ESTONIAN INFORMATION TECHNOLOGY COLLEGE

Tanel Peet

# AUTOMATIC CLASSIFICATION OF BIRD VOCALIZATIONS USING CONVOLUTIONAL NEURAL NETWORKS

Diploma Thesis

**INFOTECHNOLOGICAL SYSTEMS DEVELOPMENT COURSE**

Supervisor: Professor Carlos Ferreira

Consultant: Linnar Viik

Tallinn 2017

# AUTORIDEKLARATSIOON

Deklareerin, et käesolev diplomitöö, mis on minu iseseisva töö tulemus, on esitatud Eesti Infotehnoloogia Kolledžile lõpudiplomi taotlemiseks Infosüsteemide arendamise erialal. Diplomitöö alusel ei ole varem eriala lõpudiplomit taotletud.

Autor T. Peet: .....

Töö vastab kehtivatele nõuetele

Juhendaja C. Ferreira: .....

# Acknowledgement

I would like to thank INESC TEC and its Laboratory of Artificial Intelligence and Decision Support (LIAAD) for giving me the opportunity to do research on a topic I was interested in and which was finally formed into this thesis. I appreciate the support, feedback and suggestions made by my colleagues from LIAAD - Juan Colonna, João Gama, Alípio M. Jorge, Elsa Ferreira Gomes, but especially my supervisor, Carlos Abreu Ferreira. In addition I would like to thank Matthew Davies from the Sound and Music Computing Research group from INESC TEC for getting me into contact with experts in the field and giving me new ideas.

I appreciate the effort of Liisa Merila who pointed out grammatical errors in this thesis and even in this sentence. I would like to thank Proekspert, who supported me and helped me in different ways with this work.

I would like to thank Linder team who has helped out with giving me new ideas and making sure that the results of this work would be used and made available for everyone. I appreciate the answers I got from Riho Kinks from Estonian Ornithological Society and Veljo Runnel, an acknowledgest nature sound recorder and creator of the My Nature application.

Finally I want to thank everyone who has contributed to Xeno-canto or My Nature Sound database and made the data collection an easy task for scientists.

# Contents

# Acronyms

| | |
|---|---|
| ANN | Artificial Neural Network |
| API | Application Programming Interface |
| CNN | Convolutional Neural Network |
| CPU | Central Processing Unit |
| DCT | Discrete Fourier Transform |
| ELU | Exponential Linear Unit |
| FC | Fully Connected (layer) |
| FFT | Fast Fourier Transform |
| FN | False Negative |
| FP | False Positive |
| GPU | Graphics Processing Unit |
| ReLU | Rectified Linear Unit |
| STFT | Short-Time Fourier Transform |
| TN | True Negative |
| TP | True Positive |

# 1 Introduction

The goal of this thesis is to propose and implement a machine learning solution for automatically classifying the most popular bird species in Estonia. The proposed solution would be the first prototype, which would be improved in future works. Therefore, no strict goal for the accuracy is set, although it should be clear that the proposed algorithm was able to learn from the data given to it.

The birds play important role in ecosystems and can be used as indicators of ecological stress (Carignan & Villard, 2002). Being able to automatically classify bird species would improve monitoring of birds. In addition, the solution can be used to raise awareness and demonstrate the importance of birds, by creating a mobile application for classifying bird vocalizations. Although the creation of the mobile application is not in the scope of this work, it is kept in mind when developing the machine learning algorithm. The need for this kind of algorithm has been confirmed by the member of Estonian Ornithological Society, Riho Kinks, and one of the most famous nature sounds experts in Estonia, Veljo Runnel.

This thesis will look at different research papers to find a solution which can then be modified to fit the problem defined in this work. The differences between the problems described in related works and the current work are being analyzed and a solution is found that would fit the problem of this thesis. The databases needed for gathering data for training and evaluating the machine learning algorithm, are being described and analyzed. The species, which will be included in the classification task, are being selected based on the information gathered from these databases. The software frameworks and programming languages are being analyzed to find the best ones for the problem at hand. The algorithms

are implemented which would allow to preprocess audio recordings, classify them, evaluate the performance and allow to perform experiments for choosing good parameters for the problem. These algorithms and experiments are also being described and analyzed in this work.

Everything described in this work is analyzed, proposed and implemented by the author of this work, although improvements have been made due to the mistakes and shortcomings noticed by the supervisor. The idea behind the final solution is provided by the work of others, but the implementation is made by the author with some modifications in order to fit the problem.

The structure of this thesis is divided into several chapters. In Chapter 2 the background information about the problem is discussed, including the importance of it, description of different image processing and machine learning approaches. Chapter 3 analyzes different available solutions, databases and development tools, based on the information from Chapter 2. Chapter 4 describes the implementation, based on previous analysis, discusses how the solution was implemented and which parameters were used for machine learning algorithm based on the experiments done. Results from the experiments and final implementation are being described and analyzed in Chapter 5. The thesis is summed up in Chapter 6 together with suggestion for future work.

# 2 Background

The theory and background information about the task of classifying birds by their vocalization is described in this chapter. The importance of birds and the need for classifying them automatically is described in Section 2.1. The problem, goals and scope of this project are defined in Section 2.2. The theory behind signal processing and machine learning, which is needed for solving the defined problem, is discussed in Sections 2.3 and 2.4.

## 2.1 Importance

Biological diversity plays an important roll in ecosystem functioning and in the provision of ecosystem services, which are essential in supporting human existence. Biological diversity is increasingly threatened by increase in human population, economic growth, land use and climate change. As a consequence, biodiversity has shown signs of decrease at unprecedented rates. (Christie, Fazey, Cooper, Hyde, & Kenter, 2012)

Maintaining or restoring the ecological integrity of an ecosystem requires indicators. Birds have been shown to respond to environmental changes, they are relatively easy to detect and identify by their vocalization and they can be monitored over large spatial scales. Birds occurrence and reproductive success have been shown to be influenced by the nature and configuration of surrounding habitats (Carignan & Villard, 2002). In addition it is shown by (Böhning-Gaese & Lemoine, 2004) that climate change affects breeding biology and population dynamics of birds. Therefore, birds can be one of the indicators of ecological integrity.

