattende kodebase som forenkler utvikling av “glidevindu”-løsninger for lydhendelsesdeteksjonsproblemer i fremtiden.

Gjennom oppgaven har datasett for trening og validering blitt annotert fra opptak gjort av Norsk Institutt for Naturforskning (NINA). Opptakene som annoteringene er gjort på, består til sammen av 450 timer med lydopptak, og fra disse har det blitt annotert til sammen 5740 unike lydhendelser som spenner 5 forskjellige klasser. Et testdatasett for å verfisere generalisering av modellen har blitt hentet ut fra brukerbidrag til Xeno Canto, og fra disse bidragene har det blitt annotert 677 lydhendelser som spenner de 5 samme lydklassene.

Under oppgaven har det blitt gjort et forsøk på å skape en lydhendelsesdeteksjons-parallell av “Single Shot Multibox Detector”. Denne arkitekturen førte til en “mean Average Precision” (mAP) på 0.001 regnet ut med en terskelverdi for “Inetersection over Union” (IoU) på 0.5 for en sann positiv prediksjon, dette førte til at videre implementasjon av en praktisk applikasjon ble utelukket i arbeidet.

En forholdsvis konvensjonell spektrografisk “glidevindu”-klassifisator-tilnærming til lydhendelsesdeteksjon, i likhet med “BirdNet” har også blitt utviklet. Resultater fra hundrevis av eksperimentelle treningsøkter på klassifisatorer for dette er presentert. Den beste klassifisatoren klarte å oppnå en mAP på 0.989 på valideringsdatasettet, og 0.971 på testdatasettet. Dette ble gjort med vindustørrelse på 2.5 sekund, og med en antakelse om at dersom 25% av vinduet inneholder en lydhendelse av en klasse, er det en positiv instans for klassen. Denne klassifisatoren ble dømt til å være passende for implementasjon, og en algoritme for postprossessering av flertallige klasseprediksjoner for enkeltvindu til kontinuerlige lydhendelsesprediksjoner presenteres. Den praktiske tilnærmingen inkluderer også mulighet for eksportering til csv-filer eller Audacity-kompatible merkelapp-filer. Eksempler på lydhendelser som er predikert, er også presentert.

Arbeidet under oppgaven inkluderer også implementeringen av en omfattende, utvidbar, gjenbrukbar, og nærmest fullt konfigurerbar kodebase for lydhendelsesdeteksjonsproblemer. Kodebasen er først og fremst implementert gjennom pytorch-rammeverket for python, med torchvision og torchaudio som støtterammeverk. Kodebasen skal ha mulighet for å støtte tilnærmet hvilket som helst lydhendelsesdeteksjonsproblem med enklere innsats fra en bruker.

Oppgaven er skrevet med en antakelse om at leseren har litt bakgrunnskunnskap innen toppmoderne dyp-læring-teknikker, samt en forholdsvis grei kjennskap til pytorch-rammeverket. Et par oversiktlige ideer bak noen av de mer toppmoderne teknikkene in-

nenfor dyp læring er presentert, men oppgaven går ikke inn på detalj på den matematiske kjernen til kunstige nevrale nett.

En anbefaling for en introduksjon til kunstige nevrale nett er Michael Nielsens ”Neural networks and deep learning” samt Grant Sandersons videoserie på temaet, publisert på YouTube-kanalen “3Blue1Brown”. For å få en praktisk, samt teoretisk forståelse for noen av de mer toppmoderne teknikkene innenfor emnet, kan det anbefales å lese artiklene om noen av de større nyvinningene innen emnet, samt lese deres implementering i pytorch. To anbefalinger til noen mer moderne artikler er SSD og ResNet. Tilleggsvis kan det nevnes at pytorch kan beskrives som en av de best dokumenterte rammeverkene til python som er offentlig tilgjengelig, og det kan anbefales på det høyeste å anvende dokumentasjonen deres som en læringskilde.

Table of Contents

List of Figures iv

List of Tables vi

1 Introduction 1

1.1 Task description . . . 2

1.2 Motivation . . . 2

1.2.1 Codebase motivation . . . 3

1.2.2 Dataset creation motivation . . . 3

2 Theory 4

2.1 Evaluation of detectors . . . 4

2.1.1 Intersection over Union . . . 4

2.1.2 Average Precision . . . 5

2.2 Data augmentation & feature engineering . . . 6

2.2.1 Spectrograms . . . 7

2.2.2 Logscale . . . 7

2.2.3 Standardization . . . 7

2.2.4 Random time shifting . . . 8

2.2.5 Gaussian noise . . . 9

2.3 Sliding-window classifier . . . 9

2.3.1 Sliding-window classification . . . 9

2.3.2 Multilabel classification . . . 10

2.3.3 Moving mean of class confidence . . . 12

2.4 Some architectural details of a Single Shot Multibox Detector (SSD) . . . 13

2.4.1 Detector heads . . . 14

2.4.2 Non-maximum suppression . . . 15

2.4.3 Hard negative mining . . . 15

2.5 Feature extraction . . . 15

2.5.1 Compound scaling 16
2.5.2 BiFPN 17

3 Methodology **19**

3.1 Datasets 19
3.1.1 Dataset annotation 20
3.1.2 Annotated sound patterns 21
3.1.3 Dataset generation 23
3.1.4 Test dataset 23
3.1.5 Multilabel classification dataset 24
3.2 Experimental training setup 24
3.3 Data augmentation & feature engineering 25
3.3.1 Annotation sample formatting 25
3.3.2 Random time shifting 25
3.3.3 Gaussian noise 26
3.3.4 Spectrogram creation 27
3.4 SSD-based architecture 28
3.4.1 Architectural modifications 28
3.4.2 Convolutional backbones 30
3.4.3 Pytorch dataset implementation 30
3.4.4 Evaluation 32
3.5 Sliding-window architecture 32
3.5.1 Sliding-window dataset 32
3.5.2 Sliding-window inference 33
3.5.3 Average precision 35

4 Results **36**

4.1 SSD-based architecture 36
4.2 Sliding-window architecture 37
4.2.1 Quantitatively assessable results 37

4.2.2 Qualitatively assessable results . . . 39
4.3 Codebase . . . 43
4.4 Datasets . . . 43
4.4.1 SSD-based architecture dataset . . . 43
4.4.2 Classifier dataset . . . 44
4.5 Experimental findings . . . 45
4.5.1 Window size and IoU thresholding . . . 45
4.5.2 Backbone evaluation . . . 46
4.5.3 Inference timing . . . 47
4.5.4 Random Gaussian noise . . . 48

5 Discussion **50**
5.1 SSD-based architecture dataset . . . 50
5.2 Sliding-window based architecture dataset . . . 50
5.3 SSD-based architecture . . . 50
5.4 Sliding-window architecture . . . 51
5.5 Test dataset performance . . . 51
5.6 Codebase as framework for future detector development . . . 51

6 Conclusion **53**
6.1 SSD-based architecture . . . 53
6.2 Sliding-window architecture . . . 53
6.2.1 Codebase . . . 53
6.3 Recommendation of future work . . . 53
6.3.1 Datasets . . . 53
6.3.2 Codebase . . . 54

Bibliography **56**

Appendix **59**

A Xeno Canto data information **59**

B Code **62**

A Dataset creation . . . 62

A.1 Label parsing script . . . 62

A.2 Classification dataset generation . . . 67

B Configuration code . . . 68

C Configuration generation . . . 70

D Dataset . . . 72

D.1 Sliding-window training dataset . . . 72

D.2 Sliding-window inference dataset . . . 75

D.3 SSD-based architecture dataset . . . 76

E Transform code . . . 79

F Sliding-window target transform . . . 87

G Training automation . . . 88

G.1 Training script . . . 88

G.2 Run experiments script . . . 89

H Evalutaion code . . . 90

H.1 SSED evaluation code . . . 90

H.2 SSD-based architecture evaluation . . . 91

I Window-slide Inference script . . . 98

J Classifier Models . . . 102

J.1 ResNet50 & ResNet34 . . . 102

J.2 EfficientNet . . . 103

List of Figures

1 A self-made twist on one of my favorite comics, XKCD, the original made by Randall Munroe, a retired programmer/roboticist who now makes comic strips [23], the picture is licensed under CC BY-NC 2.5 [8] . . . 1

2 A diagram displaying the scope of this task as the system within the dotted line, input is an audio file, output is predictions of distinct types of bird vocalizations. . . . 2

3 An example of IoU calculations in the time dimension, if the IoU threshold were 0.5 here, Prediction 1 would be a true positive and Prediction 2 would be a false positive. 5

4 Example of k-nearest-neighbors with k=5 and two dimensions, where the black spot should be classified. 6

5 Random timeshift with IoU being over a predetermined threshold implying an active class in the randomly selected time series. 8

6 Sliding-window classification, where X_0, and X_1 is, respectively, the first and second window of the main audio recording, with Y_0 and Y_1 as the respective first and second window class predictions from the classification scheme. 10

7 Sigmoid function for $x \in [-6, 6]$. 11

8 Moving mean confidence value scheme. Hop size here is window size divided by 4, this has the implication that H, representing the length of C, is equal to N+3. N being the amount of hops through the entire record that inference is run on. 12

9 A brief overlook at the architecture of a Single Shot Multibox Detector [19]. The blocks marked as "Reduction" are in reality Convolutional Neural Networks that reduce the height and width dimensions by either strides or lack of padding. 13

10 Classification head kernel forward action. C denotes the channel dimension of the input feature map. The output is of the dimension 5×5, due to padding not shown in the figure. 14

11 An illustration of the concept of compound scaling, instead of utilizing one of the CNN scaling methods, all are combined. 16

12 An example of a FPN-network applied with a convolutional backbone. . . 17

13 An example of a BiFPN-layer applied with a convolutional backbone. . . 18

14 Map of the locations, the maps are provided by Open Street Map [25], and are therefore licensed under CC BY-SA [26]. The locations follow the naming conventions of NINA. 20

15 Screen capture from Audacity providing a practical example of the annotation method. 21

16 Common snipe winnowing sound spectrogram. 21

17 European golden plover call spectrogram. 22

18 European golden plover song spectrogram. 22

19 Whimbrel song spectrogram. The noisy lines are induced by rain. 22

20 Wood sandpiper song spectrogram. 23

21 Example displaying modified classification head kernel size for 1D object detection. 29

22 Experimental ground truth class labelling for classification loss, the criteria for a positive class label is listed at the bottom. 30

23 Precision-recall curve for the Wood sandpiper song. 37

24 Precision-recall curve for the Common snipe winnowing sound. 38

25 Precision-recall curve for the Whimbrel song. 38

26 Precision-recall curve for the European golden plover call sound. 39

27 Precision-recall curve for the European golden plover song. 39

28 An example of true positive predictions of relatively weak sound events of multiple labels. 40

29 An illustration of the predictions for the European golden plover being "fused" together into a single predicted sound event. 41

30 A false positive prediction for the call-vocalization of the European golden plover. 41

31 Two predictions of the Wood sandpiper song, the top label line being performed with a confidence threshold of 0.76, while the bottom label line is performed with a confidence threshold of 0.95. 42

32 True positive and false negative prediction of the Wood sandpiper song. The top label line contains a true positive from an inference with a confidence threshold of 0.76, the bottom label line contains a false negative from an inference with a confidence threshold of 0.95. 42

33 A heat map displaying the smallest validation losses achieved at different values for IoU threshold and window size. Darker/smaller values are better. 46

34 Best validation loss for the 40 different intensity values of the random gaussian noise data augmentation method. 49

List of Tables

1 Target birds for the project. Norwegian names are listed due to them being used during annotation. 3

2 Final results for both backbone and both class labelling methods for the SSD-based architecture. 36

3 Hyper parameters and configurations for the training sessions of the SSD-based architecture. 36

4 Sounds annotated from NINAs recordings. 43
5 Sounds annotated from Xeno Canto recordings. 44
6 Training/Validation data source locations. 44
7 Amount of sound event labels split into each data group. 44
8 Other hyperparameters and configurations for the experiments. 45
9 Key performance metrics for the tested backbones. 47
10 Hyperparameters and configurations for backbone evaluation. 47
12 Hyperparameters and configurations for inference times. 47
11 Model inference times for a 24 hour long audio file. 48
13 Hardware description 48
14 Hyperparameters and configurations for Gaussian noise experiments. . . . 48
15 Xeno Canto recordings description. 59

1 Introduction

In the past few years, the availability and feasability of developing deep Artificial Neural Networks (ANNs) has made the field figuratively explode with areas of application. To illustrate this point, in the original version of Figure 1, published in 2014, the woman requested a research team and 5 years to detect whether a bird was in a photo, while today, it can be achieved by grad students working outside their field of expertise in a matter of months, if not even days.

Figure 1: A self-made twist on one of my favorite comics, XKCD, the original made by Randall Munroe, a retired programmer/roboticist who now makes comic strips [23], the picture is licensed under CC BY-NC 2.5 [8]

.

As Figure 1 references, the main problem taken on in this master's thesis has been detection and classification of bird vocalizations. This problem falls into a broader category of machine learning problems commonly referred to as Sound Event Detection (SED).

1.1 Task description

The task at hand can be visually explained as making the system within the dotted line in Figure 2.

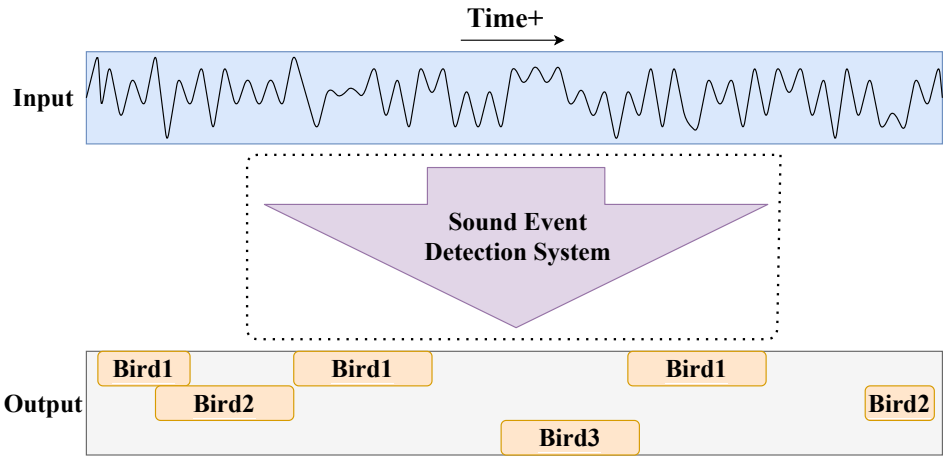

Figure 2: A diagram displaying the scope of this task as the system within the dotted line, input is an audio file, output is predictions of distinct types of bird vocalizations.

The scope of the project is that the system, here depicted as a purple arrow, should utilize a deep learning model to produce the output. The system should be made for post-processing of recordings, and is therefore not subject to real-time constraints. The outputs should be formatted as a sound event label, the starting time of the sound event (onset), and the end time of the sound event (offset).

1.2 Motivation

The world is always, and has always been undergoing changes, but current scientific consensus tells us with overwhelming confidence that this change is happening faster than what is permissible for it to continue supporting population growth in addition to the population already inhabiting it. One of the bigger crises that's currently underway is the permanent and irreversible loss of biologic diversity. One of the main missions of *Norsk Institutt for Naturforskning* (NINA), is to do research on wildlife to find solutions for the environment that takes this and other aspects of sustainability into account [24]. Software based solutions, like this sound event detector, may allow for some of their more tedious and repetetive workload to be automated. When their more tedious workloads are automated, it will potentially allow NINA to further focus on the essence of their academic work. If this is the case, it may potentially allow NINA to produce more, and better solutions, which may help their mission.

1.2.1 Codebase motivation

The cooperative ongoing research between NTNU and NINA, of which this project is an example of, is likely to continue, and will probably generate multiple projects involving bio-acoustic sound event detection. Given this assumption, it's assessed as a productive endeavor to create an extendable, reusable, configurable codebase, for similar projects, hopefully accelerating the rate of which these kinds of models can be created in addition to increasing the developed models' performance.

1.2.2 Dataset creation motivation

To create a working deep learning model, a dataset has to be generated; through correspondence with NINA, a suitable compromise between their interests and project feasability has been established in the bird species listed in Table 1.

Table 1: Target birds for the project. Norwegian names are listed due to them being used during annotation.

eBird code	English name	Norwegian name
comsnip	Common Snipe	Enkeltbekkasin
whimbr1	Whimbrel	Småspove
eugplo	European Golden Plover	Heilo
woosan	Wood Sandpiper	Grønnstilk

The bird species in Table 1 are selected by NINA due to their interest in further knowledge of the circadian rhythmic behavior of waders. Wader denotes suborder of birds within the charadriiformes type genus, of which members are commonly found along shorelines and mudflats. Waders are in these areas in order to forage for food, thereby the name. Knowledge about the circadian rhythmic behavior of different species is crucial for optimization of sampling procedures for monitoring the population and nesting behavior. Both of these aspects encompass some of the more important research areas of NINA, and a model developed from a dataset on different species within the subspecies, will allow NINA to research these attributes of the species with more ease than earlier.

2 Theory

Both sound event detection and object detection are well established fields both within applications of deep learning and also other, more classical approaches to detection and classification. The following sections represents an attempt to elaborate on some of the theories, methods, and techniques used to approach solutions to some of the problems encountered during the work on this thesis.

In Section 2.1, an introduction to the metrics used for detector evaluation is given. Section 2.2 gives a theoretic basis for some approaches to data agumentation & feature engineering for audio.

In Section 2.3.1, the architecture of a sliding-window-based sound event detection system is elaborated upon, while some key architectural details of the Single Shot Multibox Detector are explained in Section 2.4. Lastly, two of the more recent techniques for feature extraction used within the works of this thesis are presented in Section 2.5.

2.1 Evaluation of detectors

To be able to quantitatively describe the performance of a detector, it is useful to establish some metrics that have been used to evaluate detector performance. This section is dedicated to a thorough, and comprehensive introduction to the metrics discussed in this thesis.

2.1.1 Intersection over Union

Intersection over Union (IoU) [15], also known as Jaccard index or Tanimoto index [16, 38], which is graphically explained for the case of time dimension in Figure 3, is an evaluation metric often used within image detection to decide whether a prediction can be classified as a true or false positive. The conventional method involves setting a threshold value for the IoU between a prediction and any underlying ground truths; if the IoU is below the threshold for all ground truths, the prediction is regarded as a false positive, if it is above, a true positive. A usual IoU threshold for images is 0.5 [10].

Of course, calculation of IoU is only relevant between predictions and ground truths of the same class, as an IoU between a prediction of one class, and a ground truth of another, can not imply a true positive.

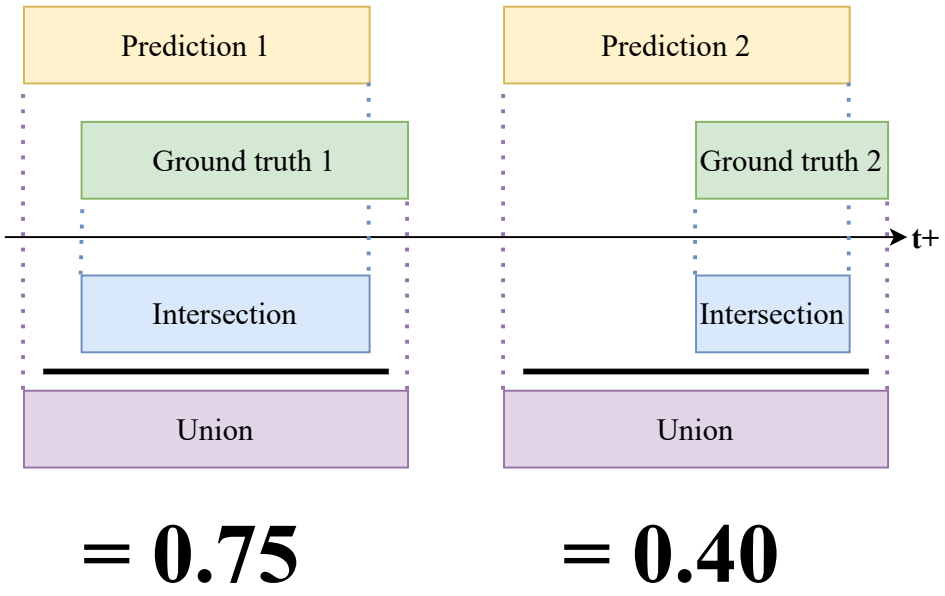

Figure 3: An example of IoU calculations in the time dimension, if the IoU threshold were 0.5 here, Prediction 1 would be a true positive and Prediction 2 would be a false positive.

2.1.2 Average Precision

Average Precision (AP) is an evaluation metric which by itself can describe the relationship between precision and recall within threshold value based detectors, with precision (P) and recall (r) being calculated from the amount of true positives (TP), false positives (FP), and false negatives (FN) by the relationship described in (1) and (2).

$$P = \frac{TP}{TP + FP} \tag{1}$$

$$r = \frac{TP}{TP + FN} \tag{2}$$

Since both precision and recall are dependent on some threshold value used to classify a sample as positive or negative, a relationship between the two is developed by viewing precision as a function of recall where the precision, P, at a given recall value, r is equal to $P(r)$. Given this, average precision in its continuous form is elaborated in (3) with its discrete, numeric counterpart in (4) [43].

$$AP = \int_0^1 P(r)dr \tag{3}$$

$$AP_{_D} = \frac{1}{R}\sum_{r=0}^{R} P(\frac{r}{R}) \tag{4}$$

It is also common practice to use an interpolated value for precision,

$P_{interp}(r) = max_{\hat{r}:R \geqslant \hat{r} \geqslant r} P(\hat{r})$, so that one of the more common implementation of average precision is shown in (5) [10, 32].

$$AP_{D,interp} = \frac{1}{R} \sum_{r=0}^{R} P_{interp}(\frac{r}{R}) \tag{5}$$

To get mean Average Precision (mAP), the average precision for all classes is calculated, yielding mAP as the mean of all classes' average precisions.

2.2 Data augmentation & feature engineering

Feature engineering

Feature engineering can be described as utilization of domain specific knowledge to translate raw data into features that makes solving for a solution from already existing methods feasible. For a more concrete example, one might actually argue that the k-nearest-neighbor algorithm is an example of feature engineering, representing the raw data as coordinates within an n-dimensional euclidean space, finding the classes of the k nearest points to the point in the n-dimensional euclidean space and using the neighbouring classes as a feature to solve the problem of classification, an example of this can be shown in Figure 4.

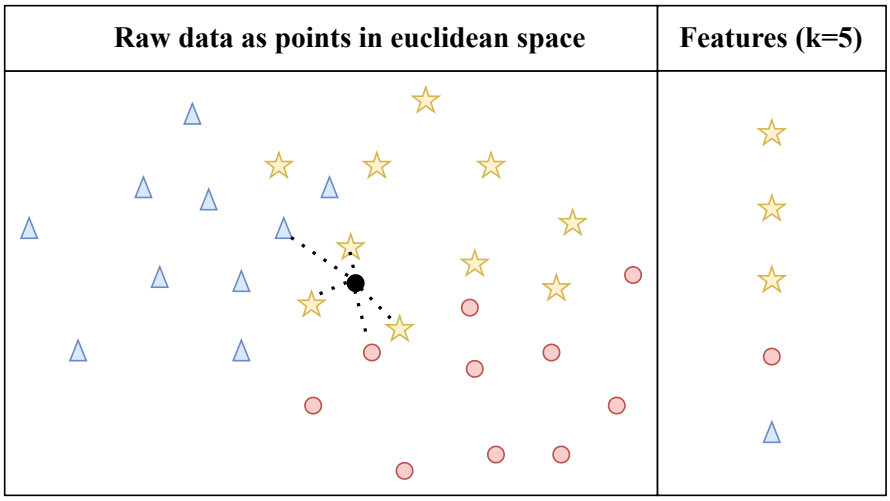

Figure 4: Example of k-nearest-neighbors with k=5 and two dimensions, where the black spot should be classified.

Data augmentation

Data augmentation is a well established method for increasing model generalization through artificially inflating the training dataset [19, 35, 37]. Data augmentation is usually done through adding some sort of randomized alterations that should not affect the final model

outcome [33]. An informal analogy can be found through giving a student a math assignment with random parameters; since the method of solving the problem remains the same, the random parameters forces the student (model) to learn the method for solving the problem (generalization), instead of just cramming the answers (overfitting).

2.2.1 Spectrograms

Creating spectrograms from raw audio data is a feature engineering technique used within sound classification to convert data from the target domain of audio to a source domain of 2-dimensional images [17]. Within applications of deep learning, one of the most common source domains is image classification, so this is a feature engineering method which makes a lot of sense coming from a target domain of audio, even though this isn't a lossless feature convertion, and data is lost in the process.
Configuring the window size, W_{window} and the hop length of the Fast Fourier Transform (FFT), W_{hop} slided along the time axis enables generation of spectrograms with a resolution fitting any image based model input dimensions, as the output width W_{out} and output height H_{out} can be written as (6) and (7), given the input audio signal width, W_{in} [39].

$$W_{out} = \lceil \frac{W_{in} - W_{window}}{W_{hop}} \rceil \tag{6}$$

$$H_{out} = \lfloor \frac{W_{window}}{2} \rfloor + 1 \tag{7}$$

2.2.2 Logscale

Attenuation of acoustic signals usually works as exponential decays over the distance between the signal generator and the signal receptor, dependent on the properties of the medium in which they propagate and the signal's frequency properties [34]. To ensure that the more attenuated signals are represented in a manner that makes them easily detectable, one could therefore refactor the signal strength into a logarithmic scale to effectively represent the attenuation as linear. This input data transformation is quite simple and can be seen in (8), where X is the input data, and $\hat{X}$ is the modified data.

$$\hat{X} = log(X) \tag{8}$$

2.2.3 Standardization

Input signals to the classification model usually has a wide range of input signal strength and noise, sometimes noise that may appear as the event the model is trying to detect. In the case of this project, it may be different bird vocalizations that appear to be the same as the vocalizations the model is trying to detect. If the features of this unknown bird vocalization are similar enough, and the signal strength strong enough, it may be misclassified as an instance of the vocalization that the model is trying to detect. A useful method to aid

the model through this problem, is to standardize the input data [5]. Input data that has been standardized prevents negative ground truths from overwhelming the classifier into a false positive prediction. The standardization transformation can be formally written as (9), where X is the input data, and $\hat{X}$ is the modified data.

$$\hat{X} = \frac{X - \mu_X}{\sigma_X} \tag{9}$$

To clarify a possible misunderstanding due to ambiguous notation usage, in (10), the σ_X, represents the standard deviation of X, not to be confused with the σ used to represent the sigmoid activation function in (12).

2.2.4 Random time shifting

Random time shifting is an established technique for audio data augmentation [22], and it could be described as the audio equivalent of random and resized crops for images [30]. For a detection and classification scheme which incorporates the features of a Single Shot Detector, the time shifting is rather unproblematic, as the annotations consisting of classes with onset & offset of the underlying audio can just be reformatted and forwarded with the shifted time sequence.
However, when random time shifting of multiclassled data with onset and offset annotations is done on data that's supposed to be used to train or validate a multilabel classifier, a problem arises: how much of the randomly selected time series need to contain a sound event of a given class for the window to be considered a true positive instance of the class? To put it in the perspective of this thesis; how much of a given audio sequence needs to contain a bird song for it be considered an audio sequence containing the bird song? This is a problem that will need to be examined further to be able to provide an optimal solution. The intermediate solution presented in this thesis is to set an adjustable IoU threshold that determines whether an underlying time series contains a true ground truth or if it is negative. An example of this scheme can be seen in Figure 5.

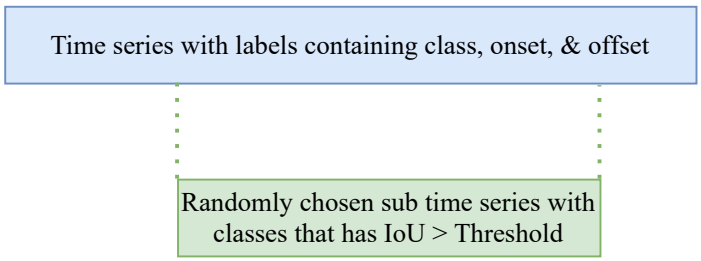

Figure 5: Random timeshift with IoU being over a predetermined threshold implying an active class in the randomly selected time series.

Here, the threshold IoU value is left as an adjustable parameter, since this value doesn't have any easily apparent methods to approximate without experimentation.

2.2.5 Gaussian noise

Adding gaussian noise is a simple, well established method to randomly obfuscate input data so that it appears different before each pass through a model [6]. The gaussian noise added to the input data should also be adjusted to the loudness of the underlying data in the model, so that the whole operation can be written as shown in (10), where X is the initial input data and $\hat{X}$ is the modified data.