Many biologists count and identify birds in a specific area to estimate long-term population

trends of different bird species, because birds can be easily observed by specialists. Most people often hear the birds, rather than see them and most of the bird vocalizations have evolved to be species specific, which means that automatically classifying bird species from their vocalizations is a natural and adequate way for observing birds. (Lee, Han, & Chuang, 2008)

## 2.2 Goals and Project Scope

There is a lot of research available in the field of automatically classifying bird vocalizations, but only a couple of solutions were available for public use. Warblr mobile application (Warblr LTD, 2015) and Bird Song Id Automatic UK application (Sunbird Images, 2015) are available for iOS and Android devices, but are not free. In addition, these application are marketed for UK or other European countries, from which the closest to Estonia is Sweden. In Estonia, no such application exists by the knowledge of the author of this paper and the representative of Estonian Ornithological Society, Riho Kinks.

The goal of this work is to develop an algorithm for automatically classifying the most popular bird species in Estonia. The performance of the algorithm should be evaluated in order to see how well it was able to learn from data. This algorithm can later be used by experts to monitor species in Estonia but also to raise awareness about the importance of birds and their environment. The solution proposed by this work is the first step towards bird song recognition for Estonian species and the work continuous in the future. Only a small number of ideas and techniques can be implemented, tested and described due to the time and size limitation related to this thesis.

The first real-life implementation of this algorithm will be a mobile application for recording and recognizing bird vocalizations. Although the development of a mobile application is not in the scope of this project, the requirements for using the solution in mobile devices has taken into consideration. In addition, the usage of lower computational power devices, such as (Raspberry Pi, 2016), for analyzing field recordings in real time, has taken into account when developing the algorithms for recognizing bird vocalizations.

## 2.3 Signal Processing

This section describes different methods used in signal preprocessing, before it can be fed to machine learning algorithm. In Section 2.3.1, the definition of spectrogram is given together with a explanation of how it can be created from raw audio signal. Section 2.3.2 describes different image processing techniques, which can be implemented on spectrograms for removing noise and filtering out signal from noise.

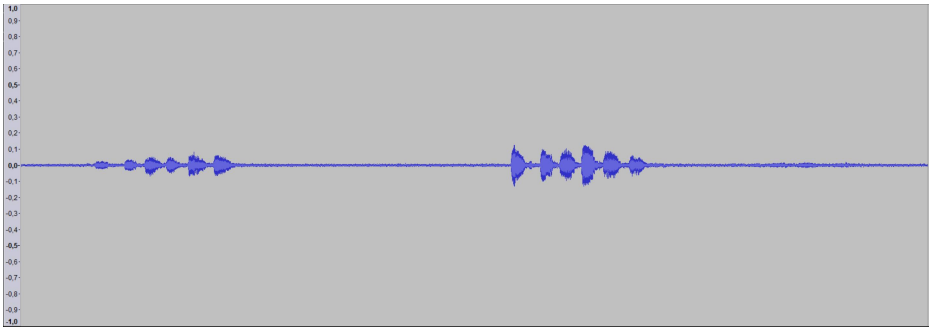
### 2.3.1 Spectrogram

Spectrogram, also called sonogram, is a visual representation of the spectrum of frequencies in a signal. It is often used in research related to various sounds, sonars, radars and seismology. Spectrogram gives more intuitive description of the signal, than the plotting of regular signal (waveform). Raw signal and the waveform representation of it, represents an amplitude of that signal on the vertical axis. In spectrograms, the frequencies are represented on the vertical axis instead, which makes it a good option for solving many problem where frequencies play important role. On Figure 2.1 the difference between waveform and spectrogram, of the same audio recording, can be seen. The creation of spectrogram is explained in the following subsections by explaining the concepts of Fourier transform and short-time Fourier transform. (Smith et al., 1997)

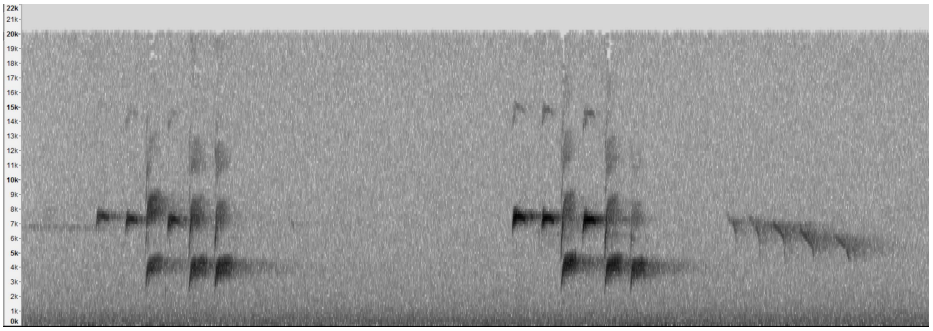
#### Fourier Transform

Fourier transform is the output of the Fourier transformation where function of time, also called signal, is decomposed into the frequencies that make it up. Fourier transform of a signal is a complex-valued function of frequency, where the absolute value represents the amount of that frequency present in the original signal. That is the reason why Fourier transform is also called the frequency domain representation (Bracewell, 1965). In computer science the signal is usually discrete, sampled at fixed time steps. For discrete signals the discrete Fourier transform (DFT) can be computed (Weisstein, 2002).