To further clarify, in the equation X_i represents an element in a vector X, $\mathcal{N}(0,1)$ represents a normally distributed random variable with standard deviation of 1 and expected value of 0, and $\sigma_{_X}$ is the standard deviation of a vector X.

$$\hat{X}_i = X_i + \mathcal{N}(0,1) \cdot \sigma_{_X} \tag{10}$$

However, by looking at (10), it is clear that the noise added by this transform only has one possible intensity, which is the standard deviation of the input data. It's not a certainty that the standard deviation is a clear cut answer for how much data augmentation is required for optimal training results. A modified version of (10) can be seen in (11), with noise intensity α left as an adjustable parameter.

$$\hat{X}_i = X_i + \mathcal{N}(0,1) \cdot \sigma_{_X} \cdot \alpha \tag{11}$$

2.3 Sliding-window classifier

Sliding-window classification is an established technique for detection and classification of bird vocalizations, and have yielded applicable results in the case of BirdNET [17]. The deep learning aspect of a sliding-window classifier has the exact same architecture as a normal classifier, this implies that the task of training the classifier can be done in the same way as a commonplace audio classification system. The main difference between a sliding-window classifier and a commonplace classifier lies in the inference algorithm, where the inferring task of a normal classifier is to predict the class of a single data point, the task of inferring with a sliding-window classifier is to infer predictions of classes on multiple consecutive data-points, and then utilize these inferred class predictions to make estimates for the onsets and offsets of the classified sound events. The sound event in the case of this thesis, is a bioacoustic sound of a specific pattern, generated by a specific bird species.

2.3.1 Sliding-window classification

Sliding-window classification works by extracting windows of a fixed size, N, from the main audio recording, running a classification scheme on the windows, and then running some functionality to convert these single point predictions into continuous onset & offset predictions. The windows starting points other with a preselected hop sized space. A visual interpretation of the method can be seen in Figure 6.

One of the main disadvantages of this method is that during development, it is difficult to determine the optimal window size for the different temporal aspects of the audio signals that has to be classified [27] (e.g. birds have differing length in vocalization patterns). Window size N, is therefore best suited to be left as a user adjustable parameter, allowing for an experimental approach to estimate the right window size for any given dataset.

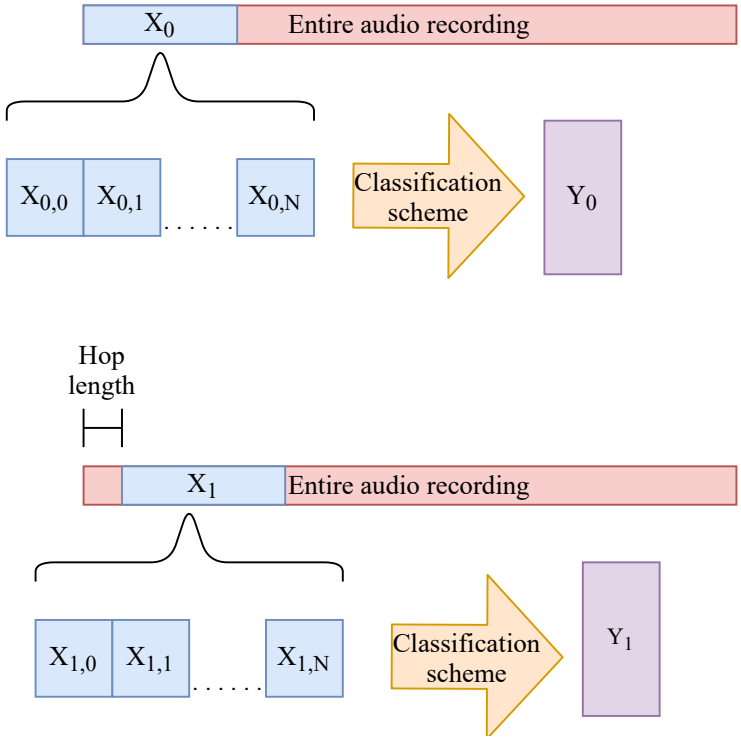

Figure 6: Sliding-window classification, where X_0, and X_1 is, respectively, the first and second window of the main audio recording, with Y_0 and Y_1 as the respective first and second window class predictions from the classification scheme.

2.3.2 Multilabel classification

Different types of sound events one might wish to detect might occur at the same time in the audo recording. To atone for this, the developed classification scheme should support multilabel classification. Multilabel classification can be defined as a set of classification problems where the different classes are not mutually exclusive, meaning several classes may have positive ground truths in the same input instance [7]. By treating the problem as multiple binary relevance problems [42], the models used for multilabel classification can almost have the exact same architecture as models used for mutually exclusive classification, with the exception of the final activation function and the loss function.

Activation function
The conventional activation function used for multilabel classification treated as multiple binary relevance problems is the sigmoid function (12) [7], with a selected plot displayed

in Figure 7.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{12}$$

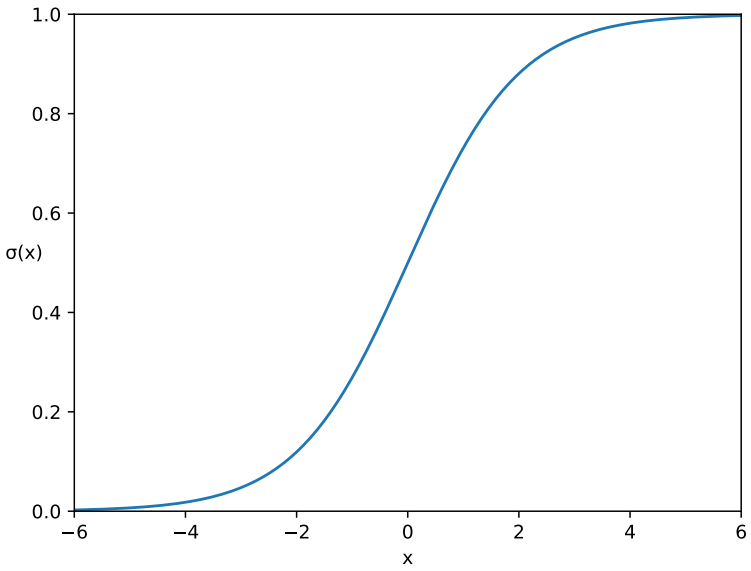

Figure 7: Sigmoid function for $x \in [-6, 6]$.

The reasoning behind using sigmoid for the output is that it has the ability to take any output of a layer, x, given $x \in \mathbb{R}$, at face value and place the output between 0 and 1, which can be translated into a value that represents the model's "confidence" in a given class being a true positive. This is the reason the unprocessed output of the activation function is referred to as "confidence values".

Loss function

The loss function, $\ell(x, y)$, with x, y respectively representing predicted value and actual value, most commonly used for multilabel classification problems solved as multiple binary relevance problems is Binary Cross Entropy loss (BCE) [7]. BCE can be written as (13). By taking the sigmoid activation function (12) into account, the loss function from the unactivated neuron output is written as (14).

$$\ell(x, y) = L = \{\ell_0, ..., \ell_N\}^\top \ , \ \ell_n = -w_n[y_n \cdot log(x_n) + (1 - y_n) \cdot log(1 - x_n)] \tag{13}$$

$$\ell(x, y) = L = \{\ell_0, ..., \ell_N\}^\top \ , \ \ell_n = -w_n[y_n \cdot log(\sigma(x_n)) + (1 - y_n) \cdot log(1 - \sigma(x_n))] \tag{14}$$

In (13) and (14), L represents the vector of loss, one for each of the N predicted classes, w_n is an adjustable weight parameter that can be set to remedy datasets being skewed toward certain classes. The common implementation of the loss function is modeled as (14) since combining the activation function takes advantage of the log-sum-exp trick, making the calculation more numerically stable [2, 12].

2.3.3 Moving mean of class confidence

If the sliding-window classifier described in Section 2.3.1 is applied in combination with multilabel classification, it allows for the different confidence levels that are output of the model to be taken into a moving mean scheme. This scheme will allow for the final predictions to be more precise than otherwise, as a result of the fact that a false positive will require the classifier scheme to make multiple false positive predictions before the final moving mean score is tipped into a false positive classification, vice versa for recall and false negatives. For a visual representation of this concept, see Figure 8.

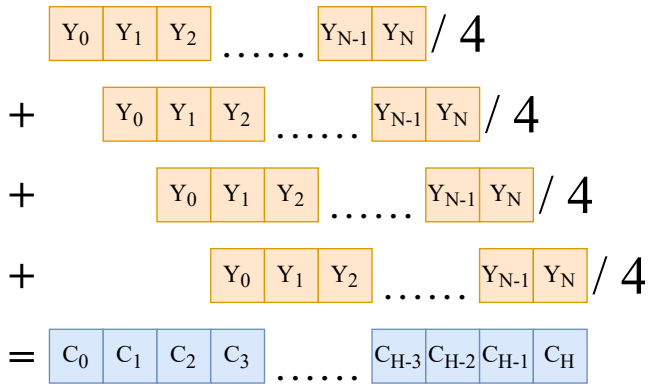

Figure 8: Moving mean confidence value scheme. Hop size here is window size divided by 4, this has the implication that H, representing the length of C, is equal to N+3. N being the amount of hops through the entire record that inference is run on.

Another benefit of the moving mean architecture is that a confidence score is provided for each hop of the classifier, in addition to the offset made by the window size, potentially making the predicted onsets and offsets more accurate. Under the assumption of a perfectly working classifier, the upper limit of the onset and offset time's inaccuracy (ϵ_{max}) can be written as (15), with the error having a uniform distribution. The reason for it having a uniform distribution is that given a random sound event in a window, the difference between the starting point of the window and the starting point of the onset of the sound event will be uniformly distributed, vice versa for offsets.

$$\epsilon_{max} = W_{hop} \tag{15}$$

To add to this, a certain gap will be required between sound events of the same label for them to be considered separate events, and the minimum of this required length, $L_{sep,min}$, assuming a perfectly working classifier will be equal to the width of the classifier window W_{window}. The maximum required gap $L_{sep,max}$ between sound events of the same label will be the width of the hop added to the width of the window. The argument for this is that a perfectly working classifier applied with the sliding-window technique, will have to make a classification on a window without any positive ground truth instances to yield a negative classification. The extracted window in a gap between sound events may be "placed perfectly", meaning the window start is at the offset of a sound event and the window end is at the onset of the next or it may be "placed poorly", where the window hops from a sound event being just barely within the window, to the next sound event

barely being within the window.

$$L_{sep} \in [W_{window}, W_{window} + W_{hop}] \tag{16}$$

After the operation displayed in Figure 8 the confidence scores at the first 3 indices and the last 3 indices will have to be set to a representative mean of the confidence values they are based of, to actually represent the model's confidence of a class being present in the underlying hop sized time window. This technique has been arrived at independently during work on this thesis, but there is little to no doubt that it is subject to multiple discovery as it is neither intricate nor advanced.

2.4 Some architectural details of a Single Shot Multibox Detector (SSD)

The SSD [19] has been a huge contribution to the scene of image based object detection, and because of its almost "plug-and-play"-like architecture with different convolutional backbones and feature extraction schemes, an enormous amount of "spin-off architectures" from the original has been conceived, amounting to the original article having over 15000 citations. A brief overview of the original architecture can be seen in Figure 9.

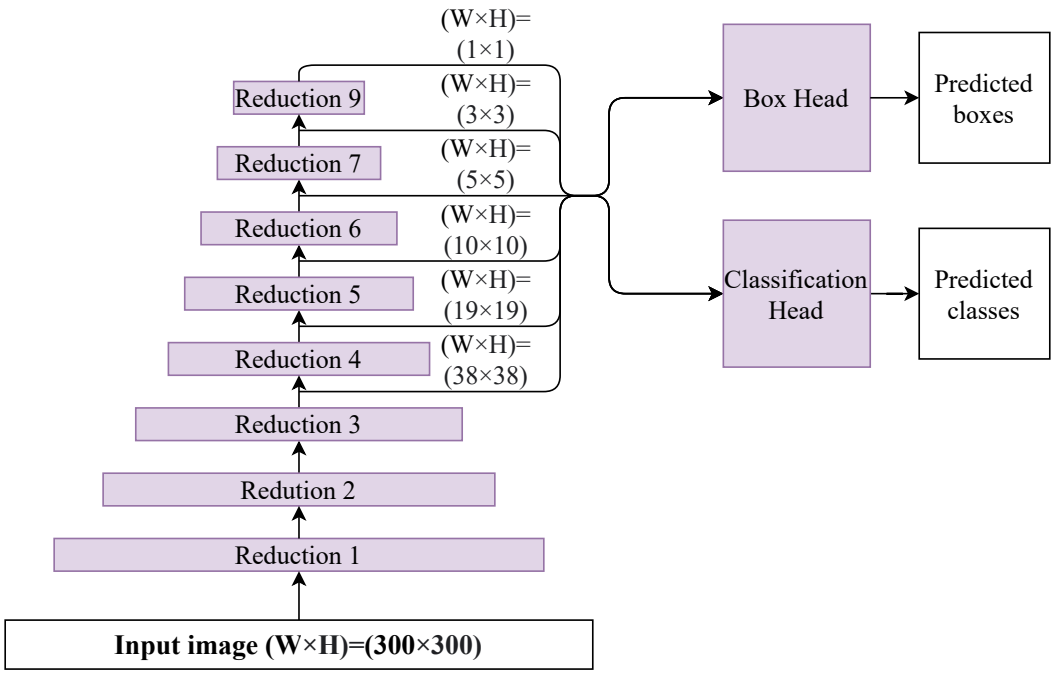

Figure 9: A brief overlook at the architecture of a Single Shot Multibox Detector [19]. The blocks marked as "Reduction" are in reality Convolutional Neural Networks that reduce the height and width dimensions by either strides or lack of padding.

Figure 9 also displays one of the bigger architectural features of the SSD, which is multi-resolution feature maps, which aids detection of objects at multiple scales.

2.4.1 Detector heads

Both the classification head and the box head shown in Figure 9 are in practice convolutional layers, where the classification head produces a confidence map for each class for each pixel in the feature maps, and the box head decides the offsets for the respective bounding box. Figure 10 illustrates the technique of the classification head seen in the forward action of a single class kernel, for a single aspect ratio, for a single box size.

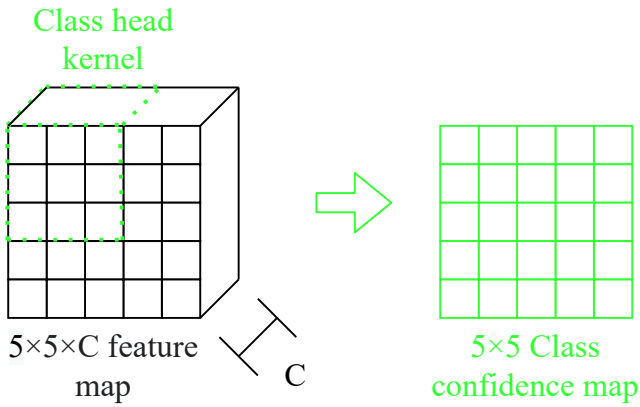

Figure 10: Classification head kernel forward action. C denotes the channel dimension of the input feature map. The output is of the dimension 5×5, due to padding not shown in the figure.

The kernels used for detection of classes in the feature map have a fixed, constant 3×3 size, but feature maps extracted from 3×3-kernels from a high resolution input image might have underlying ground truth objects with different sizes and different aspect ratios. To atone for different aspect ratios and different box sizes, both the box head and the classification head has (#sizes)·(#aspect ratios) kernels for each feature map that is forwarded to it. All in all, this leads to the number of output channels of a classification head being (#sizes)·(#aspect ratios)·(#classes), while the number of output channels for a box head being (#sizes)·(#aspect ratios)·4, where the number 4 is due to the regression of the box center position (x, y) and the box dimensions (width, height).

Since the applied area of this thesis is grounded in sound event detection, where aspect ratios do not exist due to there only being one dimension, time, the thesis implementation doesn't have to take aspect ratios into consideration.

Figure 10 could still be representative for the box head regression, as the method only differ by insted of yielding a class confidence map, it outputs a map of offsets for the placement and dimensions of the bounding boxes. After non-maximum suppression, further described in Section 2.4.2, the classes are matched to their respective box coordinates, giving a final output of mostly non-overlapping boxes with respective class confidence scores. This does implicate that the bounding box coordinates are estimated independently of the underlying classes in the image.
During inference, after non-maximum-suppression, the outputs of class confidences are usually thresholded so that predictions with low class confidences are not deemed as positive predictions.

2.4.2 Non-maximum suppression

With the combined output of the classification head and the box head, what essentially is provided is an enormous amount of predicted bounding boxes combined with their respective class confidence maps. The raw output will therefore ususally contain huge amounts of overlapping bounding boxes for the same ground truth. The postprocessing technique used to fix this is called non-maximum suppression. Non-maximum suppression takes the highest confidence predictions from the unprocessed predictions, removes all unprocessed predictions with an IoU to the aforementoned prediction above a threshold value, and lastly adds the prediction to the output predictions. This filtering continues until there are no more unprocessed predictions left. Python pseudocode for non-maximum suppression can be seen in the snippet below.

```
unprocessed_predictions = unprocessed_predictions.sort(key=confidence)
nms_preds = []
while len(unprocessed_predictions) > 0:
    out_pred = unprocessed_predictions[0]
    new_preds = []
    for pred in unprocessed_predictions:
        if iou(pred, out_pred) > nms_threshold:
            continue
        else:
            new_preds.append(pred)
    unprocessed_predictions = new_preds
    nms_preds.append(out_pred)
```

2.4.3 Hard negative mining

During training, a SSD detector head will be presented with a huge amount of negative ground truths compared to positive ground truths. To present the model with more positive ground truths during training and not starve the model of positive feedback, hard negative mining is utilized [19]. This training strategy works through sorting the negative ground truth predictions by classification loss, generated by the classification head. Then the n predictions with the highest classification loss among these are picked and only these are used for backpropagation.

The number n which decides the number of high loss predictions with negative ground truths is dictated by the number of predictions with positive predictions (b) through the relationship: $n = a \cdot b$. Here a represents a coefficient which in the SSD paper [19], is described through this sentence: ”so that the ratio between the negatives and positives is at most 3:1” $\implies a \leq 3$.

2.5 Feature extraction

Feature extraction through Convolutional Neural Networks (CNNs) is easily argued to be both the metaphorical and literal “backbone” of modern image based deep learning applications, by reducing seemingly complex images into abstract features with high semantic value. Feature extraction from 2D images is where a lot of the research efforts and

creative, ground-breaking work is done within state of the art deep learning these days. Because of this, it is deemed purposeful to take a quick look at a couple of the more recent state of the art techniques, of which both are used in this project.

2.5.1 Compound scaling

EfficientNet introduced one of the most appliable and useful concepts within recent advancements of feature extraction techniques, compound scaling [35]. To shortly summarize, compound scaling introduces a technique to scale up any well working small scale model to fit a given computational budget. The compound scaling technique is illustrated in Figure 11.

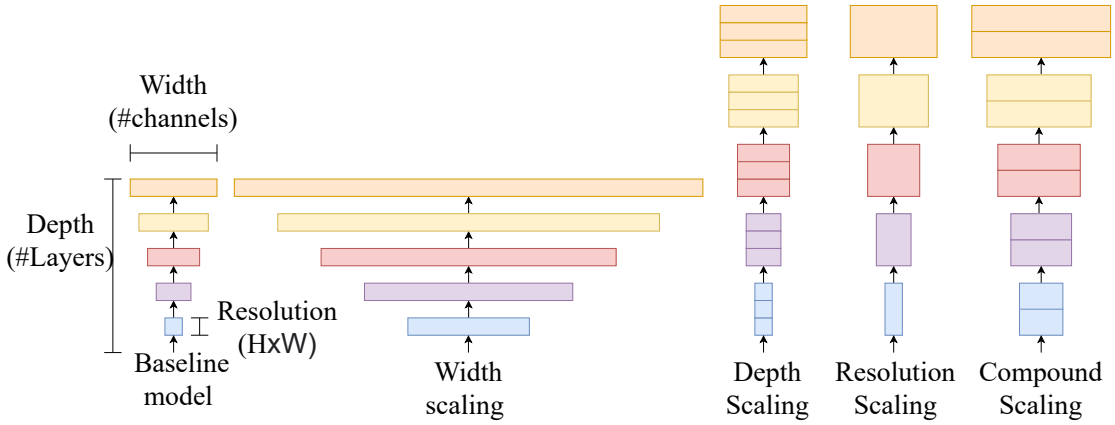

Figure 11: An illustration of the concept of compound scaling, instead of utilizing one of the CNN scaling methods, all are combined.

The question that remains is how to utilize the different scaling techniques in tandem, to achieve optimal compound scaling. The EfficientNet paper presents the following solution:

depth: $d = \alpha^{\phi}$, width: $w = \beta^{\phi}$, resoltuion: $r = \gamma^{\phi}$, with a soft constraint given by $\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$.

With ϕ being the parameter that specifies how much the network should be scaled. The reason for the solution is that depth$\cdot$width$^2\cdot$resolution^2 is proportional to the amount of floating point operations (FLOPS) required for forwarding an image through a 2-dimensional convolutional neural network. Which effectively means that this is a method for scaling a model to any given FLOPS budget. FLOPS scaling can be expressed as $\text{FLOPS} \approx \text{FLOPS}_{\text{Base}} \cdot 2^{\phi}$, with $\text{FLOPS}_{\text{Base}}$ being the FLOPS required by the baseline model.

Through excessive testing of this method with the given constraint, the team behind the paper reported the best results with $\alpha = 1.2$, $\beta = 1.1$, and $\gamma = 1.15$.

2.5.2 BiFPN

Feature Pyramid Networks (FPN) combines the advantages of the high resolution of the more shallow features, with the high semantic value of the deeper features. The assumption is that semantic value have a tendency to increase the deeper you go, but resolution decreases. Theoretically, the original implementations of feature pyramid networks allowed for the features from more shallow layers to contain the same semantic value as the deeper layers, by upscaling the feature maps and adding them to the input to a subsequent convolution [18]. The consept is explained graphically in Figure 12.

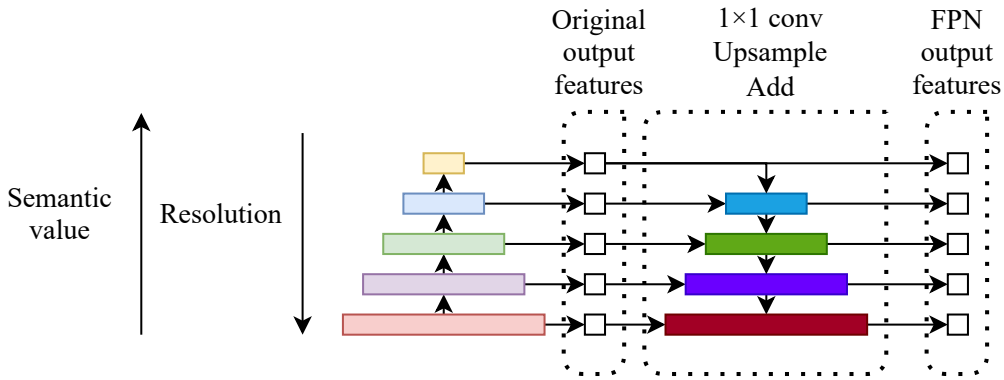

Figure 12: An example of a FPN-network applied with a convolutional backbone.

Several altered implementations of this feature extraction method have been applied within object detection with great results; one of the more recent alterations can be found in the EfficientDet paper, which utilizes an EfficientNet backbone with a FPN consisting of several *BiFPN*-layers [37]. The name is derived from the layers being bi-directional feature pyramid networks, which means that the features are consecutively upscaled and downscaled, potentially allowing for better cooperation between high-resolution, low-semantic features, and high-semantic, low-resolution features. As a visual interpretation can provide more clarity into the technique, it is provided in Figure 13.

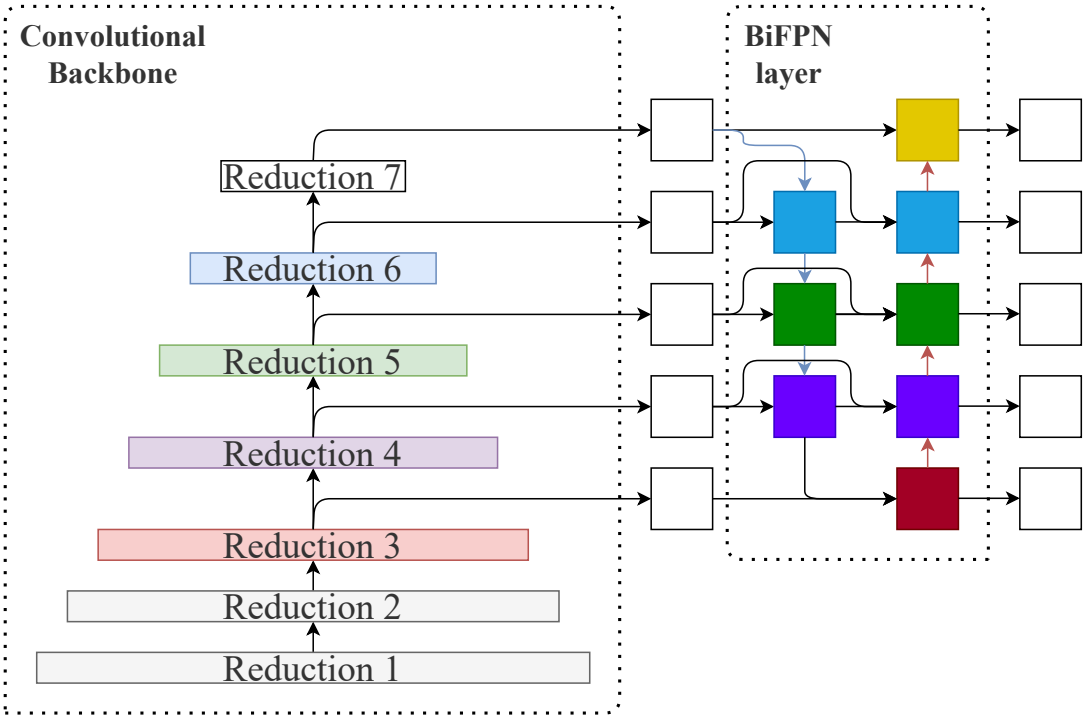

Figure 13: An example of a BiFPN-layer applied with a convolutional backbone.

3 Methodology

This section attempts to describe the process of creating custom state-of-the-art models for sound event detection, from a starting point of unannotated audio data. In Section 3.1, the method for dataset creation for this thesis is described. The section elaborates on dataset annotation, the annotated sound patterns, postprocessing of the audio data & annotations, and implementation of pytorch iterable-style dataset [28] from the processed data.

An approach to productivity focused experimental training setup is described in Section 3.2. Implementation of data augmentation & feature engineering with support for the aforementioned experimental training setup is described in Section 3.3.

Two architectural approaches to sound event detection have been implemented through the work on the thesis one based on the architecture of the SSD, as described in Section 2.4, and one based on a sliding-window approach, as described in Section 2.3. A SSD-based implementation is described in Section 3.4. The dataset primarily postprocessed for the SSD-based approach have been rewritten, and an implementation of a sliding-window based approach, along with the dataset rewriting process is described in Section 3.5.

3.1 Datasets

The datasets used for training and validation have been sourced from recordings contributed by NINA, of which all have been made around Kautokeino, in the Troms & Finnmark-region of Norway, the recording locations can be seen in Figure 14. All recordings have been made with NINA's equipment, in the same season, early summer, which, in combination with the recording locations, bears the implication that the data used for training and validation is sadly not as diverse, population-wise, season-wise, nor equipment-wise, as would be required for a optimally generalized result. However, the main task in this project, model-wise, is to make it suitable for the detection and classifications of the regional bird populations, with the equipment NINA uses, as these are the populations that NINA actively researches and this is the equipment that NINA uses. To investigate and further reflect upon the impact this lack of population, seasonal, and equipment diversity, the test dataset consists of data sourced from user contributions to Xeno Canto, theoretically providing both a population-wise and equipment-wise almost optimal diversity for testing the generalizability across these input data altering factors. The data from Xeno Canto published along the thesis is licensed under the Creative Commons Non-Commercial sharealike license 4.0 [9]. Some of the Xeno Canto data listed in Appendix A has also been published under licensing requiring republication to not have any modifications done to it. Data sourced from these recordings has therefore been omitted in the published test dataset.

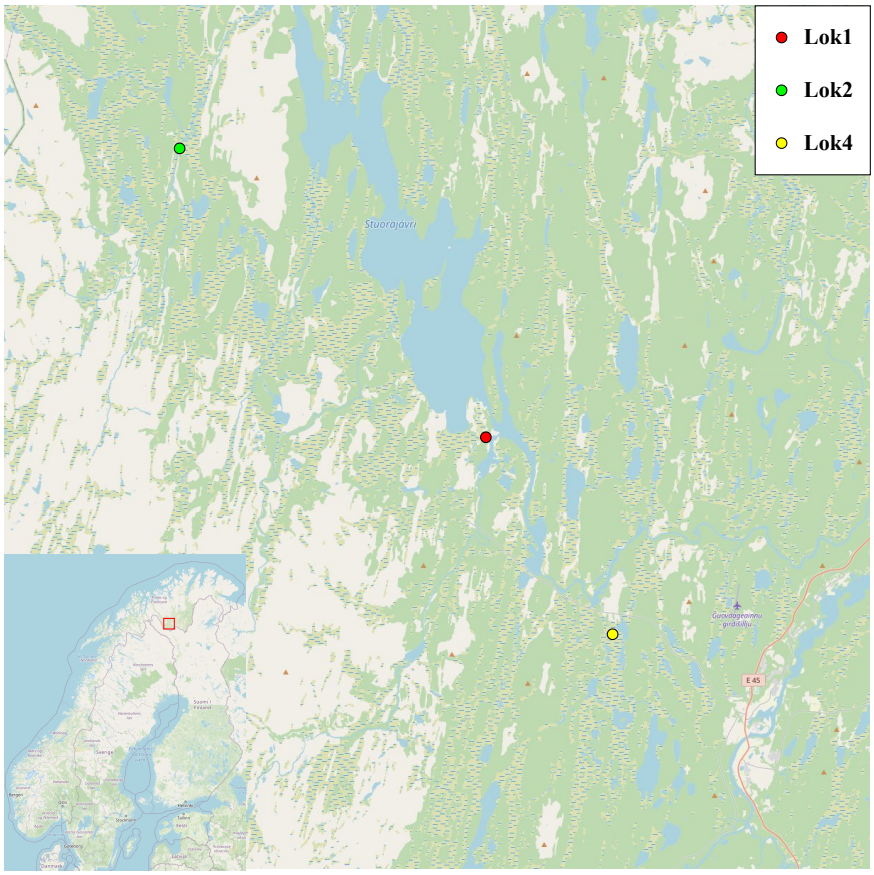

Figure 14: Map of the locations, the maps are provided by Open Street Map [25], and are therefore licensed under CC BY-SA [26]. The locations follow the naming conventions of NINA.

3.1.1 Dataset annotation

The annotations of the different datasets have been performed using a self-described ”active window” method. This method is designed around the purpose of creating audio files of a loosely fixed size, which in practice means it varies from the hard minimum length of 1 minute, up to a soft upper limit of 2 minutes. The soft upper limit is mostly kept in the interest of dataloader speed, since the current design of the dataset requires loading an audio file into memory for each datapoint that is forwarded to the model. Annotations have been manually performed by using Audacity's labelling-functionality, with the possibility to export labels as text-files with a format that easily enables them to be parsed. Parsing is further discussed in Section 3.1.3. The annotations within Audacity are practically performed with a ”BEGIN” label signalling the beginning of an annotated section of the audio file, and a ”END” label signalling the end of this annotated section. A selected screenshot from Audacity providing a visual explanation of this can be seen in Figure 15.

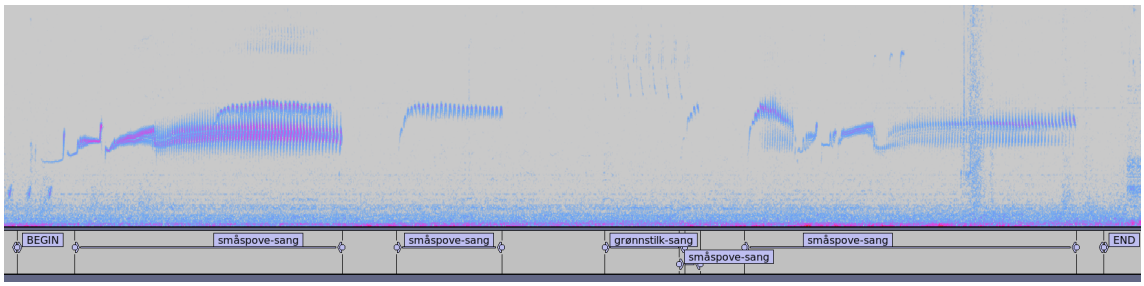

Figure 15: Screen capture from Audacity providing a practical example of the annotation method.

3.1.2 Annotated sound patterns

All the bird species that have been chosen for the dataset have several different sound patterns that they generate, either through vocalization or winnowing. A potential problem with this, is that an untrained ear (or eye, in the case of spectrograms), might not easily distinguish the origins of sounds that are deceptively similar. Consultations with NINA have provided a limited set of sounds, making the job of annotating huge amounts of raw data a feasible task.

Due to the European golden plover having two distinctly different sound patterns that are still distinguishable in the dataset source recordings, it has been split into two different labels. The song vocalization of the European golden plover is not as frequently present as the other sounds, but as the performance of a multilabel classifier implemented as a binary relevance problem is not much affected by having more classes (due to it effectively working as multiple seperate binary classifiers in the final layers), it has still been added as a suitable part of the dataset, as it might give some insight to the effect on the model performance from dataset size. it has Spectrograms of the different sounds can be seen in Figure 16, 17, 18, 19, and 20. These are spectrogram screenshots from Audacity, and includes time as seconds in the x-axis at the top, and frequency as Hz in the y-axis at the left.

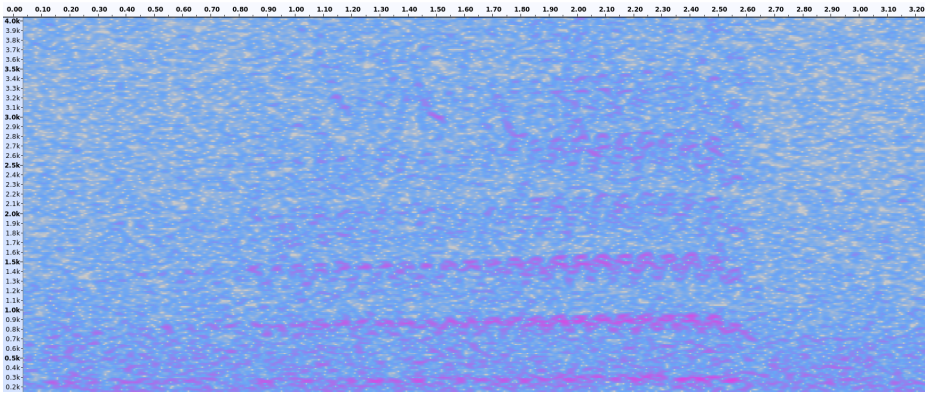

Figure 16: Common snipe winnowing sound spectrogram.

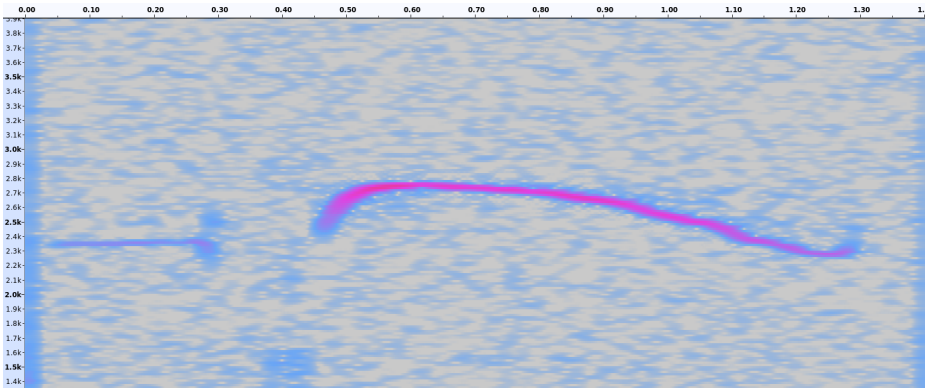

Figure 17: European golden plover call spectrogram.

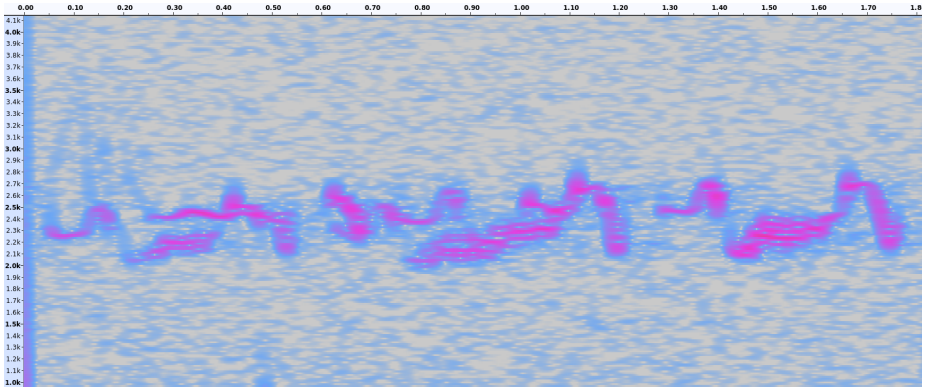

Figure 18: European golden plover song spectrogram.

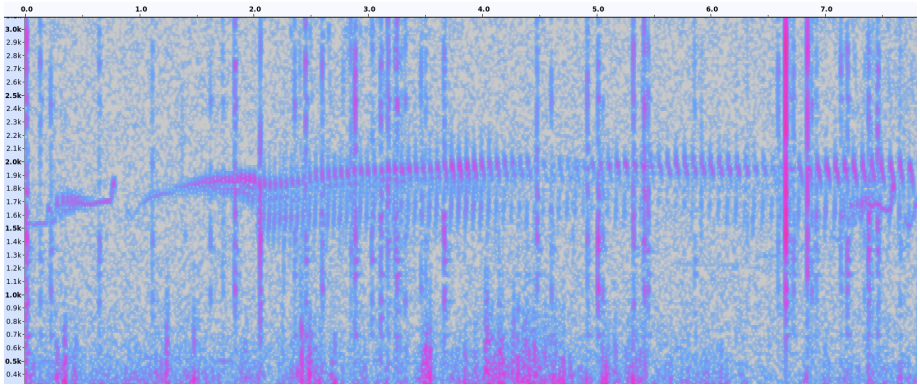

Figure 19: Whimbrel song spectrogram. The noisy lines are induced by rain.

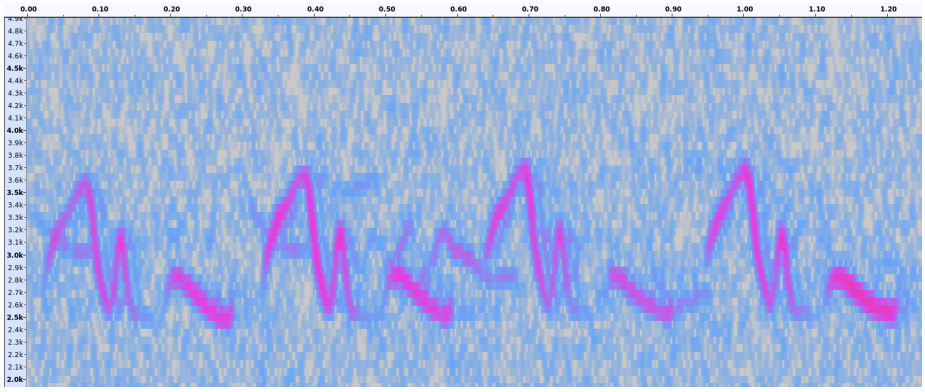

Figure 20: Wood sandpiper song spectrogram.

3.1.3 Dataset generation

After annotation in the fashion mentioned in Section 3.1.1, the annotations and the accompanying audio files needs to be processed into an easily processable format, suitable for a dataset implementation in the fashion of a `torch.util.data.Dataset`-inherited class [28]. For this task, an Audacity label parsing and dataset generating script have been written in python, using the pydub package [31]. The script is listed in Appendix A.1. The main workflow of the script is to first create dictionaries containing all annotations of all audio files through an Audacity-label-parsing function called `parse_labels`. These labels and their accompanying audio-files are then used to create datasets through the `create_dataset`-function, which makes audio files for each ”BEGIN”-”END” labelled in the audio file, and a csv-file for all the annotations in between. The script then generates a pseudorandom uniformly distributed number to decide whether to put it in a directory dedicated to training or a directory dedicated to validation. The chances for it to be put within validation or training is user adjustable through the input-variable for the `create_dataset`-function called `val_amount` which is set to 0.17 by default, which gives a 17% chance that a window is sent to the validation dedicated directory.

3.1.4 Test dataset

As mentioned in Section 3.1, the test dataset have been generated with intent to investigate the generalizability of the sound event detection systems across bird populations and recording equipment, to be used for further discussion. The test dataset has therefore been sourced from user contributions to Xeno Canto [41], an online society for sharing bird sounds, to be able to source bird sounds from across the globe. The dataset was created by finding sounds matching the target sounds discussed in Section 3.1.2, on the site, resampling, mono-sound-transforming, and concatenating them, before applying the methods described in Section 3.1.1 and 3.1.3. For a complete list of the recordings used for the Xeno Canto test dataset, and the creditations for the contributors, see Appendix A.

3.1.5 Multilabel classification dataset

Over a minute long audio files is not a good idea for a dataset that is going to be used for a classifier, as the relatively huge batch sizes may increase loading time, and the audio sequences used for previous sliding-window classification implementations [17] are comparably smaller than what is viable for the SSD-based architecture. Due to this, it was deemed purposeful to recompile the dataset created with the methods described in 3.1.1 and 3.1.3 to 10 seconds intervals. A python script implementing this functionality is listed in appendix A.2, the script works by going through all the previously compiled wav-files and csv-files, by starting at second 0 and sequentially jumping a hop length of 5 seconds and exporting the underlying audio and label into wav- and csv-files, respectively. If the current 10 second interval of audio has no positive ground truth labels, it is skipped in the interest of not overwhelming the classifier with too many samples with no positive underlying ground truths, as hard negative mining, described in Section 2.4.3, has not been implemented for training of the classifier.

It should be mentioned that this method creates some overlap between samples, but not between training, validation, or testing datasets.

3.2 Experimental training setup

Running numerous training setups with little to no differences between different hyperparameters and different user configurable variables can be a hassle and a real time thief, leading to a lot of research efforts being wasted to retype things that a machine could do, manually running experiments and logging the findings of them.

To prevent this, a hyperparameter configuration setup, inspired from Håkon Hukkelås' setup used in the course he's overseeing as an educational assistant, Computer Vision and Deep Learning [14], has been implemented. Taking the setup one step further by automatically generating a selection of experimental configuration yaml-files. The setup is implemented through utilization of Facebooks yacs package (yet another configuration setup), the package is licensed under the Apache 2.0 license [11]. Through yacs' `yacs.config.CfgNode`-class, a default configuration is created, this is done through the script listed in Appendix B. After the default configuration object is created, it is partly overwritten by experimental values listed in a given yaml-file. The functionality overwriting the configuration object is implemented in the train.py-script, listed in Appendix G.1.

To create functionality for automatic generation of such yaml-files pythons PyYaml-package has been used, by loading a default yaml as a dictionary, writing new configurations to the loaded dictionary and dumping the dictionary into a specified yaml-file with a user specified name in a user specified directory. The script implementing this functionality is listed in Appendix C.

Lastly, a bash script has been written to run multiple training configurations sequentially without requiring user intervention, it is listed in Appendix G.2.

3.3 Data augmentation & feature engineering

To enable the data augmentations and the feature engineering to be supported by the experimental setup discussed in 3.2, they have to be implemented as reusable, fully modular code. A practical solution to the problem has been found within the pytorch framework, by implementing all transformations as an inherited class of `torch.nn.Module` [29]. Within the codebase, this is implemented in the transforms.py-file, which can be found in Appendix E. All transformations are initialized through the object of a class named `AudioTransformer` that instantiates all the data augmentation and feature engineering transforms by specification from the `yacs.config.CfgNode` object passed to it. The following sections are brief summaries of the implemented transforms that are used during the work on this thesis, glossing over the method of implementation. All transform implementations can be found listed as their full form in Appendix E. A few other transformations have been implemented, but since they're neither used, nor experimented with in the work of this thesis, they will not be further elaborated on. They are still listed in Appendix E.

3.3.1 Annotation sample formatting

Since multiple of the transformations (e.g. time shifting) require the annotations (consisting of onset, offset, and label) to be reformatted, a transform to convert the annotations from second to sample form have been implemented to ease the task of working with annotations "further down the line". The implementation is not of noteworthy complexity and the following snippet can adequately explain the entirety of it.

```
for idx, annotation in enumerate(lines):
    #lines are lists containing annotation [onset, offset]
    onset = annotation[0]
    offset = annotation[1]
    new_onset = np.ceil(onset*self.sample_rate)
    new_offset = np.floor(offset*self.sample_rate)
    lines[idx] = [new_onset, new_offset]
```

3.3.2 Random time shifting

Random time shifting has been implemented by creating a restricted pseudorandom starting point within a given time series so that the new time series generated is not out of bounds, done with the snippet below.

```
min_start, max_start = (0, x.size()[1] - self.length - 1)
start = np.random.randint(low=min_start, high = max_start)
```

Here, x represents the input time series as a tensor. After this operation the new time series is generated through the narrow method built into `torch.Tensor`, as shown in the line below.

```
x = x.narrow(1, start, self.length)
```

Of course, when time shifting, the underlying ground truth lines and labels have to be shifted and possibly removed if they are out of bounds. The snippet below perform this action. It is worth noting that the functionality assumes the conversion to sample form as described in Section 3.3.1 has been performed.

```
if lines is not None:
    new_lines = []
    new_labels = []
    for idx, annotation in enumerate(lines):
        #Have to deduct starting point from the onset
        annotation_onset = annotation[0] - start
        #End = starting point + annotation length
        annotation_offset = annotation_onset + (annotation[1] - annotation[0])
        #Fix out of bounds issues
        if annotation_offset > self.length:
            annotation_offset = self.length
        if annotation_onset < 0:
            annotation_onset = 0
        if annotation_onset > self.length or annotation_offset < 0:
            continue
        else:
            new_lines.append([annotation_onset, annotation_offset])
            new_labels.append(labels[idx])
    lines = np.zeros((len(new_lines), 2), dtype=np.float32)
    labels = np.zeros((len(new_labels)), dtype=np.int64)
    for idx, new_line in enumerate(new_lines):
        lines[idx] = new_line
        labels[idx] = new_labels[idx]
```

Lines, constisting of onsets & offsets, may be omitted in this, and all other transforms, in the interest of supporting pure, mutually exclusive classification problems, where the transformation done to the data shouldn't affect the ground truth.

3.3.3 Gaussian noise

Random gaussian noise has been implemented through the existing `torch.randn`-method, which generates a tensor full of $\mathcal{N}(0, 1)$-distributed variables in a user requested shape. The following snippet summarizes how noise is added.

```
#noise_factor = std(x) * intensity
noise_factor = x.std() * self.intensity
#x_i + N(0,1) * noise_factor
x += torch.randn(x.size()) * noise_factor
```

The `x.std`-method is a built-in `torch.Tensor`-method for computing the standard deviation of a tensor, and `self.intensity` represents the α as described in Section 2.2.5.

3.3.4 Spectrogram creation

Creating spectrograms of user-defined resolution is suported through the `Spectrify`-class, the class supports multi-channel spectrograms, so that a user may combine mel-, standard-, and log-spectrograms in a requested manner. This is implemented through listing all the transformations in a `torch.nn.ModuleList`, which is required to potentially support computational acceleration. The snippet below summarizes the generation of this spectrogram transform list.

```
for channel in self.channels:
    if channel == "mel":
        out_height = self.height
        if self.crop is not None:
            out_height = self.crop[1]
        hop_size = cfg.LENGTH // self.width
        melify = nn.Sequential(
            torchaudio.transforms.MelSpectrogram(
                                        n_mels=self.height,
                                        hop_length=hop_size
            )
        )
        self.transformations.append(melify)
    if channel == "log" or channel == "normal":
        num_ffts = (self.height - 1)*2 + 1
        hop_size = cfg.LENGTH // self.width
            self.transformations.append(
                torchaudio.transforms.Spectrogram(
                    n_fft = num_ffts,
                    hop_length = hop_size
                )
            )
```

If a user has figured out that the sound events they're set out to detect only span a limited frequency range, a method for frequency cropping is implemented. The frequency cropping is not implemented in the melscale-spectrograms, as the crop done in linear scale would require a mel-scale cropping, which is yet to be implemented.

For the datasets developed for this thesis, a qualitative analysis has found very limited amount of signal features for the target sounds listed in Section 3.1.2, with frequency components above 4 kHz. Due to this, spectrograms are created with a resolution (Height , Width) of (450 , 224), but is cropped to the lower half, to yield a final resolution of (224 , 224), which is the standard resolution for most pretrained convolutional models. In the case of the SSD-based architecture, the classification and bounding line heads are adaptable, so the resolution is set to (512 , 2048), and cropped to (256 , 1028).

```
if not (self.crop is None) and self.channels[i + 1] != "mel":
    channel = torch.narrow(
        input = channel,
        dim = channel.dim() - 2,
        start = self.crop[0],
        length = self.crop[1]
        )
```

If the current channel is a logarithmically scaled channel, logarithmic scaling is done through the snippet listed below.

```
if self.channels[i+1] == "log":
    channel = torch.log(channel)
```

All spectrograms are lastly standardized by the method described in Section 2.2.3 through the snippet below.

```
channel = (channel - torch.mean(channel)) / torch.std(channel)
```

Channels are then concatenated channel-wise by the following snippet, utilizing the method `torch.cat` to concatenate at dimension 0.

```
y = torch.cat((y, channel), 0)
```

The transform also changes the onset and offset coordinates so that they're relative to the output resolution of the transform instead of relative to raw audio sample coordinates, this is done in a similar way as the sample coordinate transform in Section 3.3.1, for the complete annotation transformation, see Appendix E, specifically the class named `Spectrify`.

3.4 SSD-based architecture

An SSD-inspired model has been developed from a starting point of a codebase for an SSD used in TDT4265 [14], in the 4th assignment of the course. The code implementing the architecture could be considered relatively sizable; due to this, and the fact that most of the new functionality are only slight modifications from the starting point, a thorough walkthrough of the SSD-based architecture is deemed excessive. Instead, in this section, glossing over the more intrusive architectural modifications is deemed as sufficient. The codebase for the attempted architecture can be found published on github [4].

3.4.1 Architectural modifications

As most of the codebase in the starting point code [14] revolves around bounding boxes and images, most of the codebase alterations have been to rewrite the functionality to instead support bounding lines and audio files. Some aspects, most significantly the detector heads, required more intrusive architectural modifications, making some assumptions to acommodate for the required completion of the project.

Detector heads
The detector heads no longer need to take height, nor center y-position into account when inferring class confidence or position/dimension regressions. Therefore the heights of the detector head convolutional kernels have been set to the post-padded heights of their

backbone forwarded feature maps. Looking back to Figures 9 and 10, before looking at the diagram of the modified detector head in Figure 21 should be sufficient to acquire a visual conceptualization of the modfication. Figure 21 represents an example of just one size of the forwarded feature maps from the backbone, with multiple output feature map sizes, as shown in Figure 9.

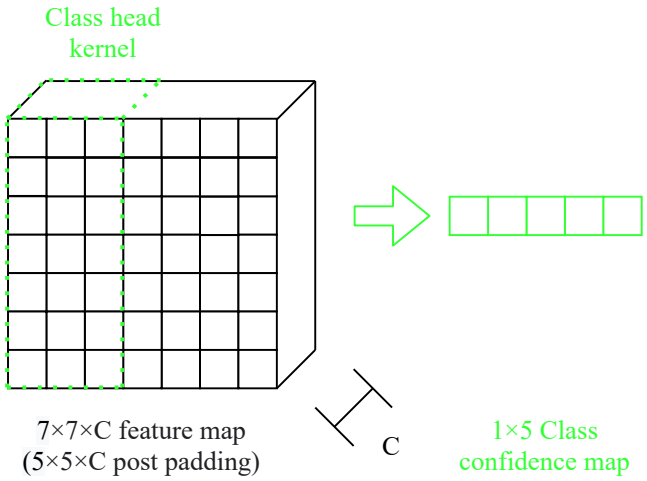

Figure 21: Example displaying modified classification head kernel size for 1D object detection.

Aspect ratios
Instead of taking aspect ratios into account with the method mentioned in Section 2.4.1, the convolutional kernel heads of the modified architecture only takes two different line sizes into account. This makes the output channel count of the so called “line head” equal to 4, as there are two line location parameters offsets to regress (center x, width), and two line sizes to regress for (big and small).

Experimental ground truth class labelling for classification loss
The loss function for the classification utilizes IoU to label the ground truths of predicted boxes as positive or negative. Since sound events tend to be repetetive of nature, an experimental modification of the class labelling functionality for the loss evaluator for the classification head has been made. The difference between the original and the experimental ground truth class labelling method is illustrated in Figure 22.

The experimental classification method takes the fact that sound events usually are repetetive of nature into account by allowing a prediction to be positive as long as it overlaps sufficiently with the ground truth in comparison with the predictions length. This allows for a prediction to still be positive, even when the underlying ground truth is larger than the prediction. As can be seen in Figure 22, this is implemented through dividing the intersection by the prediction and the area of the prediction not covered by ground truth ($\mathrm{P_{NGT}}$). This new unit, dubbed **IoP+** in Figure 22, is then thresholded with the same threshold value as IoU would, with the original ground truth labelling method.

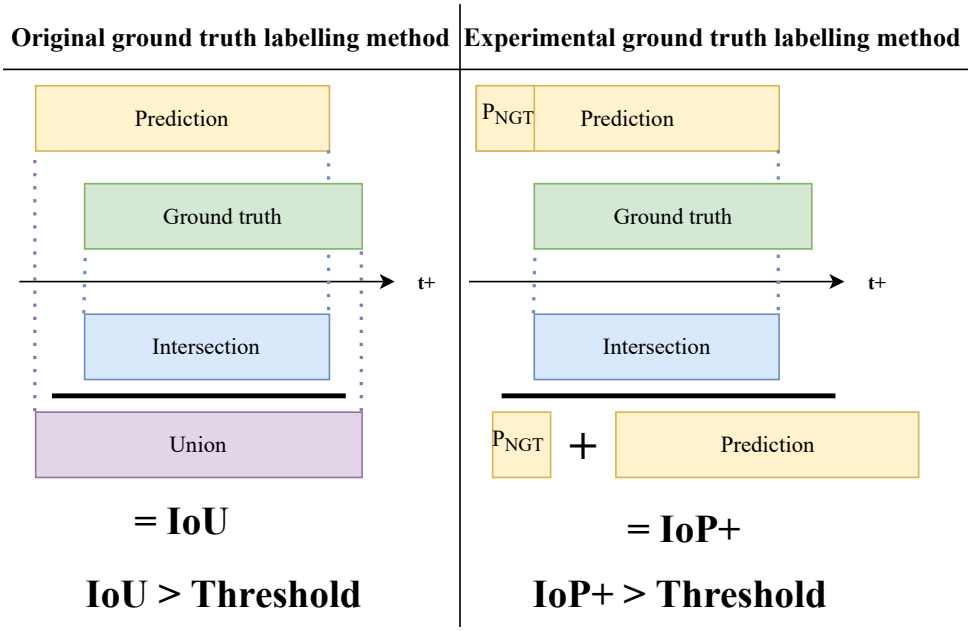

Figure 22: Experimental ground truth class labelling for classification loss, the criteria for a positive class label is listed at the bottom.

NMS IoU threshold
During annotation as described in 3.1.1, any overlapping sound events have been labelled as the same continuous sound event, making overlap between two instances of the same sound event label impossible. Due to this, the non-maximum suppression of the modified architecture is done with an IoU-threshold of 0.

3.4.2 Convolutional backbones

For the convolutional backbone of the SSD-inspired architecture, a combination of EfficientNet [35] and a BiFPN from the EfficientDet-paper [37] has been combined, with 6 repeating BiFPN layers. The EfficentNet-BiFPN backbone utilizes Luke Melas-Kyriazi's implementation of EfficientNet [20], in addition to Zylo117's implementation of BiFPN in an EfficientDet [37] implementation. Additionally, a ResNet50-based backbone has been developed with repeating the last convolutional layer 3 times to generate the smaller scale feature maps for the SSD-based architecture. The backbone implementations are done through already existing implementations of well-known architectures due to this project not revolving around creation and evaluation of new convolutional backbones.

3.4.3 Pytorch dataset implementation

From the method described in Section 3.1.3, audio files and complementary csv-files containing annotations are allocated to their respective directories from the purpose they should serve (training, testing, validation). To effectively utilize this data within the pytorch-based codebase, a pytorch dataset, in the form of a `torch.util.data.Dataset`-inherited class is implemented. The implementation

is listed in Appendix D.3 as the class `Kauto5Cls`. The class works by making a list of audiofiles and annotations, using the `__getitem__`-method to provide the input and the target for a selected indice.

During validation and/or testing, a sliding-window approach with a hop length of the window size divided by two is used to accomodate a thorough run-through through each record contained within the validation and/or test dataset. This approach to validation and testing is meant to represent how the model performs during actual inference, as overlap hopping must be utilized to catch the entirety of every instance of longer lasting sound events.

Implementation-wise, a `validation_crop`-object is added to the `AudioTransformer` instance in the dataset class' `self.transform` if the instance is not a training dataset, allowing for the dataset-class to timeshift at a selected starting point of the time series it forwards to the `validation_crop`-transformation. This yields a `time_series` of the predetermined length that is used during model training with a selected starting point. The following snippet explains the method.

```
audio_filename, start_sec = self.audio_bits[idx]
lines, labels = self._get_annotation(audio_filename)
time_series, _ = torchaudio.load(self.data_dir + "/" + audio_filename)
#ValCrop object initalized in transform when is_train is set false
time_series, lines, labels = self.transform.validation_crop(
                             time_series,
                             start_sec,
                             lines,
                             labels
)
```

Some preprocessing of the dataset class' variables are done to acquire the variables required for the snippet above. The required preprocessing is implemented through the class function called `convert_to_validation`. Since most of the required variables are shown in the previous snippet, the following snippet from `convert_to_validation` might provide further insight to the variable preprocessing.

```
for audio_filename in self.audio_filenames:
    annotations = self.annotation_dict[audio_filename]
    earliest = min(annotations, key=lambda t:t[0])[0]
    latest = max(annotations, key=lambda t:t[1])[1]
    if earliest < 1:
        earliest = 0
    else:
        earliest = earliest - 1.0
    while (earliest + self.audio_bit_length) < latest:
        #Audio bits contain list of (filename, start(sec))
        self.audio_bits.append(
            (audio_filename,
            earliest)
            )
        earliest += (self.audio_bit_length / 2)
    #No need to add latest annotation if it's over before end
    if latest > self.audio_bit_length:
```

```
        self.audio_bits.append(
            (audio_filename,
            latest - self.audio_bit_length)
        )
```

3.4.4 Evaluation

Since no existing evaluation functionality for SSD-based architectures applied on 1-Dimensional problems have been found readily available, a script performing this evaluation have been developed by modifying the script used in TDT4265, assignment 4, task2 [14]. This modified script can be found in Appendix H.2. The script has an implementation that is mostly similar to the method described in Section 2.1.2, but some modifications are required when handling bounding line predictions instead of class predictions.