Fast Fourier transform (FFT) is any algorithm which can create DFT in time complexity of $\mathcal{O}(N \log N)$ or less, compared to evaluating the DFT definition directly, which has time complexity of $\mathcal{O}(N^2)$. FFT produces the same results as evaluating the DFT definition



*(a)* Waveform representation, where time is on horizontal axis and amplitude of the signal on vertical axis



*(b)* Gray-scale representation of the spectrogram, where time is on horizontal axis and frequencies on the vertical axis

*Figure 2.1.* Waveform and spectrogram of a Great Tit audio recording

directly, but it is much faster and can be more accurate in the presence of round-off error. The most common FFT algorithm is the Cooley-Tukey algorithm, which is described by Equation 2.1, where $X_k$ is the $k$-th an element of Fourier transform vector, $N$ the number of signal samples and $x_n$ the $n$-th element of the signal. This algorithm works best if the amount of samples is a power of two. (Cooley & Tukey, 1965)

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i \frac{nk}{N}} \qquad k = 0, \ldots, n-1. \tag{2.1}$$

## Short-Time Fourier Transform

Fourier transform of a signal has little practical value if the signal is evolving in time in an unpredictable way. With complex signals it would be more useful to see how frequency amplitudes evolve in time. The signal can be divided into smaller overlaping slices of equal length and the Fourier transform can be computed on these slices. The complex results are added to a matrix, which records magnitude and phase for each point in time and frequency.

(Cooley & Tukey, 1965)

The crude slicing of the signal can create artifacts in the final result, since Fourier transform will interpret the jumps at the boundaries of the signal slices as an abrupt variation of the signal. The concept of windowing has been introduced for avoiding these artifacts. A smooth window-function, which is close to one near the origin and decays towards zero at the edges, can be used for slicing. Some of the popular window functions are called Hamming, Hanning, Barlett and Kaiser. The resulting time-frequency analysis procedure is referred to as short-time Fourier transform (STFT). (Cooley & Tukey, 1965)

### 2.3.2 Image Processing

Image processing techniques can be used on spectrograms for noise reduction and for filtering out useful signal. Some image processing techniques, commonly used on spectrograms, are discussed in the following subsections.

#### Convolution

Convolution is used by many commonly used image processing operators. It provides a way to combine together two arrays of numbers, usually of different sizes, but same dimensionality. In image processing one of the input arrays is normally a two dimensional image layer. The second input array, known as kernel, is also two dimensional and defines how the original image is being transformed. (Fisher, Perkins, Walker, & Wolfart, 1994)

Convolution is performed by sliding the kernel over the image. Each kernel corresponds to a single output pixel, which value is calculated by multiplying together the kernel value and the underlying image pixel value for each of the cells in the kernel, and then adding all these values together. (Fisher et al., 1994)

#### Morphological Operations

Morphological operators are usually used on binary images. They take a binary image and structuring element, called kernel, as inputs and combine them using a set operator. The kernel is usually a square matrix which is shifted over the image and for each image pixel

the kernel elements are compared with the set of the underlying image pixels. If the pixels from original image matches with the pixels from kernel, by the condition defined by the set operator, the pixel from original image is set to predefined value (zero or one). All the morphological operators are combination of erosion and dilation operators. (Fisher et al., 1994)

The basic effect of erosion operator is to erode away the boundaries of regions of foreground pixels. This results in shrinking the size of foreground pixels and holes within those areas become larger. Erosion operator has two inputs - the image it is applied on and the kernel, which determines the precise effect of the erosion on the input image. If every pixel from erosion kernel matches the pixels from underlying image, then the corresponding image pixel remains the same. (Fisher et al., 1994)

The dilation operator is very similar to erosion operator, the only difference being that if at least one pixel from dilation kernel matches the pixel from underlying image, then the corresponding image pixel is set to foreground pixel's value. The basic effect of dilation operator is to gradually enlarge the boundaries of regions of foreground pixels, therefore areas of foreground pixels grow in size while holes within those regions become smaller. Dilation operator is used to fill holes in binary images. (Fisher et al., 1994)

Morphological opening operator is defined as an erosion which is followed by a dilation, where both operators use same kernel. Opening operation removes some of the foreground pixels from the edges of regions of foreground pixels, but it is less destructive than erosion in general. Figure 2.2 illustrates morphological opening operation. (Fisher et al., 1994)

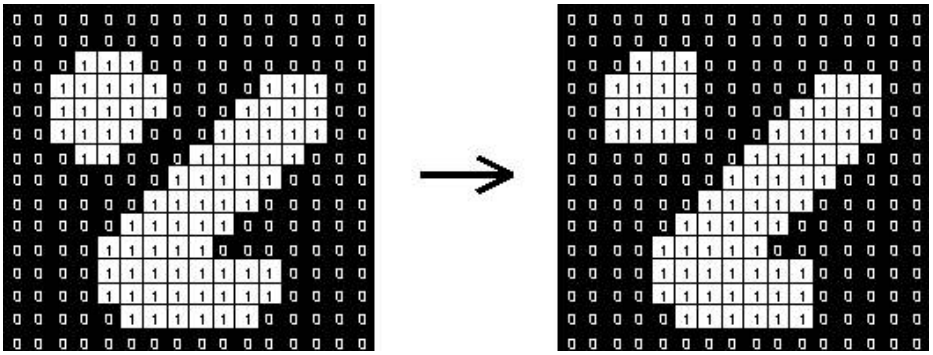


*Figure 2.2.* Example of morphological opening (Fisher et al., 1994)

Morphological closing is like opening, but done in reverse - dilation is followed by an erosion which both use the same kernel. It enlarges the boundaries of foreground areas in an image and shrinks background holes in such regions. (Fisher et al., 1994)

### Median Clipping

Median clipping, as defined by (Lasseck, 2013), is the process of removing noise from an image. The value of each pixel in the image is compared to the median value of the corresponding column and row. If it exceeds some constant, the pixel is set to one, otherwise it is zero. Therefore the result is a binary image.

## 2.4 Machine Learning

Machine learning is a sub-field of computer science, where algorithms are able to learn from data, without being explicitly programmed. It involves studying and designing algorithms which can learn from data to accomplish certain tasks. Machine learning is being used in a variety of tasks where programming explicit algorithms is unfeasible, such as computer vision, speech recognition and spam filtering. (Goodfellow, Bengio, & Courville, 2016)

Machine learning can be useful for different kinds of tasks. Some of the most common machine learning tasks include, but are not limited to, classification, regression, transcription, machine translation, structured output, anomaly detection, synthesis and sampling, imputation of missing values and denoising. This section concentrates only on classification tasks. (Goodfellow et al., 2016)

Classification is the problem of identifying which category a new observation belongs to. Machine learning algorithm needs a set of data containing observations whose category is known. In order to learn to classify unknown data samples, the classifier needs to be trained. Training phase involves presenting machine learning algorithm with data samples which are paired with their labels. The algorithm then tries to learn from the data samples and from the corresponding labels, in order to be able to predict the label of the samples which it has not seen yet. The result of training is a machine learning model, which can then be used

for classifying. In order to understand how well the machine learning algorithm performs the model needs to be evaluated, or tested, with data it has not seen yet. (Goodfellow et al., 2016)

An overview of different sets of data used in machine learning is provided in Subsection 2.4.1. Several performance metrics, for evaluating machine learning algorithms, are described in Subsection 2.4.2. Overview of a few well known machine learning algorithms, which are often used in classification tasks, can be found in Subsection 2.4.3.

### 2.4.1 Data in Machine Learning

The set of data used for training and testing machine learning algorithms is called dataset. Dataset is usually divided into training and testing set, but good practice is to also have an validation set. All these subsets should be independent from each other in order to obtain reliable accuracy measurements. Traditional classification problems require that every data sample also has a corresponding label. (Goodfellow et al., 2016)

As the name states, training data is used by the algorithm, so it could learn from the data in order to solve the given task. Evaluation data is used for fine tuning different parameters of the preprocessing or machine learning algorithm. After being satisfied with the results evaluated on validation data, testing data is used for final evaluation of the selected model. Separate testing data is needed because fine tuning parameters might have resulted in choosing parameters which are good for this subset of data, but are not performing well on new samples. (Goodfellow et al., 2016)

Most machine learning algorithms work better if training dataset is balanced, which means that each class needs to have a similar number of samples. If dataset is very unbalanced, the model might learn to predict more often the classes which have more examples, because it gives better performance on training data. Avoiding this problem can be solved by gathering more data or by undersampling or oversampling the training data, if gathering additional data is not possible. Undersampling is the process of removing samples from the classes which have more training data. Oversampling is the process of adding new samples to the classes with less training data, by duplicating samples. Oversampling can be also achieved

by data augmentation, which involves slightly modifying the duplicated data in minority classes. (Goodfellow et al., 2016)

### Overfitting and Underfitting

Overfitting occurs when model does not generalize well to unseen samples, although it performs well on training data. This is usually caused by the model, which is too complex, or in the case when there is not sufficient training data available. Overfitting means that the model has memorized the training data instead of learning from it. To avoid overfitting one can either gather or create additional training data, or use a less complex model. There are also additional methods for reducing overfitting, such as explicitly penalizing overly complex models. (Goodfellow et al., 2016)

Underfitting occurs when model is not able to learn the underlying trend of the data. The result of underfitting is that model is not able to fit training data enough. Underfitting is a result a model which is too simple. (Goodfellow et al., 2016)

Overfitting and underfitting are illustrated on Figure 2.3

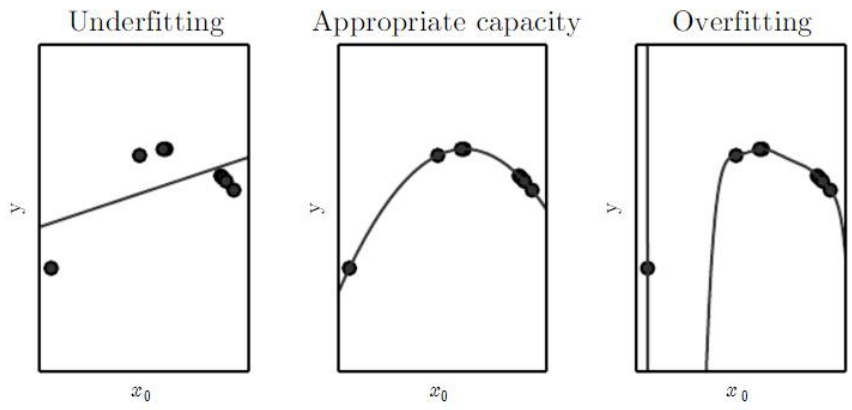



*Figure 2.3.* The cases of underfitting, approriate complexity of the model and overfitting in regression task (Goodfellow et al., 2016)

## 2.4.2 Evaluation Metrics

In binary classification tasks, a confusion matrix is often used to describe the performance of the model. The binary classification task involves predicting whether the sample is from positive or negative class. Confusion matrix for binary classification consists of four elements - number of true positives (TP), number of true negatives (TN), number of false

positive (FP) and number of false negatives (FN). Confusion matrix is illustrated by Table 2.1. True positives are the cases where the actual sample was from positive class and the prediction was also from positive class. True negatives are cases where sample was from negative class and the prediction was from negative class. False positives are the cases where sample was from negative class, but prediction was from positive class. False negatives are the cases where sample was from positive class, but prediction from negative class. (Japkowicz & Shah, 2011)

**Table 2.1**
Confusion matrix for binary classification tasks

| | **Positive (predicted)** | **Negative (predicted)** |
|---|---|---|
| **Positive (actual)** | Number of True Positives | Number of False Positives |
| **Negative (actual)** | Number of False Negatives | Number of True Negatives |

Accuracy, in the context of machine learning, is the performance measure which is the ratio between the number of correct predictions and number of total predictions. Using accuracy on unbalanced data might result in a model which shows good accuracy but actually does not solve the problem it was intended to do. This is situation is called accuracy paradox and happens when model always predicts the class which has the most data. (Japkowicz & Shah, 2011)