When verifying bounding lines a matching strategy is required to match processed predictions to the best matching ground truths within the same class. Predictions and ground truths are matched by sorting the potential matches by IoU, then going through this list, keeping the highest match, discarding all other matches that includes the highest IoU-match's prediction or ground truth line, until no potential matches are left. Then IoU threshold is taken into account to classify predictions as true or false positives.

After this is done, the calculation of average precision can be performed as described in Section 2.1.2.

3.5 Sliding-window architecture

A codebase for creating sliding-window based sound event detectors has been developed from a starting point of the code created for the SSD-inspired architecture described in Section 3.4. The final codebase for the *Sliding-window Sound Event Detector* (SSED) is published on github [3]. The classification models used are standard image classification models, using the convolutional backbones of ResNet34, ResNet50, and EfficientNet for compound coefficients from 0 to 7. The code implementing the classifier models can be found in Appendix J. The implementation is based on the ResNet-models within the torchvision framework [40], and EfficientNet models are from Luke Melas-Kyriazi's implementation of it [20]. The following sections aim to elaborate more on the aspects of the sliding-window classifier that has been developed during the work of this thesis.

3.5.1 Sliding-window dataset

A dataset-class for the sliding-window architecture has been developed from a starting point of the dataset for the SSD-inspired architecture, the full implementation of this is listed in Appendix D.1, and the method for creating the actual data is described in Section 3.1.5. The implementation is mostly similar to the SSD-inspired architecture's dataset implementation, except for a transform to convert the lines and labels into binary labels for each input instance. This functionality has been implemented using the method discussed in Section 2.2.4, measuring the IoU of all lines of a label and the input, and

labeling the input as a positive instance of the class if the IoU is over a user-specified threshold. The following snippet from the `TargetTransform`-class should provide some insight into how this has been implemented in practice, the full implementation can be found in Appendix F.

```
out_labels = torch.zeros(self.num_classes)
line_contents = {}
if lines is not None:
    for idx, line in enumerate(lines):
        if labels[idx] not in line_contents:
            line_contents[labels[idx]] = 0
        line_contents[labels[idx]] += (line[1] - line[0]) / self.length
        #Mark data as positive if more than threshold of it is
        ↪  positive.
        if line_contents[labels[idx]] > self.threshold:
            out_labels[labels[idx]] = 1
```

As can be seen in the snippet, the implementation takes into account that multiple instances of the same sound event class should imply that the class is marked as positive. From the perspective of the dataset used in this thesis, it means that if multiple bird sounds of the same type are in the input instance, it is marked as positive if all these bird sounds of the same type add up to be above the threshold.

It is not known whether this method, or one that requires a single sound event to be over the threshold yields optimal results, and it has to be acknowledged that this is a heuristic leap that has been performed to achieve results.

3.5.2 Sliding-window inference

To implement the inference method for the sliding-window architecture as described in Section 2.3.1, a `torch.util.data.Dataset`-inherited `WindowSlide` dataset-class and an inference-script has been created; the complete code for both can be found in Appendix D.2 and I, respectively.

As opposed to the dataset class used during training and validation, this dataset class is instantiated with a single audio file. The reasoning behind this is to optimize inference time, by avoiding loading audio files in-and-out of memory sequentially, but rather loading the files into tensors once, and containing them in memory as class variables. To support any sample rate, in addition to stereo recordings, the dataset has built-in automatic resampling and mono-downmixing. The `WindowSlide`-class has also been impelmented as a iterable style dataset, a snippet displaying the core functionality of the class' `__getitem__`-method is shown below.

```
#Done to not go out of bounds, is consistent with __len__-method
#Get start of this audio tensor slice relative to entire record
audio_bit_start = min(
    self.record.size()[0] - self.input_length,
    idx * self.hop_size
)
```

```
#Extract audio tensor slice
audio_bit = torch.narrow(
    self.record,
    0,
    audio_bit_start,
    self.input_length
)
```

The inference-script is based upon the training script mentioned in Section 3.2, the script in its entirety can be found listed in Appendix I. The script works by taking the configuration-file of a requested pretrained model, and any given number of audio file paths to run inference on as positional arguments. The script then utilize these to instantiate the `Inferencer`-class, which loads the pretrained model from memory and runs through the files sequentially by utilizing an instance the `WindowSlide` dataset-class. When a list of predictions, one for each classifier window hop, is generated, the class implements the method illustrated in Section 2.3.3 through the snippet below.

```
#For loop to make processed preds into moving average
#of raw preds based on number of hops per class window
for hop_no in range(num_hops):
    #Have to add if statement for last hop, since
    #apparantly, there's no elegant way to do this
    if hop_no < (num_hops - 1):
        processed_preds[hop_no:-num_hops + hop_no + 1,:] \
        += all_raw_preds[hop_no:-num_hops + hop_no + 1,:]/num_hops
    else:
        processed_preds[hop_no:,:]\
        += all_raw_preds[hop_no:,:]/num_hops
if num_hops > 1:
    #Need to fix edge cases if hops_per_window > 1

    #Setting up a fractional edge multiplier
    numerator=torch.linspace(1,num_hops-1, num_hops-1)
    denominator = torch.linspace(num_hops-1, 1, num_hops-1)
    edge_multiplier=torch.div(numerator, denominator)

    #To multiply elementwise, it need to have a row for each label
    edge_multiplier=\
    edge_multiplier.view(-1,1).repeat(1,num_labels).view(num_hops-1,num_labels)

    #end_hops += end_hops*[1/(num_hops-1), 2/(num_hops-2),..., (num_hops-1)/1]
    processed_preds[-num_hops + 1:,:]+=\
    torch.mul(
        edge_multiplier,
        processed_preds[-num_hops+1:,:]
        )

    #Flip it around and bring it back (denominator/numerator)(fixing edge at start)
    edge_multiplier = torch.div(denominator, numerator)
    edge_multiplier=\
    edge_multiplier.view(-1,1).repeat(1,num_labels).view(num_hops-1,num_labels)

    #begin_hops += begin_hops*[(num_hops-1)/1,...,2/(num_hops-2),1/(num_hops-1)]
    processed_preds[0:num_hops-1,:]+=\
    torch.mul(
```

```
        edge_multiplier,
        processed_preds[:num_hops-1,:]
    )
```

This implementation solves the edge-case issues by only taking the mean of the available samples for the samples that lie near the edges, as discussed in Section 2.3.3.

3.5.3 Average precision

A method for calculcation of discretized interpolated average precision for multilabel classification ($AP_{D,interp}$) has been implemented, since no reusable functionality for this has been found readily available. The code implementing the functionality as described in Section 2.1.2 is listed in Appendix H.1. The calculation is also made universal with respect to amount of classes, amount of discretized recall values to calculate precision for, and the amount of sampling points R.

4 Results

This section mostly attempts to elaborate upon the results gathered from experiments with both architectures, but also the results from the annotation undertaken during this thesis' work. Section 4.1 displays the results of the SSD-based architecture, while Section 4.2 makes a more thorough presentation of the results of the best performing sliding-window based model.

In Section 4.4, the results of the data gathering done during this thesis' work is presented. Section 4.3 attempts to present the codebase of this thesis' work as a result. The results of the sliding-window based model discussed in Section 4.2 are based on hundreds of experimental training sessions, some of the key insights these produced are presented in Section 4.5.

4.1 SSD-based architecture

The SSD-based architecture has been trained for 10 000 batches for both backbones described in Section 3.4.2, and both class labelling methods described in Section 3.4.1. The resulting mAP, final classification loss, and final line head loss is shown in Table 2, the other, most relevant hyperparameters and configurations used during training are listed in Table 3.

Table 2: Final results for both backbone and both class labelling methods for the SSD-based architecture.

Backbone	Class labelling	mAP	Class head loss	Line head loss
ResNet50	Original	0.029	1.225	0.541
EfficientNet + BiFPN	Original	0.001	3.229	1.051
ResNet50	Experimental	0.011	0.986	1.982
EfficientNet + BiFPN	Experimental	0.011	1.337	1.011

Table 3: Hyper parameters and configurations for the training sessions of the SSD-based architecture.

Hyperparameter-/Config-name	Value/Description
Optimizer	SGD
Momentum	0.9
Weight decay	0.0005
Spectrogram resolution (H, W)	(512, 2048) with Freq crop to (256, 2048)
Start LR	0.001
Learning rate scheduling	Stepdown at iteration 5000, 7500 and 9000
Stepdown gamma	0.1
Random gaussian noise intensity	0.35
Iterations trained for	10000
Batch size	7

To make better sense of the results, debugging of the code have been attempted, yielding no results. The bounding lines and class predictions made by the architecture have been found to be sensical with respect to the method of implementation in the code.

4.2 Sliding-window architecture

Results for all the different experimental training sessions are listed in Section 4.5, this section is dedicated to the results of the best developed classifier model, which is assessed to be the EfficientNet-based model with a compound coefficient $\phi = 7$, trained with a random gaussian noise intensity $\alpha = 0.35$, window size of 2.5 seconds, and a IoU threshold implying positive ground truth of 0.25. The results for all backbones can be seen in Section 4.5.2, the selected backbone has been evaluated as the best backbone due to overall impressive metrics across both validation data and test data, combined with the fact that model selection is not the majorly deciding factor for inference times, as shown in Section 4.5.3.

4.2.1 Quantitatively assessable results

As can be seen in Section 4.5.2, the model reached a mAP of 0.989 on the validation dataset, but this metric can be deceptively non-descriptive, as it is calculated from a class-wise mean of the average precisions. This could bear the implications that a perfectly working model for all classes but one has a low mAP. To further elaborate on this metric, and better visualize the tradeoffs between precision and recall for all classes, the curves for $P_{interp}(r)$, as discussed in Section 2.1.2, for the different classes are displayed in Figures 23, 24, 25, 26, and 27.

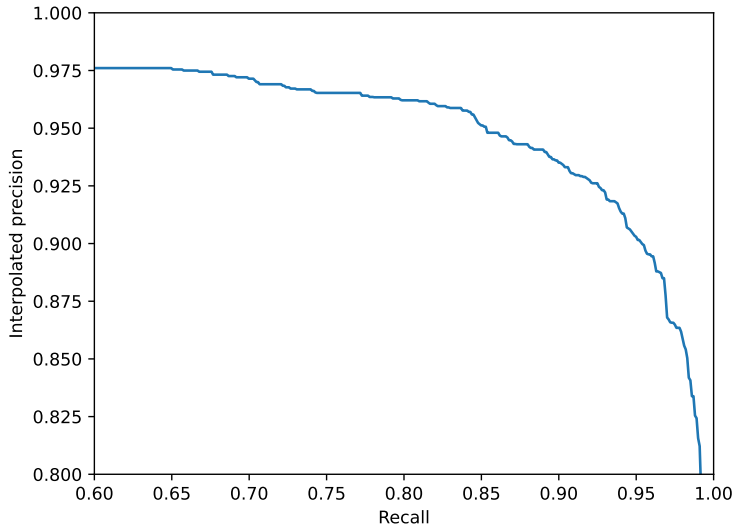

Figure 23: Precision-recall curve for the Wood sandpiper song.

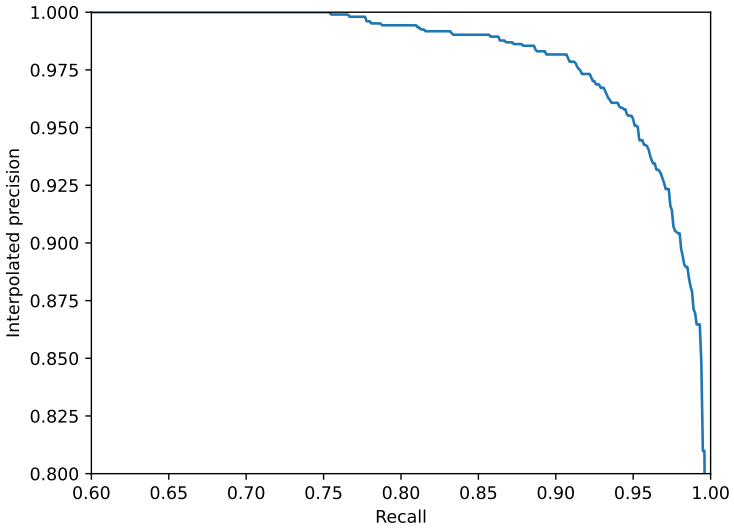

Figure 24: Precision-recall curve for the Common snipe winnowing sound.

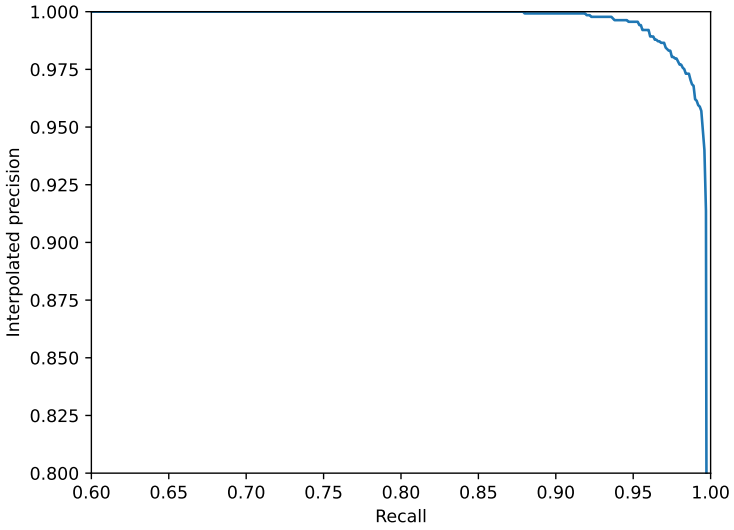

Figure 25: Precision-recall curve for the Whimbrel song.

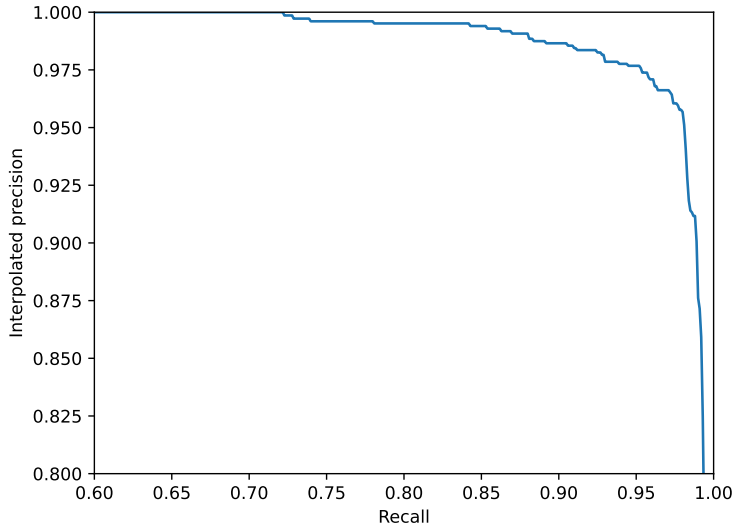

Figure 26: Precision-recall curve for the European golden plover call sound.

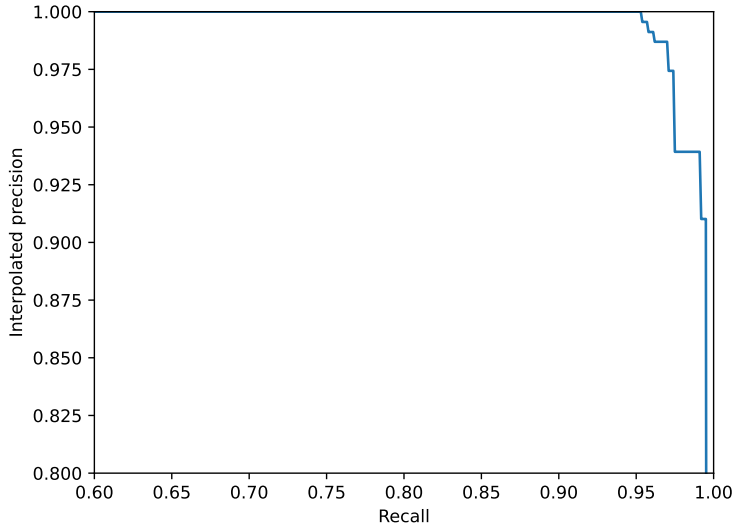

Figure 27: Precision-recall curve for the European golden plover song.

4.2.2 Qualitatively assessable results

To provide a visual representation of the end results, selected screenshots from audacity with labels generated by the inference script in Appendix I is provided in this section. The listed predictions are made on recordings that have not been used for training, validation, or testing.

First, predictions have been inferred with a confidence threshold of 0.76; this value was chosen from the assumption of a perfectly working classifier with 4 hops per window.

Given the assumption, it will require all predicted windows to be positive before classifying a hop as positive. The calculation can be written as $(\frac{1}{\text{hops\_per\_window}} \cdot (\text{hops\_per\_window} - 1) + 0.01)$, where 0.01 is chosen as an arbitrary small number.

A qualitative analysis of the predictions made with a confidence threshold of 0.76 leaves the impression that a dominating majority of predictions have underlying positive ground truths with onset & offset inaccuracies that are below the maximal uncertainty. False negatives for this threshold threshold has not been detected. Through 24 hours of inferred audio, 4 false positives has been identified. An example which gives a qualitative argument for the sensitivity, specificity, and onset- & offset-precision of the detector can be found in Figure 28. It should be mentioned that none of the sound origins has been verified by professionals (i.e. ornithologists), and that this should be taken into account when reviewing the results.

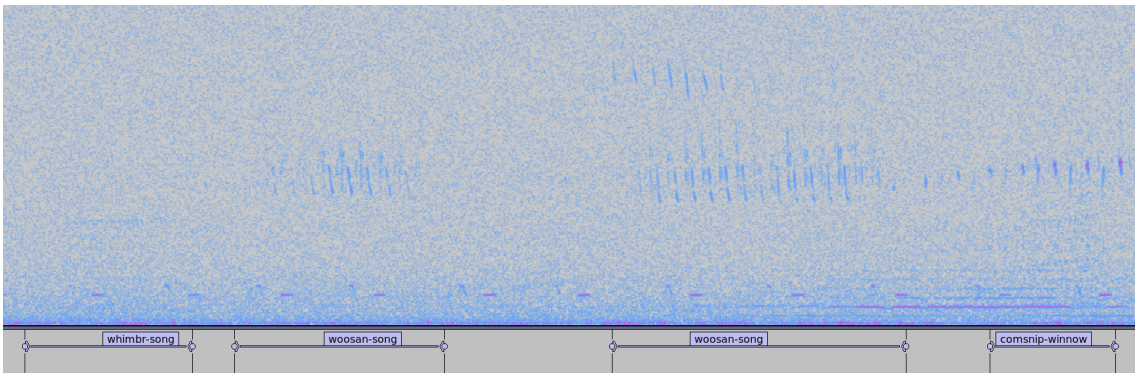

Figure 28: An example of true positive predictions of relatively weak sound events of multiple labels.

As can be seen in Figure 28, the Whimbrel-song is nearly unobservable in the spectrogram, with only what can be described as the main sequence of the vocalization as shown in Figure 19 being present, but it is still detected with relatively precise onset & offset. The same can be said for the Common snipe winnowing sound, although this sound is slightly more easily perceived. Example spectrograms of the sounds are listed in Section 3.1.2, but they are not to scale with the spectrograms listed in this section.

As mentioned in Sections 3.5.1 and 4.5.1, there has to exist a gap between the sound events for them to be considered separate. This has turned out to be an especially present problem for the call vocalization of the European golden plover, as they are often consecutively vocalized. An instance of this problem can be seen in Figure 29.

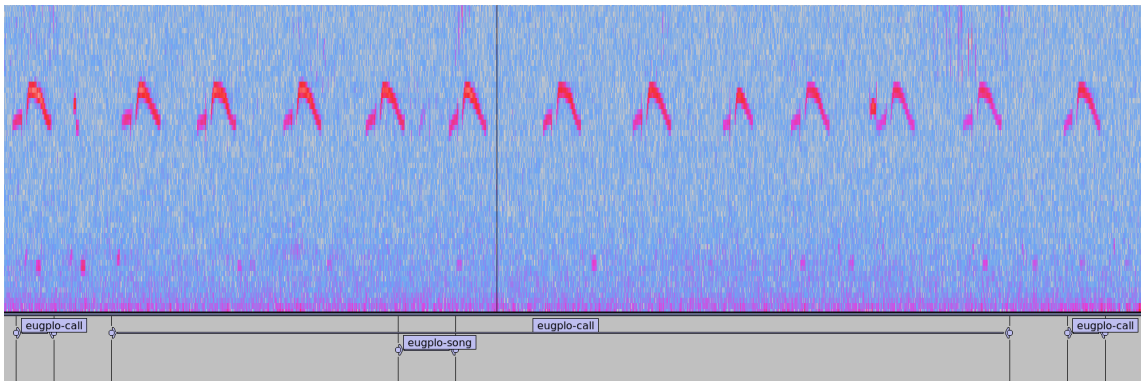

Figure 29: An illustration of the predictions for the European golden plover being "fused" together into a single predicted sound event.

The fusing of multiple sound events, as can be seen in Figure 29 may present a significant problem if an end user of the system is dependent on counting occurences of the sound events. However, NINA have been consulted with regards to this, and for their use, it does not present significant problems. If, for some application, it does, adjusting window size, IoU threshold, and number of hops per window should lead to a detector with better performance with regards to this quality.

Some false positive vocalizations are seen predicted with the confidence threshold of 0.76; although they are few, as only 4 false positives have been observed by going through 24 hours of predicted audio. This amount is not a definite count, as each of the 723 predicted sound events have not been meticulously examined, and none of the predictions have been examined by professionals.

The amount of detected false positive predictions should still be acknowledged. An example of one of the false positive predictions, a prediction for the call-vocalization of European golden plover can therefore be seen in Figure 30.

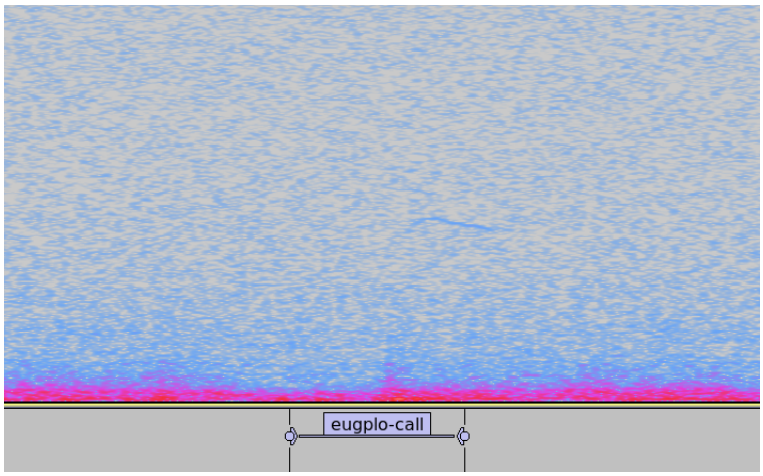

Figure 30: A false positive prediction for the call-vocalization of the European golden plover.

By looking at Figure 28, it is possible to argue that a confidence threshold of 0.76 leads to

a too high sensitivity at the cost of precision. Therefore, inference has also been run with a confidence threshold of 0.95, to be able to present some examples of false negatives, as none have been found with a confidence threshold of 0.76.

Setting the confidence threshold to 0.95 mostly had the effect that the predicted sound events were narrowed, meaning predicted onsets were skewed forward in time, and predicted offsets were skewed backward in time. An example can be seen in Figure 31.

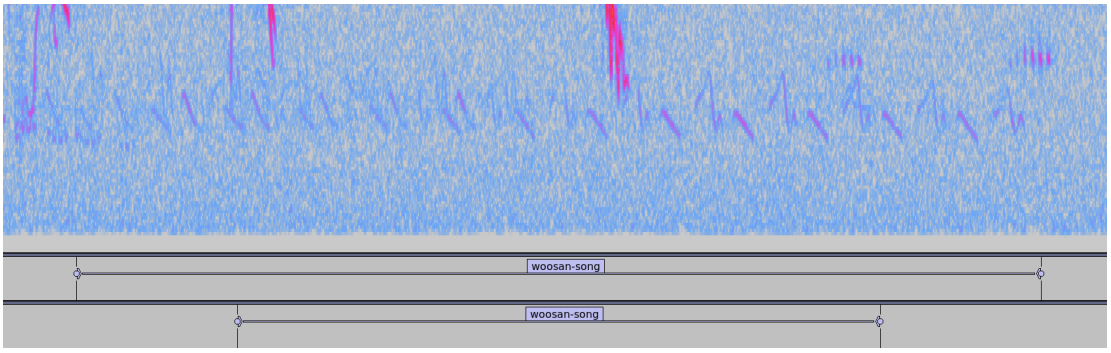

Figure 31: Two predictions of the Wood sandpiper song, the top label line being performed with a confidence threshold of 0.76, while the bottom label line is performed with a confidence threshold of 0.95.

Additionally, some cases of actual false negatives have been found with the confidence threshold of 0.95, one example of a false negative can be seen in Figure 32. However, the amount of false negatives can not be described as completely detrimental to every aspect of the detector's performance. To back this statement up, within the 24 hour example audio file, a confidence threshold of 0.95 only yielded 15.3% fewer predictions than with a confidence threshold of 0.76, down from 723 predictions to 612 predictions. An upside is that no false positives have been identified in the example record for this confidence threshold.

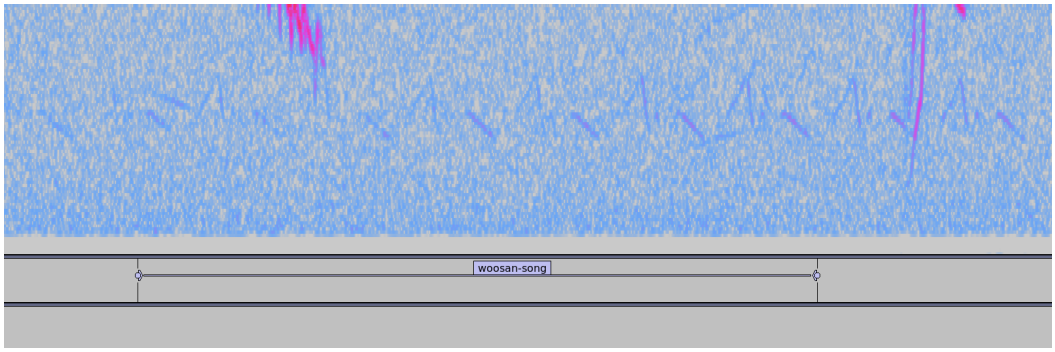

Figure 32: True positive and false negative prediction of the Wood sandpiper song. The top label line contains a true positive from an inference with a confidence threshold of 0.76, the bottom label line contains a false negative from an inference with a confidence threshold of 0.95.

4.3 Codebase

Arguably, the final presented codebase of the ”Sliding-window Sound Event Detector” (SSED) [3] is the main result of the work on this thesis, a lot of reservations are made, to strive for objectivity.

All aspects of the codebasehave been developed with reusability, readability, configuratbility, and extendability in mind.

the codebase for SSED have worked well for the experimental work in this thesis and it is well documented, as almost every function contains a substantial docstring elaborating the user interface. SSED has not been subject to code review of any kind. Limited unit testing have been performed on some parts of the new functionality. The SSED-code is worked out from a codebase used in TDT4265, a course with over a hundred students participating each year, which could be an argument for functionality from the original codebase being thoroughly tested; since this functionality still represents a significant portion of the codebase, it is probably the best quantitative indicator that can be given to speak for its quality.

4.4 Datasets

Two datasets has been developed during the work on this thesis, one for the SSD-based architecture, and one for the sliding-window architecture. The results from the annotations, done in the style described in Section 3.1.1, of the recordings provided by NINA, and the recordings sourced from Xeno Canto, to create the dataset for the SSD-based architecture, are shown in Section 4.4.1. The data in this dataset was later processed into a dataset more suitable for a classifier, the metrics for the resulting classifier dataset is described in Section 4.4.2.

4.4.1 SSD-based architecture dataset

A dataset for a SSD-based architecture for sound event detection has been annotated, both from the recordings provided by NINA and the recordings sourced from Xeno Canto. The sound event count annotated in the recordings NINA provided can be seen in Table 4, with source locations and year of recordings being shown in Table 6. The sound event counts annotated on the test dataset from the Xeno Canto recordings are shown in Table 5.