Precision represents how often the model is correct, when positive class is predicted. It is the ratio between true positives and number of positive class predictions. Recall represents how often it predicts positive class, when the sample is actually also from positive class. It is the ratio between true positives and number of samples which are actually from positive class. F-score is a measure of a test's accuracy, which considers precision and recall, by taking harmonic mean from them. This takes the class unbalance into account and is better metric than accuracy for problems with unbalanced classes. (Japkowicz & Shah, 2011)

All these metrics above can be also applied to evaluate classifiers which need to classify more than two classes.(Japkowicz & Shah, 2011)

For multi-class classification problems top-n accuracy is often used, where n presents some integer, which is smaller than the number of classes involved in the task. Top-n accuracy shows how often is the real class label of a sample included in the top *n* predictions for this sample. It is defined as the ratio of the number of times the true label of the sample is in top *n* predictions of this sample and the total number of samples. (Japkowicz & Shah, 2011)

### 2.4.3 Machine Learning Algorithms

There are many different kind of machine learning algorithms used for classification task. This section concentrates on the algorithms which are often used in bird classification tasks. In the following subsections random forest, artificial neural networks and convolutional neural networks are described.

#### Random Forest

Random forest is an ensemble learning method, that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes. Compared to decision trees, Random forest corrects the habit of overfitting to their training set. (Goodfellow et al., 2016)

Decision trees can be used as a predictive model, that maps observations about an item to conclusions about the item's target value. Observations about an item are represented in branches, while conclusions about the item's target value are represented in leaves of the decision tree. (Goodfellow et al., 2016)

#### Artificial Neural Networks

Artificial neural network (ANN) is a machine learning algorithms which contains large collection of nonlinear elements, called neurons, which are connected with many others. Each neural unit usually has a summation function which combines the values of all its inputs together. Simple ANN-s typically consist of multiple layers, where neurons from one layer are connected to all the neurons of the next siding layer. These layers are also called fully connected (FC) layers. (Goodfellow et al., 2016)

The first layer of ANN is called input layer and the last is called output layer. All the layers in between are called hidden layers. Each connection between two neurons has a weight. Input and hidden layers each has additional neuron called bias, which is added to increase the flexibility of the model to fit the data. The value of the neurons in hidden and output layers are defined by adding together the multiplications of the neurons from previous layer, and weights for the corresponding connections, followed by applying activation function on the result. Training of ANN changes the weights and biases to achieve the minimum value of a loss function. Example of such ANN with one hidden layer is seen on Figure 2.4. (Goodfellow et al., 2016)

ANN-s which have several hidden layers, are called deep neural networks and are a subset of deep learning algorithms. In deep learning the algorithms attempt to represent the information about the world as a nested hierarchy of concepts, where each concept is defined in relation to simpler concepts. For example, in face recognition task the lower level concepts can be edges detected from the image. The next level of abstraction could be features of the face, like nose, eye or a mouth, learned from the lower level concepts. The features of the faces can finally be used to learn concept of the face. (Goodfellow et al., 2016)

In ANN, an activation function is applied on the linear combination of weights and values of the neurons from previous layer, in order to get the value of the neuron in hidden or output layers. The role of the activation function is to produce a non-linearity to the model by non-linear combinations of the weighted inputs. This is needed because most of the real world problems are of non-linear nature. An activation function was introduced by (Clevert, Unterthiner, & Hochreiter, 2015), called exponential linear unit (ELU), which allows faster learning and leads to higher classification accuracy than previous activation functions. For output layers, Softmax layer is often used, which gives a vector of values, summing up to one. Therefore the output of the Softmax layer can be viewed as a probability distribution, where each value represent the probability of the corresponding class. (Goodfellow et al., 2016)

The main goal of ANN-s is to reduce the data loss by changing the values of weights and biases. For this, a loss function is used, which output represents the difference between the predictions and the real label values of the given sample. Loss function shows the quality

of the chosen weights and biases for the given classification task. In order to reduce data loss, gradient descent is often used. Gradient descent uses gradients of the loss function, in order to find how to change the parameters of weights and biases. It can be looked as following the direction of the slope of the surface created by the loss function to go downwards, towards lower loss function value. Mini-batch gradient descent is often used in ANNs, where gradient descent performs an update for some number of training samples at once, while this number remains between one and the total number of training samples. Gradient descent has a hyper-parameter called learning rate, which defines the length of the step for each update. If the value of learning rate is too large, the algorithm might start fluctuating around the minimum, if it is too small, the learning might be very slow. (Ruder, 2016)

There are several problems with using gradient descent and therefore several gradient descent optimization algorithms are created to tackle these problems. Gradient descent has trouble navigating the areas where the surface curves much more steeply in one dimension than other, which is common for areas around local optima (Sutton, 1986). Optimization algorithm, called momentum, solves that problem by helping gradient descent to accelerate in the relevant direction by adding a fraction of the update vector from the past time step to the current update vector. This fraction is defined as momentum parameter. One improvement of the momentum algorithm is the Nesterov accelerated gradient. This approach can predict the approximate future position for the parameters, so it can correct the direction of change and prevent it for taking too long steps. This results in increased responsiveness and performance in mini-batch gradient descent. (Ruder, 2016)

When the ANN is too complex or there is not enough training data available, regularization methods are often used to reduce overfitting. One such method is called dropout, which randomly drops neurons with their connections during the training, from the layer it is applied to. The probability that neuron is dropped out is specified by a hyper-parameter that can be changed, in order to find out the optimal dropout probability for the model. Dropout has shown good results in reducing overfitting in computer vision, speech recognition, document classification and computational biology tasks. (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014)

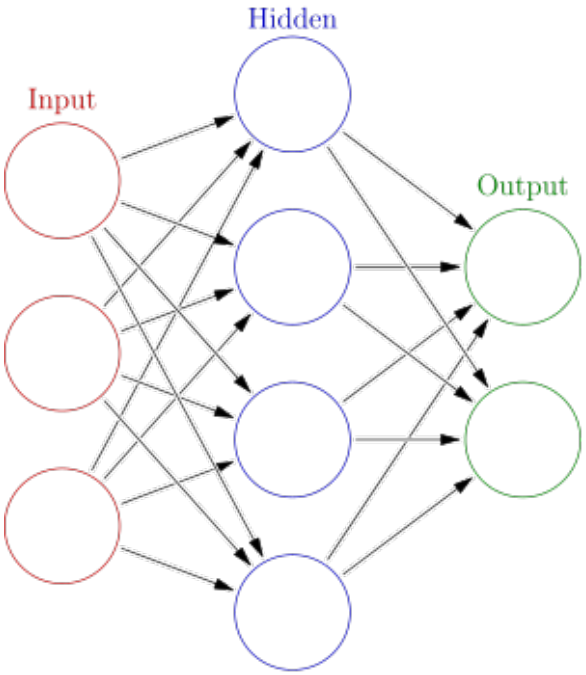



*Figure 2.4.* Example of simple Artificial Neural Network (Glosser.ca, 2013)

## Convolutional Neural Networks

Convolutional neural network (CNN) is a type of artificial neural network that uses convolution in place of general matrix multiplication, in at least one of its layers. Convolutional layers help to reduce the amount of parameters that are needed in learning process. CNNs allow the use of multi-dimensional arrays, called tensors, as an input. Therefore CNN-s are often used in image classification tasks, because an image can be viewed as a 3-dimensional tensor, where the width and height correspond to the width and height of the image and depth contains, in case of colored image, three layers indicating the intensity of red, blue and green. Example of CNN, for classifying objects from image, can be seen on figure 2.5. (Goodfellow et al., 2016)

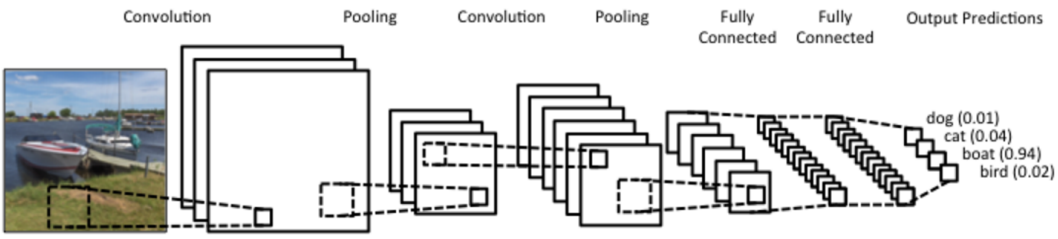



*Figure 2.5.* Example of CNN for image classification task (Britz, 2015)

The parameters of convolutional layer consist of a set of learnable kernels, which are smaller than the input tensors, but use the full depth of the input volume. Each kernel is convolved across the width and height of input, producing a 2-dimensional activation map of that kernel. This results in algorithm learning the kernels that activate when it sees some specific type of feature at some area in the input. The output volume of convolutional layer is achieved by stacking together activation maps generated by the convolution of the kernel. Three parameters are used for changing the size of the output volume of the convolutional layer - depth, stride and zero-padding. Depth of the output volume shows the number of neurons in the output volume that connect to the same region of the input volume. Stride shows how much the kernel moves with each step when constructing the activation map. Zero-padding is used to pad the input volume with zeros on the border of the input volume. (Goodfellow et al., 2016)

After the convolutional layer, the pooling layer is often used. Pooling layer is used to reduce the spatial size of the input volume in order to reduce the amount of parameters, which results in reducing the amount of computations and overfitting. The pooling layer operates independently on every activation map layer. Pooling layer uses rectangular filters to stride over the input layers, where the step size is called stride. When the filter strides over the layer of the input volume it applies some operator to output a single number. Usually finding the maximum of the underlying layer's value is used as an operator, which case the pooling layer is called max pooling layer. (Goodfellow et al., 2016)

The last layers of convolutional neural networks are usually fully connected layers. The data from convolutional layer must be reshaped into a one dimensional vector before giving it as an input to the fully connected layer. (Goodfellow et al., 2016)

# 3 Analysis

The purpose of this chapter is to analyze different approaches for bird vocalization classification and find a solution for classifying birds found in Estonia. The state of the art research is described in Section 3.1. Section 3.2 analyzes availability of the bird recordings and the question of how to combine several databases for gathering training and testing data. Different classifiers from previous researches are being analyzed in Section 3.3, concentrating on the best suited one. Different preprocessing steps, which are needed to be done before feeding the data to the chosen classifier, are described and analyzed in Section 3.4. The evaluation methods for fine-tuning parameters and choosing the optimal model, are described in Section 3.5. Finally, the selection of development tools and libraries are analyzed in Section 3.6, in order to choose the best framework and programming language for the task in hand.

## 3.1 Related Work

There have been several competitions for automatically recognizing bird species by their vocalization since 2013. In 2013, two competitions were held. The first was organized as a part of IEEE International Workshop on Machine Learning for Signal Processing (MLSP) (Briggs, Raich, Eftaxias, Lei, & Huang, 2013) and second as a part of The Neural Information Processing Scaled for Bioacoustics (NIPS4B) workshop (NIPS4B, 2016) . Since 2014 a competition is held annually by LifeCLEF, which is a lab of Cross Language Evaluation Forum dedicated to biodiversity (ImageCLEF, 2016). In 2016 the competition took place under the name of BirdCLEF.

The research on bird vocalization classification is developing fast, as are the results. Im-

provements in this field are very rapid, so only the newest research is described in this section. According to author's knowledge and findings, most and best of the research is also presented in these competitions. Therefore, only the winning works of each contest are described in this section.