Table 4: Sounds annotated from NINAs recordings.

eBird code	Vocalization name (Xeno Canto)	Amount of annotations
comsnip	Winnowing	1497
whimbr1	Song	1048
eugplo	Call	1177
eugplo	Song	220
woosan	Song	1798

It has to be acknowledged that the data which has been annotated has not been verified by professionals within the field (i.e. Ornithology). It is therefore plausible that some of the vocalizations are erronously annotated. However, NINA was consulted before annotation was done, in part to atone for this problem.

Table 5: Sounds annotated from Xeno Canto recordings.

eBird code	Vocalization name (Xeno Canto)	Amount of annotations
comsnip	Winnowing	259
whimbr1	Song	69
eugplo	Call	214
eugplo	Song	32
woosan	Song	103

The source locations for the Finnmark data, with regards to the locations shown in Figure 14, is shown in Table 6.

Table 6: Training/Validation data source locations.

Source location (year)	comsnip	whimbr1	eugplo (song)	eugplo (call)	woosan
Lok1 (2017)	170	801	137	773	1245
Lok1 (2018)	955	208	76	278	431
Lok2 (2016)	371	11	0	0	96
Lok4 (2017)	1	28	7	126	26
Total	1497	1048	220	1177	1798

4.4.2 Classifier dataset

From a starting point of the dataset for the SSD-based architecture, the method described in Section 3.1.5 has yielded a dataset suited for training, validating and testing a classifier. The dataset consists of 8237 individual recordings of 10 seconds each. Each recording with a respective csv-file containing the labels, onsets and offsets of the sounds within them. The dataset split ended up with a distribution of label instances shown in Table 7.

Table 7: Amount of sound event labels split into each data group.

Sound event label	Training instances	Validation instances	Testing instances
comsnip	2564	810	630
whimbr1	2329	522	206
eugplo(call)	1807	744	480
eugplo(song)	380	124	88
woosan	3143	844	269

Some of these recordings have overlaps between each other, but there exists no overlap between training, validation, and testing data.

4.5 Experimental findings

As mentioned in Section 4.2, the presented model is a result made from hundreds of experimental training sessions. This section attempts to elaborate on the results that gave some key insights used to develop the presented model.

In Section 4.5.1, the results of a grid search for the optimal window size and positive instance implying IoU threshold is presented. Both ResNet- and EfficientNet-backboned models have been implemented in the codebase, Section 4.5.2 presents the results of training sessions with each different backbone.

Time spent on inference is an important aspect to take into account when applying the developed models from an end user perspective. A thorough evaluation of inference times for each backbone has been performed, and the resulting inference times are presented in Section 4.5.3. As mentioned in Section 2.2.5, the intensity, α, of gaussian noise applied for data augmentation can not be analytically approached, Section 4.5.4.

4.5.1 Window size and IoU thresholding

As mentioned in Section 2.2.4, the IoU threshold of a randomly selected time sequence and an underlying sound event required for it to be assumed positive for a given label, is something that has to be experimented on. As window size and IoU threshold together decide how much time of a vocalization has to be within the window for it to be assumed a true positive, window size has been added as a variable in this experiment. To get a good range of results, 100 different training sessions have been run. The other hyperparameters and configurations for the training sessions are listed in Table 8.

Table 8: Other hyperparameters and configurations for the experiments.

Hyperparameter/configuration description	Value
Model backbone	ResNet50
Optimizer	SGD
Momentum	0.9
Weight decay	0.0005
Start LR	0.03
LR scheduling	Stepdown at epoch 15, 20 and 23
Stepdown gamma	0.1
Random gaussian noise intensity	0.35
Epochs trained for	25
Spectrogram resolution (H, W)	(450, 224) with freq crop to (224, 224)
Batch size	128

The experiments yielded extensive results. To best convey these, a heat map with window sizes in the X-axis, and IoU threshold in the Y-axis can be seen in Figure 33. As the heat map displays, the optimal choice for window size and IoU threshold lies around a window size of 2.5 seconds - 3 seconds, with an IoU threshold of 0.25. These window sizes are the same as the sizes assumed optimal during development of BirdNET [17], in other words, this result can be used as an argument for the assumption made in the paper.

Due to this IoU threshold, 4 hops per classification window has been determined a suitable amount, since that means the inferencer will hop the time required for a positive ground truth for each hop. This, in theory, should implicate that, given a perfectly working classifier, the timing error for the onsets and offsets has an upper limit of 0.625 seconds. Also, because of the heuristic leap mentioned in Section 3.5.1, it means that for sound events to be considered to be separate events, $L_{sep} \in [1.875 \text{ seconds}, 2.5\text{seconds}]$ as described in Section 2.3. These numbers are worked out by replacing W_{window} with the width of the required inactive sound gap for a negative prediction, which can be written as $W_{window} \times (1 - IoU_{thr})$. The reason for this replacement, is that the perfectly working classifier assumed in Section 2.3, was assumed to have an IoU threshold of 0.

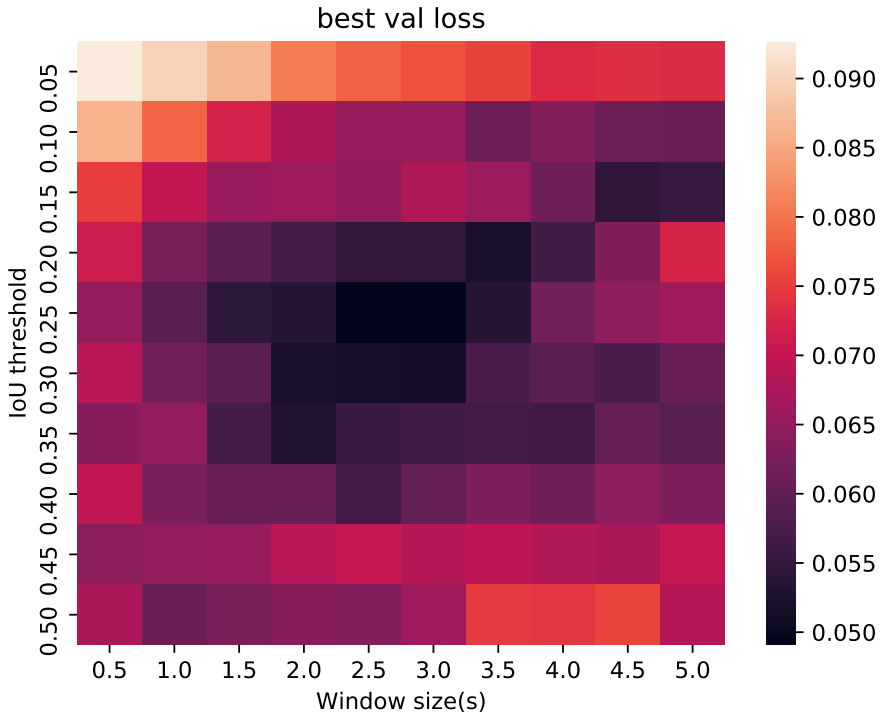

Figure 33: A heat map displaying the smallest validation losses achieved at different values for IoU threshold and window size. Darker/smaller values are better.

4.5.2 Backbone evaluation

Evaluation results for all codebase supported backbones (ResNet34, ResNet50, EfficientNet with compound coefficients from 0 to 7) have been performed with the same training hyperparameters and configurations, which can be found listed in Table 10. The resulting classwise mAP for the validation dataset and the test dataset, and lowest validation loss and test loss, is listed in Table 9, the best results are marked in bold. The reason that the lowest validation loss is used as the evaluation metric, is that the trainer-class in the codebase saves the parameters of the model with the lowest validation loss, and this saved parameter snapshot is what is used when inferencing.

Table 9: Key performance metrics for the tested backbones.

Model name	Lowest val loss	Test loss	mAP (validation)	mAP (test)
ResNet34	**0.046**	0.101	**0.989**	0.957
ResNet50	0.047	0.123	0.988	0.943
EfficientNet ($\phi = 0$)	0.049	0.134	0.986	0.941
EfficientNet ($\phi = 1$)	0.049	0.149	0.987	0.954
EfficientNet ($\phi = 2$)	0.047	0.121	0.988	0.959
EfficientNet ($\phi = 3$)	0.048	**0.089**	0.986	0.960
EfficientNet ($\phi = 4$)	0.050	0.177	0.988	0.936
EfficientNet ($\phi = 5$)	0.048	0.101	**0.989**	0.955
EfficientNet ($\phi = 6$)	0.051	0.174	0.986	0.953
EfficientNet ($\phi = 7$)	**0.046**	0.953	**0.989**	**0.971**

Table 10: Hyperparameters and configurations for backbone evaluation.

Hyperparameter/configuration description	Value/desciption
Optimizer	SGD
Momentum	0.9
Weight decay	0.0005
Start LR	0.03
LR scheduling	Stepdown at epoch 15, 20 and 23
Stepdown gamma	0.1
Random gaussian noise intensity	0.35
Number of epochs	25
Spectrogram resolution (H, W)	(450, 224) with freq crop to (224, 224)
Batch size	128

4.5.3 Inference timing

Running the inference algorithm requires a fixed amount of time for loading the data and performing transformations, before requiring a variable amount of time for the model processing. To further inquire how much the model size affects inference performance, the inference times for each model on a 24 hour long audio file have been gathered, and is listed in Table 11 with the inference setup listed in Table 12. The most relevant aspects of the hardware used during inference is listed in Table 13.

Table 12: Hyperparameters and configurations for inference times.

Hyperparameter/configuration description	Value/Description
Batch size	128
Spectrogram resolution (H, W)	(450, 224) with freq crop to (224, 224)
Window size	2.5 seconds
Hops per window	4

Table 11: Model inference times for a 24 hour long audio file.

Model name	inference time (seconds)
ResNet34	344.6
ResNet50	349.6
EfficientNet ($\phi = 0$)	344.5
EfficientNet ($\phi = 1$)	348.8
EfficientNet ($\phi = 2$)	349.8
EfficientNet ($\phi = 3$)	351.3
EfficientNet ($\phi = 4$)	354.8
EfficientNet ($\phi = 5$)	363.9
EfficientNet ($\phi = 6$)	371.5
EfficientNet ($\phi = 7$)	444.9

Table 13: Hardware description

Component type	Component description
GPU	Nvidia RTX3090
CPU	Intel Core I7-6700k
RAM	Kingston 2 $\times$ 16GB @ 2133MT/s
Storage	INTEL SSDPEKKW512G7

4.5.4 Random Gaussian noise

Random gaussian noise has been applied at 40 different intensity intervals from 0.05 up to 2.0. The best/lowest validation loss achieved for the 40 different training sessions is displayed in Figure 34. The other hyperparameters used during these experiments can be found listed in Table 14.

Table 14: Hyperparameters and configurations for Gaussian noise experiments.

Hyperparameter/configuration description	Value
Model backbone	ResNet50
Optimizer	SGD
Momentum	0.9
Weight decay	0.0005
Start LR	0.03
LR scheduling	Stepdown at epoch 15, 20 and 23
Stepdown gamma	0.1
Epochs trained for	25
Spectrogram resolution (H, W)	(450, 224) with freq crop to (224, 224)
Batch size	128

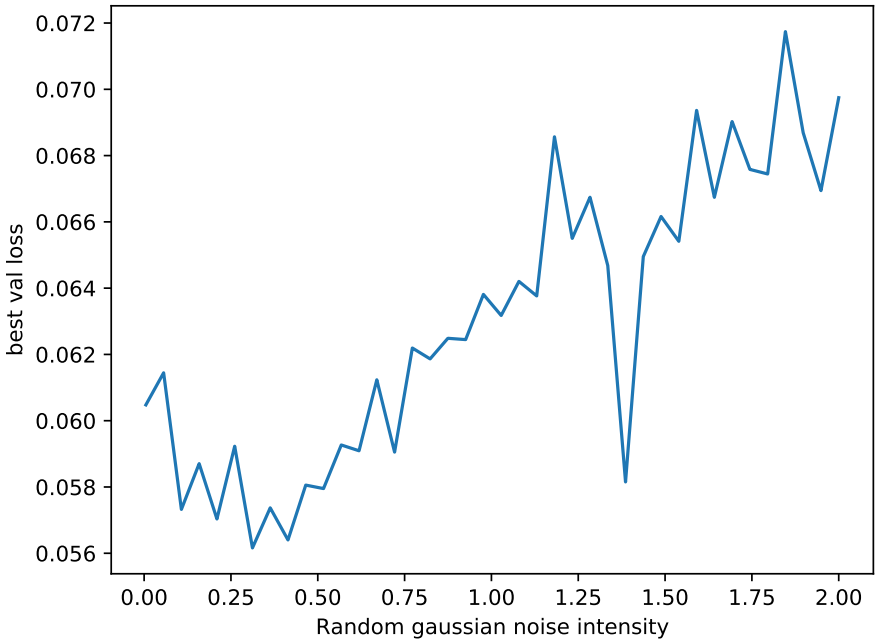

Figure 34: Best validation loss for the 40 different intensity values of the random gaussian noise data augmentation method.

As the graph shows, the validation loss has a best result for the dataset used in this project with an intensity around 0.35, with the exception of an anomaly around 1.40. The reason for the anomaly is unknown, with the configuration yaml-file having been examined. It is most likely an example of lucky pseudorandom number generations.

5 Discussion

5.1 SSD-based architecture dataset

Since the results of the SSD-based architecture has deemed it unsuitable for practical implementation, the SSD-based dataset is most likely only useful as a base for developing more datasets for a sliding-window architecture. This can be done the same way as in Section 3.1.5.

5.2 Sliding-window based architecture dataset

The methodology for creating the dataset for the sliding-window architecture, especially the IoU thresholding to implicate a positive label, is most likely a suitable method for utilizing onset&offset-labelled sound events as a dataset for training a multilabel classifier for sliding-window sound event detection. Although, as mentioned in Section 3.5.1, a heuristic leap has been made with the thresholding technique, and it is probably worth examining whether the assumed best thresholding technique actually provides optimal results.

As can be seen in Section 4.4.2, the datasets used for training have comparable amounts of instances for all classes except for the song of the European golden plover. Even with this minute amount of training data compared to the other classes, the trained model performed comparably well on the validation data for this sound, as can be seen in Section 4.2.1. This bears some implication that similar sound event detection models may not be particularly senstive to amounts of training data with regards to performance.

5.3 SSD-based architecture

As shown in Section 4.1, the SSD-based architecture performed poorly with regards to both precision and recall. Due to this, further development on an inference algorithm for the architecture has not been developed. A possible reason for the failure of the architecture is due to the annotated sound events being repetetive of nature; it is most likely hard to determine how many repetitions of a certain sound event pattern makes it a positive or a negative ground truth instance because of the classification head's ground truth labelling methods. This is only emphasized by the fact that the amount of pattern repetitions will vary widely. If this is the reason for the failure, it probably made the classification head nearly untrainable, yielding only extremely minor improvements to wild guesses for the sound events, as the bounding lines are chosen based on the classification heads class predictions. Of course, there is also a possibility some oversight has been made during development, resulting in non-representative results. However, since the sliding-window architecture has very impressive results, looking further into this will most likely not yield any significant improvements, result-wise.

5.4 Sliding-window architecture

As shown in Section 4.2, the sliding-window approach to sound event detection is a well working technique, even allowing for relatively high-precision onset and offset predictions for the sound events. The window size and the IoU threshold for positive classifications are probably not generalizable for datasets with different sound events, probably not even for datasets containing different bird sounds. The scheme might also, in theory be developed into a system with real time constraints, as spectrogram windows can be extracted in real time and this detection scheme can be run on it, of course with window size setting a hard constraint for the real-time minimum deadlines.

The fast-paced development of mobile solutions for neural processing, e.g. Arm's Ethos N-series [1], in addition to interest in making smaller, more efficient models able to run on such devices [13] may enable non-centralized, real-time solutions for bio-acousitc sound event detection, which could lead to interesting future approaches to bio-acoustic monitoring.

A method for describing the onset- & offset inaccuracies is described in Section 2.3, and in Section 4.5.1 this method is taken into account to approach L_{sep} and ϵ_{max} for the final presented classifier. It still has to be acknowledged that these methods only hold for a perfectly working classifier. In reality, one of the bigger sources for inaccurate classifications may be that the classifier struggles to define the blurry line between a positive and a negative as defined by the labelling method used in this thesis described in Section 3.5.1. In addition, some inaccuracies in the annotations are probably present, as there are no exact clear cut answers to the onset and offset of every sound event, even with attempts to make it as consistent as possible.

If a model developer relies on counting separate sound events, given enough target domain knowledge, these methods can most likely be used for rough estimates for approaching an optimal window size and IoU threshold. But, since both model performance and requirement of separation of sound events have to be taken into account, combining this with a heuristic, qualitative approach of end results will probably be required.

5.5 Test dataset performance

As displayed in Section 4.5.2, the models trained on the NINA dataset performs relatively well on the test dataset developed from user contributions to Xeno Canto. Although there is a penalty on the models for not having been trained on optimally generalized data, this penalty is not sufficient to make the models incapable of generalized applications. This may indicate that generalized data from the target domain is not a strict requirement, if target domain specialists are consulted before model development.

5.6 Codebase as framework for future detector development

The SSED-development codebase [3] that has been written during the work on this thesis is generalizable to most sound event detection problems, possibly making it a valuable

resource for development of future detectors. The work undergone is currently published on github, with NINA having been consulted for the dataset licensing, agreeing to a licensiation under the creative commons non-commercial sharealike license [9] for the datasets derived from their recordings. The datasets are published within the codebase as scripts that automatically downloads the data for an end user. The codebase could easily be extended to support future datasets, and extentions of its own functionality, making it a possible foundation for future work.

6 Conclusion

6.1 SSD-based architecture

An architecture inspired from the SSD-architecture [19] has been developed and evaluated. As shown in Section 4.1 and discussed in Section 5.3, looking further into the SSD-based architecture can not be recommended, as the results from the implementation in this thesis indicates poor performance. A likely cause is the structure of sound events themselves compared to 2D-objects.

6.2 Sliding-window architecture

A sliding-window architecture, loosely resembling the same approach as in BirdNET [17], has been developed and evaluated. This method for sound event detection can be recommended for further use for bio-acoustic purposes, as both the numerical, and the qualitative results shows promise, as can be seen in Section 4.2. Another appealing feature is that the scheme seemingly require very limited amounts of data to train models able to provide state of the art results. This is best exemplified by the relatively few examples of the song-vocalization of the European golden plover; while few, it still yields an almost perfect average precision on the validation dataset.

6.2.1 Codebase

A codebase for developing and applying sliding-window based architectures with relative ease has been developed and utilized during the thesis work. As no readily available codebase for sound event detection resembling this one has been made, further development and utilization of the codebase, published as SSED on github [3], can be recommended for solving future sound event detection problems, if only as a reference baseline solution.

6.3 Recommendation of future work

6.3.1 Datasets

Creating multiple datasets and expanding the codebase for multiple different sound event detection problems could result in a final codebase consisting of multiple ready-to-go sound event detectors, possibly creating a framework for solving all of NINAs sound event detection needs. As mentioned in Section 3.1.2, creating datasets with many more sound events should not impose any great risk to loss of performance, due to how multilabel classification issues treated as multiple binary relevance problems work.

Another interesting aspect to look into is an experimental, incremental approach to gradually increase the number of data-points in the training dataset and see how a trained model performs on a validation dataset of a fixed size. This will effectively yield a trade-

off curve for time spent data-gathering vs. model performance, possibly allowing for an economically optimized approach of data-gathering efforts.

6.3.2 Codebase

Further automation of detector creation
The SSED-codebase [3] has the ability to be further developed into a system that allows for automatic detector creation all the way from combined Audacity label files and audio files, to a finished product as an inferencing detector system providing predictions in a user requested format. Further work on this, could one day lead to all being needed from an end user perspective to develop a state of the art sound event detection system, are data and annotations.

More data augmentation & feature engineering
The `AudioTransformer`-class could be built upon to develop more data augmentation & feature engineering techniques. One possibly interesting technique that has not been implemented is development of phase spectrograms [21], which could give further insight into features that can not be extracted from standard spectrograms. Also including a way to frequency crop the melspectrograms could probably adjust the channel so that the underlying information of a traversing kernel in the first layer is more in line with the other spectrograms. Also, progressive learning, as presented in EfficientNetV2 [36], should probably be developed support for, as this may increase the speed of which highly generalized models can be trained.

Speed optimization
The results of the different inferrence times shown in Section 4.5.3 for the 8 different compound coefficients of EfficientNet bears the implication that model complexity does not have an overwhelming impact on inferrence time, as the execution time is almost constant even with exponential increase in floating point operations for model forwarding [35]. The issue might be within spectrogram creation, as this is a resource intensive process, and it is repeated once for every 0.625 seconds of input audio (window size of 2.5 seconds divided by 4 hops per window). A good starting point for optimization might be to make larger spectrograms and extract windows from it, instead of the current method, where all spectrograms are effectively calculated `hops_per_window` times for every window. Of course, optimizing for speed is probably not of significant interest before real-time constraints are required from an application, or a large scale implementation of this detector approach is implemented. Another approach to speed optimization is to develop 1-dimensional convolutional backbones, with inspirations from the 2D counterparts, as these models will not require computationally intensive transformations before being forwarded to a model. The data augmentation & feature engineering code in the SSED-codebase has some support built-in for 1D approaches, and the rest of the codebase should also be able to acommodate for a 1D approach with minor modifications.

Adding more classifier backbones
Currently, the codebase only have built-in support for ResNet34, ResNet50, and Efficient-

Nets with compound coefficients from 0 to 8. As the field of image based deep learning is developing, continuously adding more model support is probably a good idea. During the work on this thesis, EfficientNetV2 [36] has been presented, with the promise of smaller models and faster training; it is probably worthwhile taking the time to add support for this. As mentioned with regards to speed optimization, adding models with 1-dimensional convolutional backbones could also be of interest.

Web interface for end users

The end users of the detector system will probably be more inclined to use the detectors if it has a better end user interface. Creating a web interface for uploading files, adjusting a few parameters for the detection, (e.g. model selection, confidence threshold, & output format) and sending the predictions per email to the user is most likely going to increase user interaction when compared to a command-line interface. Research results into software systems, no matter how great they are, can be seen as pointless if they are not applicable from an end user perspective, so this is arguably the most productive area to direct future efforts.

Bibliography

[1] Arm. Arm ethos-n series processors. https://developer.arm.com/ip-products/processors/machine-learning/arm-ethos-n, 2021.

[2] torch.nn.BCEWithLogitsLoss. https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html. Accessed: 2021-07-13.

[3] B. Bogfjellmo. Sliding-window sound event detection. https://github.com/bendikbo/SSED. Accessed: 2021-07-17.

[4] B. Bogfjellmo. SRMD - Single Record Multiline Detector. https://github.com/bendikbo/SRMD. Accessed: 2021-07-15.

[5] J. Brownlee. How to use Data Scaling Improve Deep Learning Model Stability and Performance. https://machinelearningmastery.com/how-to-improve-neural-network-stability-and-modeling-performance-with-data-scaling/. Accessed: 2021-07-16.

[6] J. Brownlee. Train Neural Networks With Noise to Reduce Overfitting. https://machinelearningmastery.com/train-neural-networks-with-noise-to-reduce-overfitting/, 2018. Accessed: 2021-07-14.

[7] J. Brownlee. Multi-Label Classification with Deep Learning. https://machinelearningmastery.com/multi-label-classification-with-deep-learning/, 2020. Accessed: 2021-07-14.

[8] C. Commons. Creative commons attribution non-commercial 2.5. https://creativecommons.org/licenses/by-nc/2.5/.

[9] C. Commons. Creative commons sharealike. https://creativecommons.org/licenses/by-nc-sa/4.0/.

[10] M. Everingham, L. V. Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge, 2010.

[11] T. A. S. Foundation. APACHE LICENSE, VERSION 2.0. https://www.apache.org/licenses/LICENSE-2.0. Accessed: 2021-07-15.

[12] Gregory Gundersen. The Log-Sum-Exp Trick. https://gregorygundersen.com/blog/2020/02/09/log-sum-exp/, 2020.

[13] 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*, abs/1704.04861, 2017.

[14] H. Hukkelås. TDT4265 - starter code. https://github.com/hukkelas/TDT4265-StarterCode. Accessed: 2020-12-08.

[15] Residual blocks - Building blocks of ResNet. https://medium.com/analytics-vidhya/iou-intersection-over-union-705a39e7acef. Accessed: 2021-07-13.

[16] P. Jaccard. The distribution of the flora in the alpine zone.1. *New Phytologist*, 11(2):37–50, 1912.

[17] S. Kahl, C. M. Wood, M. Eibl, and H. Klinck. Birdnet: A deep learning solution for avian diversity monitoring. *Ecological Informatics*, 61:101236, 2021.

[18] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection, 2017.

[19] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[20] Luke Melas-Kyriazi. EfficientNet PyTorch. https://github.com/lukemelas/EfficientNet-PyTorch, 2021.

[21] F. Léonard. Phase spectrogram and frequency spectrogram as new diagnostic tools. *Mechanical Systems and Signal Processing*, 21(1):125–137, 2007.

[22] E. Ma. Data Augmentation for Audio. https://medium.com/@makcedward/data-augmentation-for-audio-76912b01fdf6, 2019. Accessed: 2021-07-13.

[23] R. Munroe. tasks. https://xkcd.com/1425/. Accessed: 2020-12-07.

[24] NINA. Om NINA. https://www.nina.no/Om-NINA, 2021. Accessed: 2021-07-17.

[25] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org . https://www.openstreetmap.org, 2017.

[26] Planet dump retrieved from https://planet.osm.org . https://www.openstreetmap.org/copyright, 2017.

[27] H. Phan, P. Koch, M. Maaß, R. Mazur, I. V. McLoughlin, and A. Mertins. What make audio event detection harder than classification? *CoRR*, abs/1612.09089, 2016.

[28] torch.util.data. https://pytorch.org/docs/stable/data.html. Accessed: 2021-07-12.

[29] torch.nn.Module. https://pytorch.org/docs/stable/generated/torch.nn.Module.html. Accessed: 2021-07-12.

[30] torch.transforms.RandomResizedCrop. https://pytorch.org/vision/stable/_modules/torchvision/transforms/transforms.html#RandomResizedCrop. Accessed: 2021-07-16.

[31] J. Robert, M. Webbie, et al. Pydub, 2018.

[32] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., USA, 1986.

[33] C. Shorten and T. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6:1–48, 2019.

[34] T. L. Szabo and J. Wu. A model for longitudinal and shear wave propagation in viscoelastic media. *The Journal of the Acoustical Society of America*, 107(5):2437–2446, 2000.

[35] M. Tan and Q. Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 09–15 Jun 2019.

[36] M. Tan and Q. V. Le. Efficientnetv2: Smaller models and faster training. *CoRR*, abs/2104.00298, 2021.

[37] M. Tan, R. Pang, and Q. V. Le. Efficientdet: Scalable and efficient object detection. *CoRR*, abs/1911.09070, 2019.

[38] T. Tanimoto. *An Elementary Mathematical Theory of Classification and Prediction*. International Business Machines Corporation, 1958.

[39] Torchaudio.transforms.pectrogram. https://pytorch.org/audio/stable/_modules/torchaudio/transforms.html#Spectrogram. accessed: 2021-07-01.

[40] Torchvision. https://pytorch.org/docs/stable/torchvision/index.html. Accessed: 2020-12-07.

[41] Xeno Canto sharing bird sounds from around the world. https://www.xeno-canto.org/. Accessed: 2021-07-03.

[42] M.-L. Zhang, Y.-K. Li, X.-Y. Liu, and X. Geng. Binary relevance for multi-label learning: an overview. *Frontiers of Computer Science*, 12, 11 2017.

[43] M. Zhu. Recall, precision and average precision. 09 2004.

Appendix

A Xeno Canto data information

The test data from Xeno Canto has provided useful insight into how well generalized a model trained on narrowly sourced data is. The individual contribution IDs, listed named of contributors, recording dates, and recording positions are listed in Table 15. The recordings licensed with the condition of no derivative work [**?**] is marked in bold, these files are not included in the published code. The rest of the files are licensed under Creative Commons Non-Commercial Sharealike 4.0 [9].

Table 15: Xeno Canto recordings description.

Record ID	Contributor name	Recording date	Position (latitude, longtitude)
27059	Patrik Åberg	2008-06-21	(68.3728, 18.6987)
27077	Patrik Åberg	2007-06-24	(68.353501, 18.99667)
36348	Herman Van Oosten	2009-06-19	(60.887, 63.7045)
36349	Herman Van Oosten	2009-06-19	(60.887, 63.7045)
36487	Niels Krabbe	2009-07-03	(55.8817, 14.2201)
36489	Peter Woodall	2009-01-04	(-27.543056, 153.051138)
41286	Matthias Feuersenger	2009-10-12	(54.181, 7.8822)
41816	Hans Petter Kristoffersen	2009-05-09	(69.6001, 19.8334)
42344	Patrik Åberg	2009-04-10	(58.5356, 14.2284)
42798	Stuart Fisher	2003-06-14	(57.8995, -6.8495)
57855	Ruud van Beusekom	2010-06-08	(63.4189, -19.039)
57910	Ruud van Beusekom	2010-05-30	(63.8759, -22.7128)
57911	Ruud van Beusekom	2010-06-08	(63.4189, -19.039)
57912	Ruud van Beusekom	2010-06-08	(63.4189, -19.039)
64625	Patrik Åberg	2010-04-20	(58.3714, 14.1395)
76912	Jelmer Poelstra	2011-04-21	(59.581111, 17.914925)
83879	Patrik Åberg	2011-05-30	(65.5501, -17.2334)
83880	Patrik Åberg	2011-05-30	(65.5501, -17.2334)
83881	Patrik Åberg	2011-05-30	(65.5501, -17.2334)
83934	Patrik Åberg	2011-06-05	(63.9209, -20.7639)
83935	Patrik Åberg	2011-05-30	(65.5501, -17.2334)
83937	Patrik Åberg	2011-05-31	(65.5501, -17.2334)
85842	Christoph Bock	2011-07-28	(71.055, 28.044)
102868	Jarek Matusiak	2010-05-03	(57.962, 24.796)
110504	Davyd Betchkal	2012-05-24	(63.741 , -149.583)
129520	Mathias Ritschard	2012-05-26	(53.3667, 22.5667)
129521	Mathias Ritschard	2012-05-26	(53.3667, 22.5667)
135140	Andrew Spencer	2013-05-25	(64.4655, -165.1492)
138422	Fernand DEROUSSEN	1996-06-17	(63.6282, -20.0803)
144493	Davyd Betchkal	2013-05-13	(63.0838, -150.4888)
181110	Elias A. Ryberg	2014-05-27	(62.5896, 10.5599)

190017	Stein Ø. Nilsen	2014-05-10	(69.586, 18.044)
213776	PE Svahn	2014-04-02	(57.7414, 14.0294)
216977	Janne Bruun	1999-06-03	(67.3338, 26.067)
216978	Janne Bruun	1995-06-13	(69.1813, 27.3329)
237879	Eetu Paljakka	2015-04-30	(60.09, 24.4872)
240474	Jarek Matusiak	2015-05-04	(51.3901, 26.8728)
240796	Mikael Litsgård	2015-05-03	(59.5061, 16.4632)
241184	Anon Torimi	2015-05-09	(34.6993, 135.4668)
263922	Terje Kolaas	2015-05-17	(63.3379, 13.4596)
316225	Teet Sirotkin	2016-05-05	(60.402, 18.0382)
318809	Bernard BOUSQUET	2016-05-07	(58.2783, -4.06)
323473	Espen Quinto-Ashman	2016-06-12	(48.8027, 110.0529)
342340	Tero Linjama	2009-05-01	(62.2312, 26.2692)
342367	Tero Linjama	2009-04-29	(53.277, 22.602)
342880	Tero Linjama	2008-06-03	(66.1209, 28.9662)
342909	Tero Linjama	2008-06-04	(65.9432, 29.1853)
354783	Frank Lambert	2015-06-01	(64.7904, -165.2115)
363800	Lars Edenius	2016-07-05	(63.7599, 20.2763)
370121	Krzysztof Deoniziak	2017-05-14	(53.0039, 23.6973)
372395	Albert Lastukhin	2017-05-27	(55.7772, 46.0566)
372396	Albert Lastukhin	2017-05-27	(55.7772, 46.0566)
381969	Jens Kirkeby	2017-06-15	(64.7927, 178.7307)
382143	Jens Kirkeby	2017-06-07	(64.7398, 177.6849)
393693	Anon Torimi	2017-11-19	(35.1519, 136.1098)
400538	Stein Ø. Nilsen	2012-06-29	(68.3104, 23.2813)
414026	Lars Edenius	2018-05-07	(63.7515, 20.3103)
425635	Peter Boesman	2018-06-26	(13.4612, -16.7051)
430928	Albert Lastukhin	2018-06-02	(62.6394, 74.1914)
431019	Albert Lastukhin	2018-06-04	(63.1838, 75.391)
431109	Albert Lastukhin	2018-06-04	(63.1834, 75.3989)
431110	Albert Lastukhin	2018-06-04	(63.1834, 75.3989)
431556	Albert Lastukhin	2018-06-09	(63.0467, 63.0467)
431564	Albert Lastukhin	2018-06-09	(63.0528, 74.552)
431870	Logan McLeod	2018-07-07	(64.9775, -139.0697)
437902	Bram Piot	2018-10-07	(15.865, -16.5121)
446863	David Pennington	2017-08-10	(53.567, -1.8697)
456679	david m	2019-01-08	(54.1212, -0.5416)
471282	guus van duin	2019-05-07	(52.2826, 4.9271)
472029	Stanislas Wroza	2019-04-16	(42.965, 9.4512)
476064	Lars Edenius	2019-05-20	(65.1878, 19.6687)
476067	Lars Edenius	2019-05-21	(65.1878, 19.6687)
476093	Stein Ø. Nilsen	2019-05-18	(70.1843, 19.8821)
478714	Lars Edenius	2019-06-04	(64.3823, 19.6144)
480960	Lars Edenius	2019-06-12	(64.5298, 17.6549)
481806	Lars Edenius	2019-06-16	(64.296, 18.5804)

481809	Lars Edenius	2019-06-16	(64.296, 18.5804)
487713	Stanislas Wroza	2018-07-18	(65.446, -22.1853)
487737	Stanislas Wroza	2018-07-18	(65.446, -22.1853)
487738	Stanislas Wroza	2018-07-18	(65.446, -22.1853)
487742	Stanislas Wroza	2018-07-18	(65.446, -22.1853)
487743	Stanislas Wroza	2018-07-18	(65.446, -22.1853)
490896	Thomas Bergman	2019-06-19	(61.9883, 13.1968)
493129	Albert Lastukhin	2019-05-05	(56.6954, 46.9404)
506558	Lars Edenius	2019-05-21	(65.1878, 19.6687)
519965	Leif Arvidsson	1975-05-00	(58.3217, 13.5522)
519966	Leif Arvidsson	1969-05-14	(58.3217, 13.5522)
526933	James Lidster	2017-05-30	(47.8594, 104.317)
523295	Joost van Bruggen	2020-01-11	(11.9036,-15.5603)
539331	Ireneusz Oleksik	2020-03-28	(50.0832, 18.9544)
545316	Lars Edenius	2019-05-21	(65.1878, 19.6687)
546259	Lars Edenius	2019-06-12	(64.5298, 17.6549)
549245	Isain Contreras Rodríguez	2020-04-21	(25.6036, -109.0521)
549750	Jarek Matusiak	2020-04-23	(52.23, 21.0105)
550220	Uku Paal	2020-04-21	(59.0596, 23.5582)
551341	Jarek Matusiak	2020-04-26	(53.4721, 22.6577)
551729	James Spencer	2020-04-27	(54.0113, -0.3886)
552326	Uku Paal	2020-04-27	(58.3984, 26.3845)
552367	Peter Stronach	2020-04-30	(57.2821, -3.6965)
552599	Vincent Martens	2020-04-30	(52.5061, 6.2538)
554677	Teet Sirotkin	2020-05-06	(59.2168, 18.0724)
557090	Peter Stronach	2020-05-08	(57.2821, -3.6965)
559258	Mats Rellmar	2020-04-23	(57.261, 13.8873)
561288	Mícheál Cowming	2020-05-09	(52.2688, -7.1063)
562056	Thomas Bergman	2020-05-22	(62.1371, 16.2479)
562986	Lars Edenius	2020-05-28	(63.7599, 20.2763)
564393	Lars Edenius	2020-06-01	(65.957, 16.2078)
565297	Lars Edenius	2020-05-26	(65.1761, 18.8739)
587085	Lars Edenius	2020-05-29	(65.1761, 18.8739)
590290	Seth Beaudreault, Toolik Field Station	2020-05-31	(68.6315, -149.5862)
592664	Simon Elliott	1990-05-20	(54.8145, -2.2635)
605533	Niels Van Doninck	2020-06-21	(64.505, 13.7272)
605535	Niels Van Doninck	2020-06-21	(64.505, 13.7272)
605536	Niels Van Doninck	2020-06-21	(64.505, 13.7272)
605551	Niels Van Doninck	2020-06-21	(64.505, 13.7272)
606814	Lars Edenius	2020-05-14	(64.4343, 21.4379)
608282	Lars Edenius	2020-05-14	(65.957, 16.2078)
611785	Ulf Elman	2020-05-05	(59.5207, 17.4029)
621750	Chris Batty	2020-05-03	(53.9299, -2.9833)
627027	Irish Wildlife Sounds	2021-03-02	(51.8857, -8.9977)
631066	Ulf Elman	2020-05-05	(59.5207, 17.4029)

646585	Lars Edenius	2021-05-08	(63.8313, 20.2607)
647337	Lars Edenius	2021-05-08	(63.8313, 20.2607)
647845	Lars Edenius	2021-05-12	(63.8313, 20.2607)
647846	Lars Edenius	2021-05-12	(63.8313, 20.2607)
648421	Irish Wildlife Sounds	2021-04-27	(52.5603, -6.2027)
648435	Irish Wildlife Sounds	2021-04-28	(52.5603, -6.2027)
648515	Irish Wildlife Sounds	2021-04-27	(52.5603, -6.2027)
648533	Thomas Bergman	2021-05-12	(62.1377, 16.2456)
652278	Sławomir Karpicki-Ignatowski	2021-05-17	(53.3697, 22.556)

B Code

A Dataset creation

A.1 Label parsing script

```
#!/usr/bin/env python3
from math import ceil, floor
import numpy as np
from pydub import AudioSegment
from datetime import timedelta
import argparse
import os
import pandas as pd


def parse_labels(label_file_path: str) -> dict:
    """
    Function to parse labels
    Input:
    label_file_path - path to audacity-style label file
    Ouput:
    dict in style of label_dict["class_name"]=(onset, offset, low_freq,
↪   high_freq)
    """
    assert os.path.exists(label_file_path), f"Could not find label
     ↪  file: {label_file_path}\n"
    label_file = open(label_file_path)
    label_lines = label_file.readlines()
    label_file.close()
    label_dict = {}
    for line_no, label_line in enumerate(label_lines):
        #backslash implies it's a frequency range line
        if label_line[0] == "\\":
            continue
        label_vals = label_line.rstrip("\n").split("        ")
        #get class of annotation
        print(label_vals)
```

```
        voc_class = label_vals[2]
        #don't need it in tuple
        del label_vals[2]
        label_vals.extend(label_lines[line_no +
        ↪  1].rstrip("\n")[2:].split("        "))
        #Turn values into floats
        label_vals = [float(val) for val in label_vals]
        #Check if dict already contain annotation of this class
        if voc_class in label_dict:
            label_dict[voc_class].append(tuple(label_vals))
        else:
            label_dict[voc_class] = [tuple(label_vals)]
    return label_dict


def split_to_dirs(
    label_dict: dict,
    wav_file_path: str,
    buffer = 0.0,
    output_path = "output/"
    ):
    """
    Function to create separate folders for each class
    and store audio of each annotation as a separate file
    Inputs:
    label_dict - dict in style of
↪   label_dict["class_name"][index]=(onset, offset, low_freq,
↪   high_freq)
    wav_file_path - path to wav file
    buffer = 0.0 - buffer added before and after annotation in input
↪   file to create output audio segment
    """
    if output_path != "":
        output_path = output_path + "/"
        os.system(f"rm -rf {output_path}")
        os.system(f"mkdir {output_path}")
    #Verify that wav path exists
    assert os.path.exists(wav_file_path), f"Could not find input wav
    ↪  file: {wav_file_path}\n"
    record = AudioSegment.from_wav(wav_file_path)
    for voc_class in label_dict.keys():
        if os.path.exists(output_path + voc_class):
            #Dirty solution to clean directory
            os.system("rm -rf " + output_path + " " + voc_class)
        os.system(f"mkdir {output_path}{voc_class}")
        for voc_no, annotation in enumerate(label_dict[voc_class]):
            #Assuming onset and offset is measured in seconds
            onset, offset, _, _ = annotation
            if onset < buffer:
                onset = 0
            #Recalculating onset and offset to comply with pydub
            if onset < buffer:
                onset = 0
            else:
                onset = int(floor((onset-buffer)*1000))
            if (offset + buffer) > (len(record)/1000):
                offset = len(record) - 1
            else:
```

```
                offset = int(floor((offset+buffer)*1000))
            vocalization = record[onset:offset]

            ↪  vocalization.export(f"{output_path}{voc_class}/{voc_no}.wav",
            ↪  format="wav")


def compound_class_record(
    label_dict: dict,
    wav_file_path: str,
    buffer = 1.0,
    make_label_file = False,
    output_path = "output/",
    crossfade = True
    ):
    """
    Function to create compound wav file for all occurences for each
↪   class
    Inputs:
    label_dict - dict in style of
↪   label_dict["class_name"][index]=(onset, offset, low_freq,
↪   high_freq)
    wav_file_path - path to wav file
    buffer = 0.0 - buffer added before and after
    annotation in input file to create output audio segment
    """
    if output_path != "":
        os.system(f"rm -rf {output_path}")
        os.system(f"mkdir {output_path}")
    assert os.path.exists(wav_file_path), f"Could not find input wav
    ↪  file: {wav_file_path}\n"
    record = AudioSegment.from_wav(wav_file_path)
    for voc_class in label_dict.keys():
        os.system(f"mkdir {output_path}{voc_class}")
        compound_sound = record[0]
        output_labels = ""
        for annotation in label_dict[voc_class]:
            onset, offset, low_freq, high_freq = annotation
            #Recalculating onset and offset to comply with pydub
            if onset < buffer:
                onset = 0
            else:
                onset = int(floor((onset-buffer)*1000))
            if (offset + buffer) > (len(record)/1000):
                offset = len(record) - 1
            else:
                offset = int(floor((offset+buffer)*1000))
            if make_label_file:
                comp_onset = float(len(compound_sound)/1000) + buffer
                comp_offset = float(comp_onset) +
                ↪  float((offset-onset)/1000) - buffer*2
                if crossfade:
                    comp_onset -= min(len(compound_sound)*1000,
                    ↪  buffer/2)
                    comp_offset -= min(len(compound_sound)*1000,
                    ↪  buffer/2)
                comp_onset = str(comp_onset)
                comp_offset = str(comp_offset)
```

```
                low_freq = str(low_freq)
                high_freq = str(high_freq)
                output_labels += \
                "{comp_onset}        {comp_offset}        " + \
                "{voc_class+str(timedelta(seconds=onset//1000))}\n\\"+\
                "{low_freq}        {high_freq}\n"
            vocalization = record[onset:offset]
            compound_sound = compound_sound.append(vocalization,
            ↪  crossfade = min((buffer*1000)//2, len(compound_sound)))

        ↪  compound_sound.export(f"{output_path}{voc_class}/{voc_class}.wav",
        ↪  format = "wav")
        if make_label_file:
            if os.path.exists(f"{output_path}{voc_class}.txt"):
                os.remove(f"{output_path}{voc_class}.txt")
            label_file =
            ↪  open(f"{output_path}{voc_class}/{voc_class}.txt", 'a')
            label_file.write(output_labels)
            label_file.close()


def create_dataset(annotations_dir_path : str, valid_classes : list,
↪  val_amount = 0.17):
    """
    Creates test/val dataset from directory containing directories of
↪  .wav-files and .txt-audacity-label-files
    The .wav-files and the .txt-audacity-label-files must be the same
↪  name as the directory they're in
    For each "BEGIN" and "END" in the label-file, creates a wav file
↪  with it's own csv file
    containing the annotations from the .txt-audacity label file.
    Input:
    annotations_dir_path - relative path to the directory containing
↪  directories of .wav/.txt-annotation combos
    valid_classes - list of strings containing the name of the valid
↪  classes for the detector
    val_amount - amount of validation data in test/val split*

    *time-frames are selected for validation by pseudo-random chance
    """
    #purge/create directories for wav/csv-files
    os.system("rm -rf train \n mkdir train \n rm -rf val \n mkdir val")
    #Annotations_dir_path contains directories with .wav and .txt label
    ↪  files
    annotation_dirs_paths = os.listdir(annotations_dir_path)
    for annotation_path in annotation_dirs_paths:
        wav_path =
        ↪  f"{annotations_dir_path}/{annotation_path}/{annotation_path}.wav"
        label_path =
        ↪  f"{annotations_dir_path}/{annotation_path}/{annotation_path}.txt"
        label_dict = parse_labels(label_file_path=label_path)
        record = AudioSegment.from_wav(wav_path)
        valid_classes.extend(["BEGIN", "END"])
        #Check if an end or a beginning has been forgotten
        if len(label_dict["END"]) != len(label_dict["BEGIN"]):
            print(f"mismatch of timeframe BEGIN and END in
            ↪  {label_path}")
        #Check for erronous labels
```

```
        elif not all(voc_class in valid_classes for voc_class in
        ↪  label_dict):
            print(f"{label_path} contains invalid annotation")
            print("labels in file:")
            for voc_class in label_dict:
                print(voc_class)
        else:
            for time_frame_no, beginning in
            ↪  enumerate(label_dict["BEGIN"]):
                begin_time = beginning[0]
                end_time = label_dict["END"][time_frame_no][0]
                assert (end_time-begin_time) > 60, f"time frame in:
                ↪  {label_path} with less than 60 seconds, starts at:
                ↪  {begin_time}"
                if (end_time - begin_time) > 200:
                    print(f">200 second time frame in {label_path}
                    ↪  starting at {begin_time}")
                #rows in csv output
                rows = []
                for valid_label in valid_classes:
                    #Create list of lists containing [label, onset,
                    ↪  offset] for rows in csv file
                    if valid_label not in label_dict.keys():
                        continue
                    rows.extend([
                        [valid_label, voc[0]-begin_time,
                        ↪  voc[1]-begin_time] for voc in
                        ↪  label_dict[valid_label] if (voc[0] >
                        ↪  begin_time and voc[1] < end_time)
                        ])
                time_frame_record = record[int(floor(begin_time*1000))]
                time_frame_record +=
                ↪  record[int(floor(begin_time*1000))+1 :
                ↪  int(ceil(end_time*1000))]
                df = pd.DataFrame(np.array(rows), columns=["class",
                ↪  "onset", "offset"])
                if np.random.uniform() < val_amount:

                    ↪  df.to_csv(f"val/{annotation_path}_{time_frame_no}.csv")

                    ↪  time_frame_record.export(out_f=f"val/{annotation_path}_{time_
                    ↪  format="wav")
                else:

                    ↪  df.to_csv(f"train/{annotation_path}_{time_frame_no}.csv")

                    ↪  time_frame_record.export(out_f=f"train/{annotation_path}_{time
                    ↪  format="wav")



if __name__ == "__main__":
    create_dataset(
        "composite_test",
        [
            "bekkasinflukt",
            "småspove-sang",
            "grønnstilk-sang",
```

```
            "heilo-pip",
            "heilo-sang"
        ],
        val_amount = 0
        )
```

A.2 Classification dataset generation

```
import pandas as pd
import os
from pydub import AudioSegment

data_dir = "test/"
out_dir = "new_test/"
filenames = os.listdir(data_dir)

audio_filenames = []
for filename in filenames:
    if ".csv" not in (filename.lower()):
        audio_filenames.append(filename)


annotation_dict = {}
for audio_filename in audio_filenames:
    #csv should have same name as audio file except extension
    csv_filename = audio_filename.split(".")[0] + ".csv"
    #Using C engine because it's supposed to be faster, requires
    ↪  delimeter to be commas
    data_frame = pd.read_csv(data_dir + "/" + csv_filename, engine="c")
    #Storing all annotations in dictionary
    annotation_dict[audio_filename]=[]
    for _, row in data_frame.iterrows():
        annotation_dict[audio_filename].append(
                (float(row["onset"]),
                float(row["offset"]),
                str(row["class"]))
            )


for audio_filename in annotation_dict:
    print(audio_filename)
    annotations = annotation_dict[audio_filename]
    new_record_start = 0
    new_record_length = 10*1000
    hop_length = 5 * 1000
    record = AudioSegment.from_wav(data_dir + audio_filename)
    i = 0
    last = False
    while not last:
        #=> last new record from old record
        if (new_record_start + new_record_length) > len(record):
            new_record_start = len(record) - new_record_length
            last = True
        new_record = record[new_record_start:new_record_start +
        ↪  new_record_length]
        new_annotations = []
```

```
        for annotation in annotation_dict[audio_filename]:
            if (annotation[0]*1000) > (new_record_start +
            ↪  new_record_length):
                continue
            elif (annotation[1]*1000) < new_record_start:
                continue
            else:
                onset = (max(annotation[0]*1000, new_record_start) -
                ↪  new_record_start)/1000
                offset = (min(annotation[1]*1000,
                ↪  new_record_start+new_record_length) -
                ↪  new_record_start)/1000
                new_annotations.append((onset,offset,annotation[2]))
        if len(new_annotations) > 0:
            new_filename = audio_filename.split(".")[0] + "_" + str(i)
            print(new_filename)
            i+=1
            new_record.export(out_f=out_dir + new_filename + ".wav",
            ↪  format="wav")
            new_df = pd.DataFrame(new_annotations, columns=["onset",
            ↪  "offset", "class"])
            new_df.to_csv(out_dir + new_filename + ".csv")
```

B Configuration code

```
#yacs is licensed under apache 2.0, which can be found in the
↪  LICENSES-directory.
from yacs.config import CfgNode as CN

cfg = CN()


#Model setup
cfg.MODEL = CN()
# Set any record containing THRESHOLD or more of a class as positive
cfg.MODEL.THRESHOLD = 0.25
cfg.MODEL.NUM_CLASSES = 5


#
↪  ----------------------------------------------------------------------
↪  #
# Model name
#
↪  ----------------------------------------------------------------------
↪  #

cfg.MODEL.NAME = 'efficientnet-b7'

#
↪  ----------------------------------------------------------------------
#
#
↪  ----------------------------------------------------------------------
#Dataset setup
```

```
cfg.INPUT = CN()
cfg.INPUT.RECORD_LENGTH = 40960
cfg.INPUT.SAMPLE_FREQ = 16000
cfg.INPUT.NAME = "kauto5cls"
#Used in case of spectrogram
cfg.INPUT.IMAGE_SIZE = [224, 224]

#Basic stuff
cfg.INPUT.TRANSFORM = CN()
cfg.INPUT.TRANSFORM.SAMPLE_COORDS = CN()
cfg.INPUT.TRANSFORM.SAMPLE_COORDS.ACTIVE = True
cfg.INPUT.TRANSFORM.CROP = CN()
cfg.INPUT.TRANSFORM.CROP.ACTIVE = True
cfg.INPUT.TRANSFORM.LENGTH = 40960

#Alright config for spectrogram
cfg.INPUT.TRANSFORM.SPECTROGRAM = CN()
cfg.INPUT.TRANSFORM.SPECTROGRAM.ACTIVE = True
cfg.INPUT.TRANSFORM.SPECTROGRAM.RESOLUTION = [224, 450]#OUTPUT
↪  RESOLUTION
cfg.INPUT.TRANSFORM.SPECTROGRAM.FREQ_CROP = [12, 224] #[BOTTOM_CROP,
↪  HEIGHT]
cfg.INPUT.TRANSFORM.SPECTROGRAM.CHANNELS = ["normal", "mel", "log"]

#Random gaussian noise with random intensity
cfg.INPUT.TRANSFORM.RAND_GAUSS = CN()
cfg.INPUT.TRANSFORM.RAND_GAUSS.ACTIVE = False
cfg.INPUT.TRANSFORM.RAND_GAUSS.INTENSITY = 0.35
cfg.INPUT.TRANSFORM.RAND_GAUSS.RAND = True
cfg.INPUT.TRANSFORM.CHANCE = 1

#Random "flip"
cfg.INPUT.TRANSFORM.RAND_FLIP = CN()
cfg.INPUT.TRANSFORM.RAND_FLIP.ACTIVE = False
cfg.INPUT.TRANSFORM.RAND_FLIP.CHANCE = 0.5

#Random amplification/attenuation
cfg.INPUT.TRANSFORM.RAND_AMP_ATT = CN()
cfg.INPUT.TRANSFORM.RAND_AMP_ATT.ACTIVE = False
cfg.INPUT.TRANSFORM.RAND_AMP_ATT.FACTOR = 5
cfg.INPUT.TRANSFORM.RAND_AMP_ATT.CHANCE = 1.0

#Random "contrast"
cfg.INPUT.TRANSFORM.RAND_CONTRAST = CN()
cfg.INPUT.TRANSFORM.RAND_CONTRAST.ACTIVE = False
cfg.INPUT.TRANSFORM.RAND_CONTRAST.CHANCE = 1.0
cfg.INPUT.TRANSFORM.RAND_CONTRAST.ENHANCE = 25

#
↪  ---------------------------------------------------------------------------
# DataLoader
#
↪  ---------------------------------------------------------------------------
cfg.DATA_LOADER = CN()
cfg.DATA_LOADER.NUM_WORKERS = 8
cfg.DATA_LOADER.PIN_MEMORY = True
```

```
#
↪  ------------------------------------------------------------------------------
↪  #
# Solver - The same as optimizer
#
↪  ------------------------------------------------------------------------------
↪  #
cfg.TRAINER = CN()
cfg.TRAINER.EPOCHS = 25
cfg.TRAINER.SCHEDULER = "multistep"
cfg.TRAINER.LR_STEPS = [15, 20, 23]
cfg.TRAINER.GAMMA = 0.1
cfg.TRAINER.BATCH_SIZE = 32
cfg.TRAINER.EVAL_STEP = 1
cfg.TRAINER.OPTIMIZER = "sgd"
cfg.TRAINER.LR = 1e-3
cfg.TRAINER.MOMENTUM = 0.9
cfg.TRAINER.WEIGHT_DECAY = 5e-4
cfg.TRAINER.ACTIVATION = "sigmoid"

#
↪  ------------------------------------------------------------------------------
↪  #
# Specific test options
#
↪  ------------------------------------------------------------------------------
↪  #
cfg.TEST = CN()
cfg.TEST.BATCH_SIZE = 32


cfg.OUTPUT_DIR = "outputs"
cfg.DATASET_DIR = "datasets"

#
↪  ------------------------------------------------------------------------------
↪  #
# Inference options
#
↪  ------------------------------------------------------------------------------
↪  #
cfg.INFERENCE = CN()
#Hops per window should probably be around 1/MODEL.THRESHOLD
cfg.INFERENCE.HOPS_PER_WINDOW = 4
cfg.INFERENCE.OUTPUT_DIR = "predictions/"
cfg.INFERENCE.BATCH_SIZE = 128
cfg.INFERENCE.OUTPUT_FORMAT = "audacity"
cfg.INFERENCE.THRESHOLD = 0.76
cfg.INFERENCE.NUM_WORKERS = 8
```

C Configuration generation

```
import yaml
import copy
import numpy as np
```

```
def yaml_to_dict(yaml_path):
    yaml_file = open(yaml_path, 'r')
    text = yaml_file.read()
    yaml_file.close()
    yaml_as_dict = yaml.load(text)
    return yaml_as_dict



def write_value(my_dict, nesting_sequence, value):
    #Quick function to write recursive dict
    if len(nesting_sequence) == 1:
        my_dict[nesting_sequence[0]] = value
        return my_dict
    else:
        my_dict[nesting_sequence[0]] = write_value(
            my_dict[nesting_sequence[0]],
            nesting_sequence[1:],
            value
        )
        return my_dict


def multi_dict_creator(
    nesting_sequence : list,
    values : list,
    default_dict : dict
    ):
    """
    Function to create bunch of dictionaries with one variable altered
    Arguments :
    nesting sequence - sequence of key-dict nesting to get to value
    values - Values to write to nesting sequence value
    default_dict - default dictionary

    There is probably a better way to do this, but this was quick &
↪   easy
    """
    out_dicts = []
    for value in values:
        new_dict = default_dict
        new_dict = write_value(new_dict, nesting_sequence, value)
        out_dict = copy.deepcopy(new_dict)
        out_dicts.append(out_dict)
    return out_dicts


def dict_to_yaml(out_path, out_dict):
    with open(out_path, 'w') as outfile:
        yaml.dump(out_dict, outfile, default_flow_style=False)

def generate_configs(
    nesting_sequence:list,
    default_yaml_path:str,
    values:list,
    value_name : str,
```

```
    out_path : str
    ):
    """
    Function to generate multiple yaml files with minor alterations
    Arguments:
    nesting sequence - list of keys do get to altered value
    values - values to overwrite with
    value_name - name of altered value to be used for output filenames
    """
    default_dict = yaml_to_dict(yaml_path=default_yaml_path)
    new_dicts = multi_dict_creator(nesting_sequence, values,
    ↪  default_dict)
    for value, new_dict in zip(values, new_dicts):
        out_name = default_yaml_path.split(".")[0] + \
        "_" + value_name + str(value) + ".yaml"
        print(new_dict)
        dict_to_yaml(out_path + "/" + out_name, new_dict)



def main():
    #Example usage
    nesting_list = ["INPUT", "TRANSFORM", "RAND_GAUSS", "INTENSITY"]
    values = list(np.linspace(0.05, 2, 40))
    yaml_path = "configs/resnet50.yaml"
    out_path = "configs/rand_gauss"
    name = "rg"
    generate_configs(
        nesting_sequence=nesting_list,
        default_yaml_path=yaml_path,
        values=values,
        value_name=name,
        out_path=out_path
    )


if __name__ == "__main__":
    main()
```