## MLSP 2013 Bird Classification Challenge

The goal of the MLSP 2013 competition was to classify multiple simultaneous bird species present in noisy field recordings. The full dataset consisted of 645 audio recording in WAV format, each being 10-seconds long. Each recording was paired with the names of the species heard from the recording. Organizers provided additional pre-computed data (spectogram, filtered spectogram, segmentation, segment features and a histogram-of-segments for each species). (Fodor, 2013)

(Fodor, 2013) was the winner of the competition. Fodor's approach was to first create a spectrogram for each recording and then use unsupervised segmentation methods based on image processing techniques to filter undesirable noise and separate different sounds. In this process Gaussian Filtering, local gradient, thresholding and morphological noise removal were used. The remaining holes were then filled and small segments removed. The image processing is illustrated in figure 3.1. Fodor used single-labeled training recordings for gathering patterns with previously described methods. For every spectrogram a template matching function was then used with those extracted patterns to compute the similarity, which generated 5978 dimensional feature vector for each recording. In addition the 100 dimensional histogram-of-segments and location code, provided by the organizers of the competition, were appended to the feature vector.

Fodor used aggressive feature selection for each species due to the small training size and large feature vector dimensions. The feature selection was based on uni-variate linear regression tests. The number of selected features was set manually where higher dimensions were used if the class was more balanced. There was a binary model using Random Forest Regressor (see subsection 2.4.3) for each species.

Fodor implemented his solution in Python using scikit-learn and scikit-image libraries.

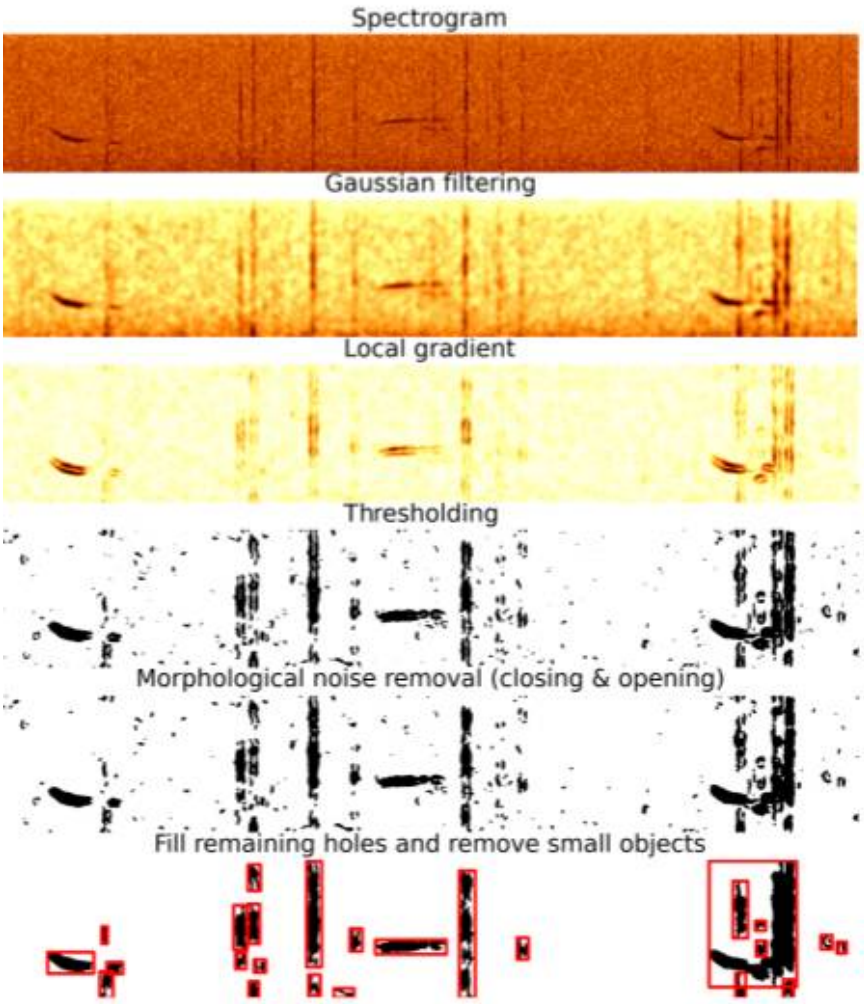



*Figure 3.1.* Image processing used in MSLP 2013 winner's work on a Hermit Thrush spectogram (Fodor, 2013)

## NIPS4B 2013 Multi-label Bird Species Classification

The NIPS4B 2013 competition challenge was to identify 87 sound classes of birds and other animals present in 1000 field audio recordings. The dataset consisted of 2 hours of recording split into smaller recordings ranging from 0.25 to 5.74 seconds. The recordings were saved in an uncompressed WAV format with a sample rate of 44.1 kHz. The 87 individual sound classes represented different bird species, while sounds from other animal species where also audible in the recordings.

(Lasseck, 2013), the winner of the NIPS4B competition, took the work of (Fodor, 2013) as a starting point. In Lasseck's approach, audio was resampled to 22050 Hz and spectrogram created for each recording. Median clipping was used to remove background noise, resulting in a binary image. The image was then further processed using standard image

processing techniques like closing, dilation and median filtering. All connected pixels exceeding a certain area were labeled as a segment and a rectangle with small added padding was used to define it's size and position. One example of these segments on a spectrogram is illustrated on figure 3.2.

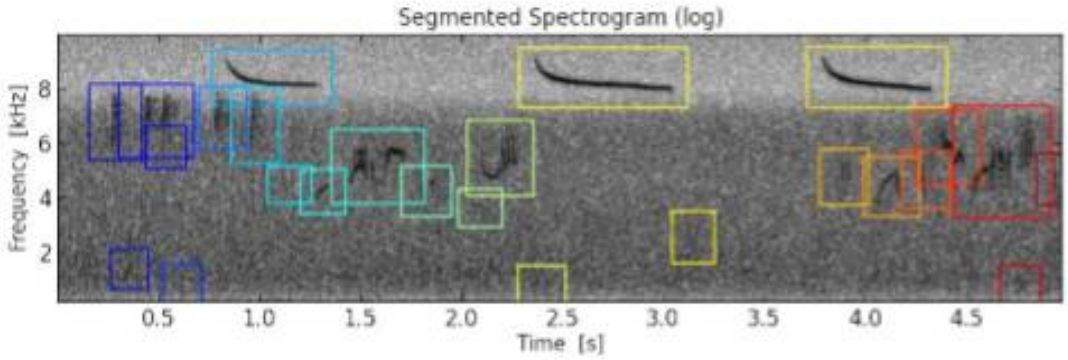



*Figure 3.2.* Segments marked on spectrogram in NIPS4B 2013 winner's work(Lasseck, 2013)