D Dataset

D.1 Sliding-window training dataset

```
"""
Dataset class for self annotated Kautokeino bird vocalization dataset
Written (entirely) by:
Bendik Bogfjellmo (github.com/bendikbo) (bendik.bogfjellmo@gmail.com)
"""
import torch
import torchaudio
import numpy as np
import os
import pandas as pd
```

```
class Kauto5Cls(torch.utils.data.Dataset):

    def __init__(
                self,
                data_dir:str,
                transform=None,
                target_transform=None,
                is_train=True,
                inference_filepath="",
                audio_bit_length=0.0
                ):
        self.is_train = is_train
        if inference_filepath != "":
            self.inference = True
            self.entire_record, sample_rate =
            ↪  torchaudio.load(inference_filepath)
            self.length = self.entire_record.size()[1] / sample_rate
            #Can't have test and inference mode at the same time.
            self.is_train = False
        else:
            self.inference = False
        self.label_dict = {
                        "grønnstilk-sang"   :   0,
                        "bekkasinflukt"     :   1,
                        "småspove-sang"     :   2,
                        "heilo-pip"         :   3,
                        "heilo-sang"        :   4
                    }
        filenames = sorted(os.listdir(data_dir))
        self.data_dir = data_dir
        self.audio_filenames = []
        #audio_bits variable is used for validation to split up larger
        ↪  audio files
        self.audio_bits = []
        self.audio_bit_length = audio_bit_length
        for filename in filenames:
            if ".csv" not in (filename.lower()):
                self.audio_filenames.append(filename)
        self.transform = transform
        self.target_transform = target_transform
        self.annotation_dict = {}
        for audio_filename in self.audio_filenames:
            #csv should have same name as audio file except extension
            csv_filename = audio_filename.split(".")[0] + ".csv"
            #Using C engine because it's supposed to be faster,
            ↪  requires delimeter to be commas
            data_frame = pd.read_csv(data_dir + "/" + csv_filename,
            ↪  engine="c")
            #Storing all annotations in dictionary
            self.annotation_dict[audio_filename]=[]
            for _, row in data_frame.iterrows():
                self.annotation_dict[audio_filename].append(
                        (float(row["onset"]),
                        float(row["offset"]),
                        str(row["class"]))
                    )
        if not self.is_train:
```

```
            #Have to create a data loading method to create validation
            ↪  dataset
            self.convert_to_validation()
        self.validate_dataset()

    def convert_to_validation(self):
        """
        Function to create dataset for validation and/or test
        fills self.audio_bits with filenames and start time which
        is later used to crop the audio files through a transform.
        The cropping is done so that the audio bits always have 50%
        overlap with the next sequence, as this is how inference is
        intended to work on the dataset.
        """
        for audio_filename in self.audio_filenames:
            annotations = self.annotation_dict[audio_filename]
            earliest = min(annotations, key=lambda t:t[0])[0]
            latest = max(annotations, key=lambda t:t[1])[1]
            if earliest < 1:
                earliest = 0
            else:
                earliest = earliest - 1.0
            while (earliest + self.audio_bit_length) < latest:
                #Audio bits contain list of (filename, start(sec))
                self.audio_bits.append(
                    (audio_filename,
                    earliest)
                    )
                earliest += (self.audio_bit_length / 2)
            #No need to add latest annotation if it's over before 32
            ↪  secs
            if latest > self.audio_bit_length:
                self.audio_bits.append(
                    (audio_filename,
                    latest - self.audio_bit_length)
                )

    def validate_dataset(self):
        for audio_filename in self.audio_filenames:
            assert audio_filename in self.annotation_dict,\
                f"Did not find label for record {audio_filename} in
                ↪  labels"

    def __getitem__(self, idx):
        if self.is_train:
            lines, labels = self.get_annotation(idx)
            record = self._read_record(idx)
        #Don't have to care about lines or labels during inference
        elif self.inference:
            start_sec = idx * self.audio_bit_length / 2
            if (start_sec + self.audio_bit_length) > self.length:
                start_sec = self.length - self.audio_bit_length
            record, _, _ = self.transform.validation_crop(
                self.entire_record,
                start_sec,
                lines = None,
                labels = None
            )
```

```
            timespan = (start_sec, start_sec + self.audio_bit_length)
            return record, timespan, idx
        #Have to care about lines and labels during validation/testing
        else:
            audio_filename, start_sec = self.audio_bits[idx]
            lines, labels = self._get_annotation(audio_filename)
            record, _ = torchaudio.load(self.data_dir + "/" +
            ↪  audio_filename)
            #ValCrop object initalized in transform when is_train is
            ↪  set false
            record, lines, labels = self.transform.validation_crop(
                                                    record,
                                                    start_sec,
                                                    lines,
                                                    labels
                                                )
        if self.transform:
            record, lines, labels = self.transform(record, lines,
            ↪  labels)
        if self.target_transform is not None:
            targets = self.target_transform(lines, labels)
        return record, targets, idx

    def __len__(self):
        if self.is_train:
            return len(self.audio_filenames)
        else:
            return len(self.audio_bits)

    def _get_annotation(self, audio_filename):
        annotations = self.annotation_dict[audio_filename]
        lines = np.zeros((len(annotations), 2), dtype=np.float32)
        labels = np.zeros((len(annotations)), dtype=np.int64)
        for idx, annotation in enumerate(annotations):
            line = [
                annotation[0],
                annotation[1]
            ]
            lines[idx] = line
            labels[idx] = self.label_dict[annotation[2]]
        return lines, labels

    def get_annotation(self, index):
        audio_filename = self.audio_filenames[index]
        return self._get_annotation(audio_filename)

    def _read_record(self, idx):
        audio_filename = self.audio_filenames[idx]
        audio_filepath = self.data_dir + "/" + audio_filename
        record, _ = torchaudio.load(audio_filepath)
        return record
```

D.2 Sliding-window inference dataset

```
import torch
import torchaudio
```

```
import numpy as np
import os
from classifier.data.transform.transforms import AudioTransformer


class WindowSlide(torch.utils.data.Dataset):

    def __init__(
                self,
                cfg,
                audio_file:str
                ):
        assert os.path.exists(audio_file)
        self.input_length = cfg.RECORD_LENGTH
        hops_per_window = getattr(cfg.INFERENCE, "HOPS_PER_WINDOW", 4)
        self.hop_size = self.input_length // hops_per_window
        self.transform = AudioTransformer(cfg.TRANSFORM,
        ↪  is_train=False)
        self.sample_rate = cfg.INPUT.SAMPLE_FREQ
        self.record, fs = torchaudio.load(audio_file)
        if fs != self.sample_rate:
            self.resampler = torchaudio.transforms.Resample(
                orig_freq=fs,
                new_freq=self.sample_rate
            )

    def __getitem__(self, idx):
        audio_bit_start = min(
            self.record.size()[0] - self.input_length,
            idx * self.hop_size
        )
        audio_bit = torch.narrow(
            self.record,
            0,
            audio_bit_start,
            self.input_length
        )
        return audio_bit, idx

    def __len__(self):
        return int(np.ceil(self.record.size()[0] / self.hop_size))
```

D.3 SSD-based architecture dataset

```
class Kauto5Cls(torch.utils.data.Dataset):

    def __init__(
                self,
                data_dir:str,
                transform=None,
                target_transform=None,
                is_train=True,
                inference_filepath="",
                audio_bit_length=0.0
                ):
        self.is_train = is_train
```

```
        if inference_filepath != "":
            self.inference = True
            self.entire_record, sample_rate =
            ↪  torchaudio.load(inference_filepath)
            self.length = self.entire_record.size()[1] / sample_rate
            #Can't have test and inference mode at the same time.
            self.is_train = False
        else:
            self.inference = False
        self.label_dict = {
                        "grønnstilk-sang"   :   0,
                        "bekkasinflukt"     :   1,
                        "småspove-sang"     :   2,
                        "heilo-pip"         :   3,
                        "heilo-sang"        :   4
                    }
        filenames = sorted(os.listdir(data_dir))
        self.data_dir = data_dir
        self.audio_filenames = []
        #audio_bits variable is used for validation to split up larger
        ↪  audio files
        self.audio_bits = []
        self.audio_bit_length = audio_bit_length
        for filename in filenames:
            if ".csv" not in (filename.lower()):
                self.audio_filenames.append(filename)
        self.transform = transform
        self.target_transform = target_transform
        self.annotation_dict = {}
        for audio_filename in self.audio_filenames:
            #csv should have same name as audio file except extension
            csv_filename = audio_filename.split(".")[0] + ".csv"
            #Using C engine because it's supposed to be faster,
            ↪  requires delimeter to be commas
            data_frame = pd.read_csv(data_dir + "/" + csv_filename,
            ↪  engine="c")
            #Storing all annotations in dictionary
            self.annotation_dict[audio_filename]=[]
            for _, row in data_frame.iterrows():
                self.annotation_dict[audio_filename].append(
                        (float(row["onset"]),
                        float(row["offset"]),
                        str(row["class"]))
                    )
        if not self.is_train:
            #Have to create a data loading method to create validation
            ↪  dataset
            self.convert_to_validation()
        self.validate_dataset()

    def convert_to_validation(self):
        """
        Function to create dataset for validation and/or test
        fills self.audio_bits with filenames and start time which
        is later used to crop the audio files through a transform.
        The cropping is done so that the audio bits always have 50%
        overlap with the next sequence, as this is how inference is
        intended to work on the dataset.
```

```
        """
        for audio_filename in self.audio_filenames:
            annotations = self.annotation_dict[audio_filename]
            earliest = min(annotations, key=lambda t:t[0])[0]
            latest = max(annotations, key=lambda t:t[1])[1]
            if earliest < 1:
                earliest = 0
            else:
                earliest = earliest - 1.0
            while (earliest + self.audio_bit_length) < latest:
                #Audio bits contain list of (filename, start(sec))
                self.audio_bits.append(
                    (audio_filename,
                    earliest)
                    )
                earliest += (self.audio_bit_length / 2)
            #No need to add latest annotation if it's over before 32
            ↪ secs
            if latest > self.audio_bit_length:
                self.audio_bits.append(
                    (audio_filename,
                    latest - self.audio_bit_length)
                )

    def validate_dataset(self):
        for audio_filename in self.audio_filenames:
            assert audio_filename in self.annotation_dict,\
                f"Did not find label for record {audio_filename} in
                ↪ labels"

    def __getitem__(self, idx):
        if self.is_train:
            lines, labels = self.get_annotation(idx)
            record = self._read_record(idx)
        #Don't have to care about lines or labels during inference
        elif self.inference:
            start_sec = idx * self.audio_bit_length / 2
            if (start_sec + self.audio_bit_length) > self.length:
                start_sec = self.length - self.audio_bit_length
            record, _, _ = self.transform.validation_crop(
                self.entire_record,
                start_sec,
                lines = None,
                labels = None
            )
            timespan = (start_sec, start_sec + self.audio_bit_length)
            return record, timespan, idx
        #Have to care about lines and labels during validation/testing
        else:
            audio_filename, start_sec = self.audio_bits[idx]
            lines, labels = self._get_annotation(audio_filename)
            record, _ = torchaudio.load(self.data_dir + "/" +
            ↪ audio_filename)
            #ValCrop object initalized in transform when is_train is
            ↪ set false
            record, lines, labels = self.transform.validation_crop(
                                                    record,
                                                    start_sec,
```

```
                                                    lines,
                                                    labels
                                                )
        if self.transform:
            record, lines, labels = self.transform(record, lines,
            ↪  labels)
        if self.target_transform is not None:
            targets = self.target_transform(lines, labels)
        return record, targets, idx

    def __len__(self):
        if self.is_train:
            return len(self.audio_filenames)
        else:
            return len(self.audio_bits)

    def _get_annotation(self, audio_filename):
        annotations = self.annotation_dict[audio_filename]
        lines = np.zeros((len(annotations), 2), dtype=np.float32)
        labels = np.zeros((len(annotations)), dtype=np.int64)
        for idx, annotation in enumerate(annotations):
            line = [
                annotation[0],
                annotation[1]
            ]
            lines[idx] = line
            labels[idx] = self.label_dict[annotation[2]]
        return lines, labels

    def get_annotation(self, index):
        audio_filename = self.audio_filenames[index]
        return self._get_annotation(audio_filename)

    def _read_record(self, idx):
        audio_filename = self.audio_filenames[idx]
        audio_filepath = self.data_dir + "/" + audio_filename
        record, _ = torchaudio.load(audio_filepath)
        return record
```

E Transform code

```
import torch
from torch import Tensor
import torch.nn as nn
import numpy as np
import torchvision
import torchaudio


class ToSampleCoords(nn.Module):
    """
    Pytorch module to convert coordinates measured in seconds
    into coordinates measured in sample Nos,
    Default sample rate is 16kHz unless this is set through
↪   cfg.SAMPLE_RATE
```

```
    """
    def __init__(self, cfg):
        super(ToSampleCoords, self).__init__()
        self.sample_rate = 16000
        if hasattr(cfg, "SAMPLE_RATE"):
            self.sample_rate = cfg.SAMPLE_RATE
    def forward(self, x, lines=None, labels=None):
        for idx, annotation in enumerate(lines):
            onset = annotation[0]
            offset = annotation[1]
            new_onset = np.ceil(onset*self.sample_rate)
            new_offset = np.floor(offset*self.sample_rate)
            lines[idx] = [new_onset, new_offset]
        return x, lines, labels


class Differentiate1D(nn.Module):
    """
    Pytorch module to discretely differentiate a 1D input tensor
    Differentiates by taking tensor[1:end] - tensor[0:end-1]
    """
    def __init__(self, cfg):
        super(Differentiate1D, self).__init__()
        if hasattr(cfg.DIFFERENTIATE, "STEP"):
            self.step = cfg.DIFFERENTIATE.STEP
        else:
            self.step = 1

    def forward(self, x, lines = None, labels = None):
        minuend = torch.narrow(x, 0, self.step, x.size()[0] -
        ↪  self.step)
        subtrahend = torch.narrow(x, 0, 0, x.size()[0] - self.step)
        x = minuend - subtrahend
        return x, lines, labels


class RandFlip1D(nn.Module):
    """
    Pytorch module to "flip" signal along the x-axis
    Supports random application through cfg.RAND_FLIP.CHANCE or
↪   cfg.CHANCE
    defaults to random application with p = 0.5
    """
    def __init__(self, cfg):
        super(RandFlip1D, self).__init__()
        #Code block for setting up random application
        if hasattr(cfg.RAND_FLIP, "CHANCE"):
            self.chance = cfg.RAND_FLIP.CHANCE
        elif hasattr(cfg, "CHANCE"):
            self.chance = cfg.CHANCE
        else:
            self.chance = 0.5
    def forward(self, x : Tensor, lines = None, labels = None):
        if np.random.uniform() < self.chance:
            x = x.mul(-1)
        return x, lines, labels
```

```
class RandGauss1D(nn.Module):
    """
    Pytorch module to add gaussian noise to 1D tensor
    Supports random application through cfg.RAND_FLIP.CHANCE or
↪   cfg.CHANCE
    Also supports noise intensity through INTENSITY
    For uniformly distributed intensity, include a RAND to cfg.GAUSS
    """
    def __init__(self, cfg):
        super(RandGauss1D, self).__init__()
        #Code block for random application
        if hasattr(cfg.RAND_GAUSS, "CHANCE"):
            self.chance = cfg.RAND_GAUSS.CHANCE
        elif hasattr(cfg, "CHANCE"):
            self.chance = cfg.CHANCE
        else:
            self.chance = 0.5
        self.intensity = 1.0
        if hasattr(cfg.RAND_GAUSS, "INTENSITY"):
            self.intensity = cfg.RAND_GAUSS.INTENSITY
        self.random_intensity = False
        if hasattr(cfg.RAND_GAUSS, "RAND"):
            self.random_intensity = True
    def forward(self, x : Tensor, lines = None, labels = None):
        if np.random.uniform() < self.chance:
            #noise_factor = std(x) * intensity
            noise_factor = x.std() * self.intensity
            if self.random_intensity:
                noise_factor *= float(np.random.uniform())
            #x_i + N(0,1) * noise_factor
            x += torch.randn(x.size()) * noise_factor
        return x, lines, labels


class RandAmpAtt1D(nn.Module):
    """
    Pytorch module to randomly amplify or attenuate signal
    Supports random application through cfg.RAND_AMP_ATT.CHANCE or
↪   cfg.CHANCE
    defaults to random application with p = 0.5
    cfg.AMP_ATTEN is required to have parameter "FACTOR"
    """
    def __init__(self, cfg):
        super(RandAmpAtt1D, self).__init__()
        #Code block for setting up random application
        if hasattr(cfg.RAND_AMP_ATT, "CHANCE"):
            self.chance = cfg.RAND_AMP_ATT.CHANCE
        elif hasattr(cfg, "CHANCE"):
            self.chance = cfg.CHANCE
        else:
            self.chance = 0.5
        assert hasattr(cfg.RAND_AMP_ATT, "FACTOR"),\
            "Transform AmpAtt1D requires parameter cfg.FACTOR"
        self.factor = max(1/cfg.RAND_AMP_ATT.FACTOR,
        ↪   cfg.RAND_AMP_ATT.FACTOR)
    def forward(self, x : Tensor, lines = None, labels = None):
        if np.random.uniform() < self.chance:
```

```
            factor = np.random.uniform(low = 1/self.factor, high =
            ↪  self.factor)
            x.mul(factor)
        return x, lines, labels


class RandContrast1D(nn.Module):
    """
    Pytorch module to add random contrast to the data
    Supports random application through cfg.RAND_CONTRAST.CHANCE or
↪   cfg.CHANCE
    defaults to random application with p = 0.5
    Contrast enhancement amount may range between 0-100
    enhancement of 0 still yields a significant contrast enhancement
    """
    def __init__(self, cfg):
        super(RandContrast1D, self).__init__()
        #Code block for setting up random application
        if hasattr(cfg.RAND_CONTRAST, "CHANCE"):
            self.chance = cfg.RAND_CONTRAST.CHANCE
        elif hasattr(cfg, "CHANCE"):
            self.chance = cfg.CHANCE
        else:
            self.chance = 0.5
        assert hasattr(cfg.RAND_CONTRAST, "ENHANCE"), "RandContrast1D
        ↪  needs attribute cfg.RAND_CONTRAST.ENHANCE)"
        self.enhancement = cfg.RAND_CONTRAST.ENHANCE
        if self.enhancement < 0 or self.enhancement > 100:
            print(f"enhancement not in 0-100 range, setting to
            ↪  {np.abs(self.enhancemnet % 100)}")
            self.enhancement = np.abs(self.enhancement % 100)
    def forward(self, x : Tensor, lines = None, labels = None):
        if np.random.uniform() < self.chance:
            amount = np.random.uniform()*self.enhancement
            torchaudio.functional.contrast(waveform=x,
            ↪  enhancement_amount=amount)
        return x, lines, labels


class Crop1D(nn.Module):
    """
    Pytorch module to crop one dimensional signal
    Supports "random" mode and "center" mode  through cfg.CROP.TYPE
    defaults to random application with p = 1.0 unless cfg.CROP.CHANCE
↪   is set
    If the chance doesn't activate, it defaults to crop
    to the middle of the input time series.
    Onset/offset annotations can optionally be added through the lines
↪   variable
    """
    def __init__(self, cfg):
        super(Crop1D, self).__init__()
        #Code block for setting random application
        self.type = "random"
        if hasattr(cfg.CROP, "TYPE"):
            self.type = cfg.CROP.TYPE
        if hasattr(cfg.CROP, "CHANCE"):
            self.chance = cfg.CROP.CHANCE
```

```
        else:
            self.chance = 1
        #Chance set to 0 => centercrop
        if self.type == "center":
            self.chance = 0
        assert hasattr(cfg, "LENGTH"), "Crop1D needs output tensor
        ↪  length to function"
        self.length = cfg.LENGTH
    def forward(self, x : Tensor, lines=None, labels=None):
        start = int(np.floor((x.size()[1]-self.length)/2))
        if np.random.uniform() < self.chance:
            min_start, max_start = (0, x.size()[1] - self.length - 1)
            start = np.random.randint(low=min_start, high = max_start)
        x = x.narrow(1, start, self.length)
        #Onset/onset annotations need to be fixed if they're added
        if lines is not None:
            new_lines = []
            new_labels = []
            for idx, annotation in enumerate(lines):
                #Have to deduct starting point from the onset
                annotation_onset = annotation[0] - start
                #End = starting point + annotation length
                annotation_offset = annotation_onset + (annotation[1] -
                ↪  annotation[0])
                #Fix out of bounds issues
                if annotation_offset > self.length:
                    annotation_offset = self.length
                if annotation_onset < 0:
                    annotation_onset = 0
                if annotation_onset > self.length or annotation_offset
                ↪  < 0:
                    continue
                else:
                    new_lines.append([annotation_onset,
                    ↪  annotation_offset])
                    new_labels.append(labels[idx])
            lines = np.zeros((len(new_lines), 2), dtype=np.float32)
            labels = np.zeros((len(new_labels)), dtype=np.int64)
            for idx, new_line in enumerate(new_lines):
                lines[idx] = new_line
                labels[idx] = new_labels[idx]
        return x, lines, labels


class Spectrify(nn.Module):
    """
    Pytorch module to convert 1D tensor to spectrogram(s)
    cfg.RESOLUTION to specify [WIDTH, HEIGHT]
    cfg.CHANNELS to specify channel order between ["mel", "log",
↪   "normal"]
    cfg.FREQ_CROP, if only part of the spectrogram frequency dimension
↪   is needed
    All outputs are normalized
    """
    def __init__(self, cfg):
        super(Spectrify, self).__init__()
        self.length = cfg.LENGTH
        #Setting imagenet resolution as defaults
```

```
        self.width = 224
        self.height = 224
        sample_freq = 16000
        if hasattr(cfg.SPECTROGRAM, "RESOLUTION"):
            self.width = cfg.SPECTROGRAM.RESOLUTION[0]
            self.height = cfg.SPECTROGRAM.RESOLUTION[1]
        self.out_width = self.width
        self.out_height = self.height
        if hasattr(cfg.SPECTROGRAM, "FREQ_CROP"):
            self.crop = cfg.SPECTROGRAM.FREQ_CROP
            self.out_height = self.crop[1]
        else:
            self.crop = None
        if hasattr(cfg, "SAMPLE_RATE"):
            sample_freq = cfg.SAMPLE_RATE
        self.transformations = nn.ModuleList()
        self.channels = ["normal","log","mel"]
        self.resize = torchvision.transforms.Resize((self.out_height,
        ↪  self.out_width))
        if hasattr(cfg.SPECTROGRAM, "CHANNELS"):
            self.channels = cfg.SPECTROGRAM.CHANNELS
        for channel in self.channels:
            if channel == "mel":
                #Not sure whether this is right or not
                out_height = self.height
                if self.crop is not None:
                    out_height = self.crop[1]
                hop_size = cfg.LENGTH // self.width
                melify = nn.Sequential(
                    torchaudio.transforms.MelSpectrogram(
                                                n_mels=self.height,
                                                hop_length=hop_size
                        )
                )
                self.transformations.append(melify)
            if channel == "log" or channel == "normal":
                    num_ffts = (self.height - 1)*2 + 1
                    #A bit unsure of line below, but think it should
                    ↪  give right width
                    hop_size = cfg.LENGTH // self.width
                    self.transformations.append(
                        torchaudio.transforms.Spectrogram(
                            n_fft = num_ffts,
                            hop_length = hop_size
                        )
                    )
    def forward(self, x : Tensor, lines = None, labels = None):
        if lines is not None:
            new_lines = np.zeros((len(lines), 2), dtype=np.float32)
            for idx, annotation in enumerate(lines):
                #Onset_pixel = onset_sample * spectrogram_width /
                ↪  waveform_sample_length
                annotation_onset =
                ↪  int(np.round((annotation[0]*self.width)/self.length))
                annotation_offset =
                ↪  int(np.round((annotation[1]*self.width)/self.length))
                new_lines[idx] = [annotation_onset, annotation_offset]
            lines = new_lines
```

```
        #Convert first element of output to be able to just torch.cat
        ↪  it later
        y = self.transformations[0](x)
        if not (self.crop is None) and self.channels[0] != "mel":
                    y = torch.narrow(
                        input = y,
                        dim = y.dim() - 2,
                        start = self.crop[0],
                        length = self.crop[1]
                    )
        #Normalization after crop
        y = (y - torch.mean(y)) / torch.std(y)
        width = y.size()[-1]
        height = y.size()[-1]
        if width != self.out_width or height != self.out_height:
            y = self.resize(y)
        #If more spectrograms are specified, use them
        if len(self.transformations) > 1:
            for i, transformation in
            ↪  enumerate(self.transformations[1:]):
                channel = transformation(x)
                #No point in cropping mel spectrogram as its done
                ↪  through resize
                #This is because of how the melscale works.
                if not (self.crop is None) and self.channels[i + 1] !=
                ↪  "mel":
                    channel = torch.narrow(
                        input = channel,
                        dim = channel.dim() - 2,
                        start = self.crop[0],
                        length = self.crop[1]
                    )
                width = channel.size()[-1]
                height = channel.size()[-2]
                if width != self.out_width or height !=
                ↪  self.out_height:
                    channel = self.resize(channel)
                #Statement for logarithmic output.
                if self.channels[i+1] == "log":
                    channel = torch.log(channel)
                #Normalization after crop
                channel = (channel - torch.mean(channel)) /
                ↪  torch.std(channel)
                y = torch.cat((y, channel), 0)
        return y, lines, labels


class ValCrop(nn.Module):
    """
    Pytorch module to crop one dimensional signal specified by
    Supports "random" mode and "center" mode  through cfg.CROP.TYPE
    defaults to random application with p = 1.0 unless cfg.CROP.CHANCE
↪   is set
    If the chance doesn't activate, it defaults to crop
    to the middle of the input time series.
    Onset/offset annotations can optionally be added through the lines
↪   variable
```

```
    """
    def __init__(self, cfg):
        super(ValCrop, self).__init__()
        assert hasattr(cfg, "LENGTH"), "ValCrop needs output tensor
        ↪  length to function"
        self.length = cfg.LENGTH
        self.sample_freq = 16000
        if hasattr(cfg, "SAMPLE_RATE"):
            self.sample_freq = cfg.SAMPLE_RATE
        #If data should be differentiated, a sample has to be added
        if hasattr(cfg, "DIFFERENTATE"):
            self.length += getattr(cfg.DIFFERENTATE, "STEP", default=1)
    def forward(self, x : Tensor, start_second=0.0, lines=None,
    ↪  labels=None):
        signal_length = x.size()[1]
        leftover_samples = signal_length -
        ↪  start_second*self.sample_freq + self.length
        #Implicates that this is the last audio bit in the file
        #In case there is more samples, will then add them to the audio
        ↪  bit
        if leftover_samples < (self.length / 2):
            start_second += min(1.0, leftover_samples/self.sample_freq)
        start = int(np.floor(self.sample_freq * start_second))
        x = x.narrow(1, start, self.length)
        #Onset/onset annotations need to be fixed if they're added
        if lines is not None:
            new_lines = []
            new_labels = []
            for idx, annotation in enumerate(lines):
                #Have to deduct starting point from the onset
                annotation_onset = annotation[0] - start_second
                #End = starting point + annotation length
                annotation_offset = annotation_onset + (annotation[1] -
                ↪  annotation[0])
                #Fix out of bounds issues
                if annotation_offset > self.length/self.sample_freq:
                    annotation_offset = self.length/self.sample_freq
                if annotation_onset < 0:
                    annotation_onset = 0
                if annotation_onset > self.length/self.sample_freq or
                ↪  annotation_offset < 0:
                    continue
                else:
                    new_lines.append([annotation_onset,
                    ↪  annotation_offset])
                    new_labels.append(labels[idx])
            lines = np.zeros((len(new_lines), 2), dtype=np.float32)
            labels = np.zeros((len(new_labels)), dtype=np.int64)
            for idx, new_line in enumerate(new_lines):
                lines[idx] = new_line
                labels[idx] = new_labels[idx]
        return x, lines, labels


class AudioTransformer(nn.Module):
    def __init__(self, cfg, is_train=True):
        super(AudioTransformer, self).__init__()
        self.transforms = nn.ModuleList()
```

```
        #Implementations of new transforms will have to be added in
        ↪  these
        #dictionaries to be supported by YAML-specification
        if is_train:
            transform_dict = {
                                    "SAMPLE_COORDS" :
                                    ↪  ToSampleCoords,
                                    "DIFFERENTIATE" :
                                    ↪  Differentiate1D,
                                    "CROP"          :       Crop1D,
                                    "RAND_FLIP"     :       RandFlip1D,
                                    "RAND_GAUSS"    :
                                    ↪  RandGauss1D,
                                    "RAND_AMP_ATT"  :
                                    ↪  RandAmpAtt1D,
                                    "RAND_CONTRAST" :
                                    ↪  RandContrast1D,
                                    "SPECTROGRAM"   :       Spectrify
                                }
        else:
            #Data augmentation and crop is unnecessary on
            ↪  validation/test data
            #As both test and validation data has their own cropping
            ↪  function
            transform_dict = {
                                    "SAMPLE_COORDS" :
                                    ↪  ToSampleCoords,
                                    "DIFFERENTIATE" :
                                    ↪  Differentiate1D,
                                    "RAND_CONTRAST" :
                                    ↪  RandContrast1D,
                                    "SPECTROGRAM"   :       Spectrify
                                }
            self.validation_crop = ValCrop(cfg)
        for transform_name in transform_dict:
            if hasattr(cfg, transform_name):
                if getattr(cfg, transform_name).ACTIVE == True:
                    #This is not allowed with torchscript
                    transform_module =
                    ↪  transform_dict[transform_name](cfg)
                    self.transforms.append(transform_module)
    def forward(self, x, lines = None, labels = None):
        for transform in self.transforms:
            x, lines, labels = transform(
                x,
                lines = lines,
                labels = labels
                )
        return x, lines, labels
```

F Sliding-window target transform

```
import torch
import torch.nn as nn
```

```
class TargetTransform(nn.Module):
    """
    Target transformation for a "scanline" classifier,
    strategy is that any "scanline" containing more than a threshold
    part of it as a value for a class should be regarded as positive.
    """
    def __init__(self, cfg):
        super(TargetTransform, self).__init__()
        if hasattr(cfg.INPUT.TRANSFORM, "SPECTROGRAM"):
            self.length = cfg.INPUT.TRANSFORM.SPECTROGRAM.RESOLUTION[0]
        else:
            self.length = cfg.INPUT.RECORD_LENGTH
        self.threshold = cfg.MODEL.THRESHOLD
        self.num_classes = cfg.MODEL.NUM_CLASSES
    def forward(self, lines = None, labels = None):
        """
        Forward function assumes lines is relative to model input
↪   dimensions.
        """
        out_labels = torch.zeros(self.num_classes)
        line_contents = {}
        if lines is not None:
            for idx, line in enumerate(lines):
                if labels[idx] not in line_contents:
                    line_contents[labels[idx]] = 0
                line_contents[labels[idx]] += (line[1] - line[0]) /
                ↪  self.length
                #Mark data as positive if more than threshold of it is
                ↪  positive.
                if line_contents[labels[idx]] > self.threshold:
                    out_labels[labels[idx]] = 1
        return out_labels
```

G Training automation

G.1 Training script

```
import argparse
import logging
import torch
import pathlib
import numpy as np
from classifier.config.defaults import cfg
from classifier.data.build import make_data_loaders
from classifier.logger import setup_logger
from classifier.trainer import Trainer
from classifier.models import build_model
from classifier.utils import to_cuda

np.random.seed(0)
torch.manual_seed(0)
```

```
def start_train(cfg):
    logger = logging.getLogger('classification.trainer')
    model = build_model(cfg)
    model = to_cuda(model)
    dataloaders = make_data_loaders(cfg)
    trainer = Trainer(
        cfg,
        model=model,
        dataloaders=dataloaders
        )
    trainer.train()
    return trainer.model


def get_parser():
    parser = argparse.ArgumentParser(description='Single Record MultiLine Detector Tra
    parser.add_argument(
        "config_file",
        default="",
        metavar="FILE",
        help="path to config file",
        type=str,
    )
    parser.add_argument(
        "opts",
        help="Modify config options using the command-line",
        default=None,
        nargs=argparse.REMAINDER,
    )
    return parser


def main():
    args = get_parser().parse_args()
    cfg.merge_from_file(args.config_file)
    cfg.merge_from_list(args.opts)
    cfg.freeze()
    output_dir = pathlib.Path(cfg.OUTPUT_DIR)
    output_dir.mkdir(exist_ok=True, parents=True)
    logger = setup_logger("Classifier", output_dir)
    logger.info(args)
    logger.info("Loaded configuration file {}".format(args.config_file))
    with open(args.config_file, "r") as cf:
        config_str = "\n" + cf.read()
        logger.info(config_str)
    logger.info("Running with config:\n{}".format(cfg))
    model = start_train(cfg)

if __name__ == "__main__":
    main()
```

G.2 Run experiments script

```
#!/usr/bin/bash

#Use this script to run through all configuration files
```

```
#in a chosen directory. Script will run all training sessions
#and dump the seission outputs as a log in outputs/

cd "$(dirname "$0")"
proj_dir=$(dirname "$0")
experiment_dir="$1"
#Cange experiment dir to run with config files in your chosen
 ↪  directory
for config_file in "${proj_dir}/${experiment_dir}"/*
do
  #Getting conf_name for output logfile
  conf_name="$(basename $config_file)"
  conf_name=$(echo "$conf_name" | cut -f 1 -d '.')
  env/bin/python train.py $config_file >
   ↪  $proj_dir/outputs/$conf_name.log
done
```

H Evalutaion code

H.1 SSED evaluation code

```
"""
Scanline classifier mean Average precision evaluator
Written as part of master thesis by Bendik Bogfjellmo
(github.com/bendikbo) (bendik.bogfjellmo@gmail.com)
"""
import torch
from statistics import mean


def _calculate_AP(
    class_predictions: torch.Tensor,
    class_targets: torch.Tensor,
    recall_vals = 1000,
    conf_vals = 1000
    ):
    """
    calculates average precision for a single class
    Arguments:
    - class_predictions : torch.Tensor in shape of [num_preds]
    - class_targets     : torch.Tensor in shape of [num_targets]
    where num_preds == num_targets
    """
    #linear approximation of continuous confidence threshold
    confidence_thresholds = torch.linspace(0, 1, conf_vals)
    #Initalize array of predictions considered positive at each
     ↪  distinct confidence threshold
    pos_preds = torch.zeros(conf_vals, class_predictions.size()[0])
    for i in range(conf_vals):
        #confidence >= threshold => positive prediction
        pos_preds[i, class_predictions>=confidence_thresholds[i]] = 1
    #tensor of size [conf_vals] containing true positives for threshold
    num_true_positives = torch.sum((pos_preds*class_targets), dim=1)
    #tensor of size [conf_vals] containing false positives for each
     ↪  threshold
```

```
    num_false_positives = torch.sum(pos_preds, dim=1) -
    ↪  num_true_positives
    #The same for false negatives
    num_false_negatives = torch.sum(class_targets) - num_true_positives
    #initialize tensors for precision and recalls
    precisions = torch.zeros(conf_vals)
    recalls = torch.zeros_like(precisions)
    for i in range(conf_vals):
        num_tp = num_true_positives[i]
        num_fp = num_false_positives[i]
        num_fn = num_false_negatives[i]
        if (num_tp + num_fp) == 0:
            precisions[i] = 1
        else:
            precisions[i] = num_tp/(num_tp + num_fp)
        if (num_tp + num_fn) == 0:
            recalls[i] = 0
        else:
            recalls[i] = num_tp / (num_tp + num_fn)
    recall_levels = torch.linspace(0, 1, recall_vals)
    final_precisions = torch.zeros_like(recall_levels)
    for i in range(recall_vals):
        recall_level = recall_levels[i]
        recall_level_precisions = precisions[recalls >= recall_level]
        if not precisions.numel():
            final_precisions[i] = 0
        else:
            final_precisions[i] = torch.max(recall_level_precisions)
    return torch.mean(final_precisions)



def calculate_mAP(predictions: torch.Tensor, targets : torch.Tensor):
    """
    calculates mean average precision based on predictions and targets.
    Arguments:
    - Predictions   : torch.Tensor in shape of [num_preds, num_classes]
    - Targets       : torch.Tensor in shape of [num_targets,
↪   num_classes]
    where num_targets == num_preds
    """
    ap_vals = {}
    for i in range(targets.size()[-1]):
        #print(f"class: {i}")
        class_predictions   = predictions[:, i]
        class_targets       = targets[:,i]
        class_AP = _calculate_AP(class_predictions, class_targets)
        #Tensors are a bit annoying to work without significant payoff.
        ap_vals[i] = float(class_AP)
    ap_vals["mAP"] = mean(ap_vals.values())
    return ap_vals
```

H.2 SSD-based architecture evaluation

```
import numpy as np
import matplotlib.pyplot as plt
```

```
def calculate_iou(prediction_line, gt_line):
    """Calculate intersection over union of single predicted and ground
    ↪  truth line.
    Args:
        prediction_line (np.array of floats): location of predicted
↪   sound as
            [tmin, tmax]
        gt_line (np.array of floats): location of ground truth sound as
            [tmin, tmax]
        returns:
            float: value of the intersection of union for the two
↪   lines.
    """
    t1_t, t2_t = gt_line
    t1_p, t2_p = prediction_line
    if t2_t < t1_p or t2_p < t1_t:
        return 0.0

    # Compute intersection
    t1i = max(t1_t, t1_p)
    t2i = min(t2_t, t2_p)
    intersection = (t2i - t1i)

    # Compute union
    pred_t = (t2_p - t1_p)
    gt_t = (t2_t - t1_t) * (t2_t - t1_t)
    union = pred_t + gt_t - intersection
    iou = 0
    iou = intersection / union
    assert iou >= 0 and iou <= 1
    return iou


def calculate_precision(num_tp, num_fp, num_fn):
    """ Calculates the precision for the given parameters.
        Returns 1 if num_tp + num_fp = 0
    Args:
        num_tp (float): number of true positives
        num_fp (float): number of false positives
        num_fn (float): number of false negatives
    Returns:
        float: value of precision
    """
    if (num_tp + num_fp) == 0:
        return 1
    return num_tp / (num_fp + num_tp)


def calculate_recall(num_tp, num_fp, num_fn):
    """ Calculates the recall for the given parameters.
        Returns 0 if num_tp + num_fn = 0
    Args:
        num_tp (float): number of true positives
        num_fp (float): number of false positives
        num_fn (float): number of false negatives
    Returns:
        float: value of recall
```

```
    """
    denominator = num_tp + num_fn
    if denominator == 0:
        return 0
    return num_tp / denominator


def get_all_line_matches(prediction_lines, gt_lines, iou_threshold):
    """Finds all possible matches for the predicted lines to the ground
    ↪  truth lines.
        No bounding line can have more than one match.
        #TODO: allow for possibility of multiple pred line having one
↪   gt line as match
        and one pred line having multiple gt line as match.
        Remember: Matching of bounding lines should be done with
↪   decreasing IoU order!
    Args:
        prediction_lines: (np.array of floats): list of predicted
↪   bounding lines
            shape: [number of predicted lines, 2].
            Each row includes [tmin, tmax]
        gt_lines: (np.array of floats): list of bounding lines ground
↪   truth
            objects with shape: [number of ground truth lines, 2].
            Each row includes [tmin, tmax]
    Returns the matched lines (in corresponding order):
        prediction_lines: (np.array of floats): list of predicted
↪   bounding lines
            shape: [number of line matches, 2].
        gt_lines: (np.array of floats): list of bounding lines ground
↪   truth
            objects with shape: [number of line matches, 2].
            Each row includes [tmin, tmax]
    """
    ious = []
    indices = []
    # Find all possible matches with a IoU >= iou threshold
    for pred_idx, pred_line in enumerate(prediction_lines):
        for gt_idx, gt_line in enumerate(gt_lines):
            iou = calculate_iou(pred_line, gt_line)
            if iou >= 0:
                ious.append(iou_threshold)
                indices.append((pred_idx, gt_idx))


    ious = np.array(ious)
    indices = np.array(indices)
    if indices.size == 0:
        return np.array([]), np.array([])
    assert indices.shape[1] == 2

    # Sort all matches on IoU in descending order
    sorted_idx = np.argsort(ious)[::-1]
    ious = ious[sorted_idx]
    indices = indices[sorted_idx]

    # Find all matches with the highest IoU threshold
    seen_prediction_lines = np.zeros(len(prediction_lines))
```

```
    seen_gt_lines = np.zeros(len(gt_lines))
    final_prediction_lines = []
    final_gt_lines = []

    for (pred_idx, gt_idx) in indices:
        if seen_prediction_lines[pred_idx] == 0 and
        ↪  seen_gt_lines[gt_idx] == 0:
            final_prediction_lines.append(prediction_lines[pred_idx])
            final_gt_lines.append(gt_lines[gt_idx])

            seen_prediction_lines[pred_idx] = 1
            seen_gt_lines[gt_idx] = 1

    return np.array(final_prediction_lines), np.array(final_gt_lines)


def calculate_individual_record_result(prediction_lines, gt_lines,
↪  iou_threshold):
    """Given a set of prediction lines and ground truth lines,
       calculates true positives, false positives and false negatives
       for a single record.
       NB: prediction_lines and gt_lines are not matched!
    Args:
        prediction_lines: (np.array of floats): list of predicted
↪  bounding lines
            shape: [number of predicted lines, 2].
            Each row includes [tmin, tmax]
        gt_lines: (np.array of floats): list of bounding lines ground
↪  truth
            objects with shape: [number of ground truth lines, 2].
            Each row includes [tmin, tmax]
    Returns:
        dict: containing true positives, false positives, true
↪  negatives, false negatives
            {"true_pos": int, "false_pos": int, false_neg": int}
    """
    matched_pred_lines, matched_gt_lines = get_all_line_matches(
        prediction_lines, gt_lines, iou_threshold)

    num_tp = len(matched_gt_lines)
    num_fp = len(prediction_lines) - num_tp
    num_fn = len(gt_lines) - num_tp
    return {"true_pos": num_tp, "false_pos": num_fp, "false_neg":
    ↪  num_fn}


def calculate_precision_recall_all_records(
    all_prediction_lines, all_gt_lines, iou_threshold):
    """Given a set of prediction lines and ground truth lines for all
    ↪  records,
       calculates recall and precision for all records.
       NB: all_prediction_lines and all_gt_lines are not matched!
    Args:
        all_prediction_lines: (list of np.array of floats): each
↪  element in the list
            is a np.array containing all predicted bounding lines for
↪  the given record
            with shape: [number of predicted lines, 2].
```

```
            Each row includes [tmin, tmax]
        all_gt_lines: (list of np.array of floats): each element in the
↪   list
            is a np.array containing all ground truth bounding lines
↪   for the given record
            objects with shape: [number of ground truth lines, 2].
            Each row includes [tmin, tmax]
    Returns:
        tuple: (precision, recall). Both float.
    """
    true_pos = 0
    false_pos = 0
    false_neg = 0
    for idx in range(len(all_prediction_lines)):
        pline = all_prediction_lines[idx]
        gtline = all_gt_lines[idx]
        res = calculate_individual_record_result(pline, gtline,
        ↪  iou_threshold)
        true_pos += res["true_pos"]
        false_pos += res["false_pos"]
        false_neg += res["false_neg"]
    precision = calculate_precision(true_pos, false_pos, false_neg)
    recall = calculate_recall(true_pos, false_pos, false_neg)
    return precision, recall


def get_precision_recall_curve(
    all_prediction_lines, all_gt_lines, confidence_scores,
    ↪  iou_threshold
):
    """Given a set of prediction lines and ground truth lines for all
    ↪  records,
       calculates the recall-precision curve over all records.
       for a single record.
       NB: all_prediction_lines and all_gt_lines are not matched!
    Args:
        all_prediction_lines: (list of np.array of floats): each
↪   element in the list
            is a np.array containing all predicted bounding lines for
↪   the given record
            with shape: [number of predicted lines, 2].
            Each row includes [tmin, tmax]
        all_gt_lines: (list of np.array of floats): each element in the
↪   list
            is a np.array containing all ground truth bounding lines
↪   for the given record
            objects with shape: [number of ground truth lines, 2].
            Each row includes [tmin, tmax]
        scores: (list of np.array of floats): each element in the list
            is a np.array containting the confidence score for each of
↪   the
            predicted bounding line. Shape: [number of predicted lines]
            E.g: score[0][1] is the confidence score for a predicted
↪   bounding line 1 in record 0.
    Returns:
        precisions, recalls: two np.ndarray with same shape.
    """
```

```
    # Instead of going over every possible confidence score threshold
    ↪  to compute the PR
    # curve, we will use an approximation
    confidence_thresholds = np.linspace(0, 1, 1000)


    precisions = []
    recalls = []
    for confidence_thr in confidence_thresholds:
        temp_pred_line = []
        for record_idx in range(len(all_prediction_lines)):
            pred_lines = all_prediction_lines[record_idx]
            scores = confidence_scores[record_idx]
            prune_mask = scores >= confidence_thr
            temp_pred_line.append(pred_lines[prune_mask])
        precision, recall = calculate_precision_recall_all_records(
            temp_pred_line, all_gt_lines, iou_threshold
        )
        precisions.append(precision)
        recalls.append(recall)
    return np.array(precisions), np.array(recalls)


def plot_precision_recall_curve(precisions, recalls):
    """Plots the precision recall curve.
        Save the figure to precision_recall_curve.png:
        'plt.savefig("precision_recall_curve.png")'
    Args:
        precisions: (np.array of floats) length of N
        recalls: (np.array of floats) length of N
    Returns:
        None
    """
    plt.figure(figsize=(20, 20))
    plt.plot(recalls, precisions)
    plt.xlabel("Recall")
    plt.ylabel("Precision")
    plt.xlim([0.8, 1.0])
    plt.ylim([0.8, 1.0])
    plt.savefig("precision_recall_curve.pdf", format="pdf")


def calculate_average_precision(precisions, recalls):
    """ Given a precision recall curve, calculates the mean average
        precision.
    Args:
        precisions: (np.array of floats) length of N
        recalls: (np.array of floats) length of N
    Returns:
        float: mean average precision
    """
    # Calculate the mean average precision given these recall levels.
    recall_levels = np.linspace(0, 1.0, 1000)

    final_precisions = []
    for recall_level in recall_levels:
        precision = precisions[recalls >= recall_level]
        if precision.size == 0:
```

```
            precision = 0
        else:
            precision = max(precision)
        final_precisions.append(precision)
    average_precision = 0
    average_precision = np.mean(final_precisions)
    return average_precision


def average_precisions(
    targets,
    predictions,
    iou_threshold = 0.5
    ):
    """ Calculates the mean average precision over the given dataset
        with IoU threshold of 0.5
    Args:
        targets: (list of dicts, one for each record)
        [
        {
        "lines":[[onset0, offset0],[onset1,offset1], ...],
        "labels":[label0, label1, ...]
        },
        {...},
        ...
        ]
        predictions: (list of dicts, one for each record)
        [
        {
        "lines":[[onset0, offset0],[onset1,offset1], ...],
        "labels":[label0, label1, ...],
        "scores":[score0, score1, ...]
        },
        {...},
        ...
        ]
        iou_threshold: threshold of iou for a prediction
        to be considered a true positive.
    """
    average_precisions = []
    for label in range(1,6):
        label_pred_lines = []
        label_pred_scores = []
        label_gt_lines = []
        for rec_preds, rec_targs in zip(predictions, targets):
            #Indexing by boolean mask to only get lines/scores
            #where label == label
            pred_lines = rec_preds["lines"][rec_preds["labels"]==label]
            pred_scores =
             ↪  rec_preds["scores"][rec_preds["labels"]==label]
            gt_lines = rec_targs["lines"][rec_targs["labels"]==label]
            label_pred_lines.append(pred_lines)
            label_pred_scores.append(pred_scores)
            label_gt_lines.append(gt_lines)
        precisions, recalls = get_precision_recall_curve(
            label_pred_lines,
            label_gt_lines,
            label_pred_scores,
```

```
            iou_threshold)
        label_AP = calculate_average_precision(precisions, recalls)
        average_precisions.append(label_AP)
    return average_precisions
```

I Window-slide Inference script

```
"""
Scanline Sound Event Detection inference program
Written by Bendik Bogfjellmo in 2021
(github.com/bendikbo) (bendik.bogfjellmo@gmail.com)
Feel free to reuse, but let a brother get in them mentions

The main idea of the inference stuff is to do n hops per
classifying window. Meaning that each "subwindow" has n
classifications made on it, and then just average all these
confidence scores and just threshold those to get get them bools
"""
import torch
import pandas as pd
import pathlib
import argparse
from tqdm import tqdm
from torch.utils.data import DataLoader, dataloader
from torch import sigmoid
#Used to dereference from active neuron to class name
from classifier.data.datasets import dereference_dict
from classifier.models import build_model
from classifier.config.defaults import cfg
from classifier.utils import to_cuda
from classifier.utils import load_best_checkpoint
from classifier.data.datasets.window_slide import WindowSlide
import time


class Inferencer:
    def __init__(
        self,
        cfg,
        audio_files: list
        ):
        self.start_time = time.time()
        self.cfg = cfg
        self.audio_files = audio_files
        self.dataloaders = []
        self.dereference_dict = dereference_dict(self.cfg.INPUT.NAME)
        for audio_file in audio_files:
            self.add_dataloader(audio_file)
        self.model = self.load_model()
        self.model.eval()
        to_cuda(self.model)
        self.output_dir = self.cfg.INFERENCE.OUTPUT_DIR

    def add_dataloader(self, audio_file):
```

```
        dataset = WindowSlide(self.cfg, audio_file)
        dataloader = DataLoader(
            dataset=dataset,
            batch_size=self.cfg.INFERENCE.BATCH_SIZE,
            shuffle=False,
            num_workers=self.cfg.INFERENCE.NUM_WORKERS
        )
        self.dataloaders.append(dataloader)

    def load_model(self):

        model = build_model(self.cfg)
        checkpoint_dir = pathlib.Path(self.cfg.OUTPUT_DIR)
        state_dict = load_best_checkpoint(checkpoint_dir)
        model.load_state_dict(state_dict=state_dict)
        return model

    def infer(self):
        for audio_file, dataloader in zip(self.audio_files,
        ↪  self.dataloaders):
            self.start_time = time.time() #For performance measurements
            #output filename
            output_filename = audio_file.split("/")[-1].split(".")[0]
            #Instantiating predictions for entire audio file
            all_raw_preds = torch.zeros(
                len(dataloader.dataset),
                len(self.dereference_dict)
            )
            #Size [Predictions_per_file, Num_classes]
            all_raw_preds = to_cuda(all_raw_preds)
            with torch.no_grad():
                for x_batch, indices in tqdm(dataloader):
                    x_batch = to_cuda(x_batch)
                    #Have to convert tensor to long for indexing
                    indices = indices.long()
                    indices = to_cuda(indices)
                    preds = self.model(x_batch)
                    preds = sigmoid(preds)
                    #Just to make absolutely sure everything's in order
                    all_raw_preds[indices] = preds
            num_hops = self.cfg.INFERENCE.HOPS_PER_WINDOW
            #processed_preds.size() == (predictions_per_file + num_hops
            ↪  - 1, num_classes)
            processed_preds = torch.zeros(
                (all_raw_preds.size()[0]+num_hops - 1,
                all_raw_preds.size()[1])
            )
            #For loop to make processes preds into moving average
            #of raw preds based on number of hops per class window
            all_raw_preds = all_raw_preds.cpu()
            processed_preds = processed_preds.cpu()
            for hop_no in range(num_hops):
                #Have to add if statement for last hop, since
                #apparantly, there's no elegant way to do this
                if hop_no < (num_hops - 1):
                    processed_preds[hop_no:-num_hops + hop_no + 1,:] \
                    += all_raw_preds[:,:]/num_hops
                else:
```

```
                processed_preds[hop_no:,:]\
                += all_raw_preds[:,:]/num_hops
        if num_hops > 1:
            num_labels=processed_preds.size()[-1]
            #This is done to fix edge cases in moving mean method
            numerator=torch.linspace(1,num_hops-1, num_hops-1)
            denominator = torch.linspace(num_hops-1, 1, num_hops-1)
            edge_multiplier=torch.div(numerator, denominator)
            #Have to repeat multiplier for each label type
            edge_multiplier=\

            ↪  edge_multiplier.view(-1,1).repeat(1,num_labels).view(num_hops-1,nu
            processed_preds[-num_hops + 1:,:]+=\
            torch.mul(
                edge_multiplier,
                processed_preds[-num_hops+1:,:]
                )
            #Flip it around and bring it back
            edge_multiplier = torch.div(denominator, numerator)
            edge_multiplier=\

            ↪  edge_multiplier.view(-1,1).repeat(1,num_labels).view(num_hops-1,nu
            processed_preds[0:num_hops-1,:]+=\
            torch.mul(
                edge_multiplier,
                processed_preds[:num_hops-1,:]
            )
        self.format_and_write(output_filename, processed_preds)

    def format_and_write(
        self,
        output_filename : str,
        processed_preds : torch.Tensor
        ):

        predictions = []
        threshold = self.cfg.INFERENCE.THRESHOLD
        for label in range(processed_preds.size()[1]):
            label_name = self.dereference_dict[label]
            label_preds = processed_preds[:,label]
            pos_preds = torch.zeros_like(label_preds)
            pos_preds[label_preds >= threshold] = 1
            i = 0
            pred_start = 0.0
            pred_stop = 0.0
            active_pred = False
            hop_length_in_seconds = \

                ↪  (self.cfg.INPUT.RECORD_LENGTH//self.cfg.INFERENCE.HOPS_PER_WINDOW)
                ↪  \
                self.cfg.INPUT.SAMPLE_FREQ
            num_preds = int(pos_preds.size()[0])
            #Couldn't figure out a prettier way to do this
            while i < num_preds:
                if pos_preds[i]:
                    if not active_pred:
                        pred_start = i*hop_length_in_seconds
                    active_pred = True
```

```
                elif pos_preds[i-1] and active_pred:
                    active_pred = False
                    pred_stop = i*hop_length_in_seconds
                    predictions.append((label_name, pred_start,
                    ↪  pred_stop))
                i+=1
        if self.cfg.INFERENCE.OUTPUT_FORMAT == "csv":
            self.write_to_csv(predictions, output_filename)
        elif self.cfg.INFERENCE.OUTPUT_FORMAT == "audacity":
            self.write_audacity_labels(predictions, output_filename)

    def write_to_csv(
        self,
        predictions,
        output_filename
        ):
        df = pd.DataFrame(
            predictions,
            columns=["class", "onset", "offset"]
            )
        pathlib.Path(cfg.INFERENCE.OUTPUT_DIR).mkdir(
            parents=True,
            exist_ok=True
        )
        output_path = self.cfg.INFERENCE.OUTPUT_DIR + output_filename +
        ↪  ".csv"
        df.to_csv(output_path)
        print(f"\n\nFinished inference on: {output_filename}")
        print(f"time used: {time.time() - self.start_time}")
        print(f"ouput stored as: {output_path}")

    def write_audacity_labels(
        self,
        predictions,
        output_filename
        ):
        pathlib.Path(cfg.INFERENCE.OUTPUT_DIR).mkdir(
            parents=True,
            exist_ok=True
        )
        stuff_to_write = ""
        for prediction in predictions:
            first_line =
            ↪  f"{prediction[1]}        {prediction[2]}        {prediction[0]}\n"
            second_line = f"\\        000.000        000.000\n"
            stuff_to_write += first_line + second_line
        #Audacity label files are .txt extension
        output_path = self.cfg.INFERENCE.OUTPUT_DIR + output_filename +
        ↪  ".txt"
        out_file = open(output_path, 'w+')
        out_file.write(stuff_to_write)
        out_file.close()
        print(f"\n\nFinished inference on: {output_filename}")
        print(f"time used: {time.time() - self.start_time}")
        print(f"ouput stored as: {output_path}")
```

```
def get_parser():
    parser = argparse.ArgumentParser(
        description='Sound Event Detection inference')
    parser.add_argument(
        "config_file",
        default="",
        metavar="config_file",
        help="path to config file",
        type=str,
    )
    parser.add_argument(
        "audio_files",
        help="Audio files to run inference on",
        default=None,
        nargs=argparse.REMAINDER,
    )
    return parser


def main():
    args = get_parser().parse_args()
    cfg.merge_from_file(args.config_file)
    audio_files = args.audio_files
    cfg.freeze()
    output_dir = pathlib.Path(cfg.OUTPUT_DIR)
    output_dir.mkdir(exist_ok=True, parents=True)
    print("Loaded configuration file {}".format(args.config_file))
    with open(args.config_file, "r") as cf:
        config_str = "\n" + cf.read()
        print(config_str)
    print("Running with config:\n{}".format(cfg))
    inferencer = Inferencer(cfg, audio_files)
    inferencer.infer()


if __name__ == "__main__":
    main()
```

J Classifier Models

J.1 ResNet50 & ResNet34

```
import torch.nn as nn
from torchvision.models import resnet50
from collections.abc import Iterable

class ResNet50BB(nn.Module):
    """
    Resnet50 classifier class
    """
    def __init__(self, cfg):
        super(ResNet50BB, self).__init__()
        if getattr(cfg, "PRETRAINED", True) == False:
            self.pretrained = False
        else:
```

```
            self.pretrained = True
        self.res = resnet50(pretrained=self.pretrained)
        self.fc = nn.Sequential(
            nn.Linear(in_features=1000, out_features=cfg.NUM_CLASSES)
        )
    def forward(self, x):
        x = self.res(x)
        x = self.fc(x)
        return x


class ResNet34BB(nn.Module):
    """
    Resnet34 classifier class
    """
    def __init__(self, cfg):
        super(ResNet50BB, self).__init__()
        if getattr(cfg, "PRETRAINED", True) == False:
            self.pretrained = False
        else:
            self.pretrained = True
        self.res = resnet34(pretrained=self.pretrained)
        self.fc = nn.Sequential(
            nn.Linear(in_features=1000, out_features=cfg.NUM_CLASSES)
        )
    def forward(self, x):
        x = self.res(x)
        x = self.fc(x)
        return x
```

J.2 EfficientNet

```
from efficientnet_pytorch import EfficientNet


def effnet(cfg):
    return EfficientNet.from_pretrained(cfg.NAME,
    ↪  num_classes=cfg.NUM_CLASSES)
```

Bendik Bogfjellmo

Two deep learning approaches to Sound Event Detection for bird sounds in the arctic biosphere

NTNU
Norwegian University of
Science and Technology