(Lasseck, 2013) used features from three different sources: File-Statistics, Segment-Statistics and Segment-Probabilities. File-Statistics used minimum, maximum, mean and standard deviation taken from all values of the unprocessed spectrogram. In addition, the spectrogram was divided into 16 equally sized frequency bands and their minima, maxima, means and standard deviations were also used. Segment-Statistics included number of segments per file plus minimum, maximum, mean and standard deviation for width, height and frequency position of all segments per audio recording. Segment-Probabilities used absolute-intensity template matching. The multi-label classification problem was turned into 87 individual classification problems. An ensemble of randomized decision trees is applied for each sound class.

The solution was implemented in Python using scikit-learn and OpenCV libraries.

## LifeCLEF 2014 Bird Task

LifeCLEF 2014 Bird Task was a competition where participants need to automatically identify birds from 501 species. The dataset consisted of 14027 recordings from (The Xeno-canto Foundation, 2016) which were saved as 16 bit WAV files. Audio recordings were associated with various meta-data, produced by Xeno-canto community. (Goëau et al.,

2014)

The winner of LifeCLEF 2014 Bird Task was (Lasseck, 2014) who also won NIPS4B competition described above. Lasseck used similar approach, but instead of File-Statistics and Segment-Statistics, he used meta-data provided by the organizers of the competition and extracted features with openSMILE (Eyben, Wöllmer, & Schuller, 2010) feature extraction tool. Meta-data included time of the recording, geographical coordinates, locality index and author index. Features from openSmile summed up to 6669 features for each audio recording. The Segment-Probabilities method is same as described above with very slight changes.

Classification problem is divided into 501 independent classification problems, using one classifier for each species. The classification was done by training ensembles of randomized decision trees with probabilistic outputs.

The solution was implemented in Python where scikit-learn library was used for classification.

## LifeCLEF 2015 Bird Task

LifeCLEF 2015 Bird Task was a competition similar to the LifeCLEF 2014 Bird Task described above. LifeCLEF 2015 competition uses larger dataset with 999 species from 33203 recordings. Audio recordings are saved in 16 bit WAV formats and paired with meta-data. (Joly et al., 2015)

The best solution for LifeCLEF 2015 Bird Task was done by (Lasseck, 2015), who also won the last two competitions described above. The approach was very similar to the previous year's solution. Compared to previous year, no metadata was used in the process. Small changes were done in obtaining features with openSmile and also in template matching. Due to the big amounts of data the template matching algorithm was improved through prior downsampling.

Lasseck also used similar classification methods. In addition, for both feature sets (openSmile and Segment-Probabilities) training was performed in two passes to obtain best feature

selection parameters for each species.

Python with scikit-learn library was used to implement the solution.

### BirdCLEF 2016

The BirdCLEF 2016 competition was very similar to LifeCLEF 2015 Bird Task competition. The training dataset remained the same as in LifeCLEF 2015 competition. Compared to previous year's competition some additional testing data was added and several teams used successfully convolutional neural networks in their work. (Goëau, Glotin, Vellinga, Planqué, & Joly, 2016)

The winner team of BirdCLEF 2016 competition was (Sprengel, Martin Jaggi, & Hofmann, 2016) whose approach involved deep learning. In preprocessing phase they created a spectrogram and using image processing techniques to separate clean signal and noisy signal. The result was two different spectrograms for each recording, one containing only clean signal and second noisy parts. The spectrograms were split into two equally sized chunks, where chunks from the signal part were used as training or testing samples for neural network. Noisy chunks were used on clean signal chunks to generate new data for avoiding overfitting. The approach used convolutional neural network with five convolutional and one dense layer. Every convolutional was followed by a max-pooling layer and used rectify as activation function.

## 3.2 Data Gathering

Machine learning algorithms need need a large amount of data in order to learn well. Luckily, there are good crowd-sourced databases available, with large amounts of data. These databases are described in the following two subsections.

### 3.2.1 Xeno-canto

Most of the bird classification competitions described in section 3.1 used Xeno-canto for gathering audio recordings of bird vocalizations. Xeno-canto is a collaborative project where everyone can upload, download or identify bird sound recordings. By the time of

writing, the Xeno-canto database consisted of 325983 recordings, summing up to 4687 hours of audio, while 3289 recordists have shared bird songs from 9648 species. Xeno-canto database stores its record in mp3 format, while the number of channels, bit-rate and sampling rate can vary. (The Xeno-canto Foundation, 2016)

Xeno-canto has API service which is suitable for filtering and downloading audio recordings and meta-data for needed specie. The meta-data contains information about each recording - location, time, foreground species, background species, length of the recording, type of sound (call, song) and quality of the recording. (Jongsma, 2013)

Xeno-canto also contains audio recordings from Estonia. By the time of writing this work, it contained 361 audio recordings from Estonia, from 140 species. As machine learning works better with large amount of data, additional data should be gathered for training. One solution is to use recordings all over the world, but only for species which can be found in Estonia.This data can then be used for training the machine learning algorithm.

### 3.2.2 My Nature Sound

My Nature Sound is a project in which everyone can record sounds from nature with a mobile application (TÜ Loodusmuuseum, 2016). These recordings can be uploaded to the PlutoF database with the same application (TÜ Loodusmuuseum, 2012). The database consist mainly of audio recordings from bird vocalization, but also includes sounds created by other species. The audio recordings recorded with My Nature Sound application are noisy, because of the recording quality on mobile devices and the environment they were recorded in. Often the audio has been recorded in urban environments, which add additional foreground noise.

The My Nature Sound database in PlutoF consist, at the time of writing, 1610 recordings from 164 species. From these 164 species, 130 are birds with 1418 recordings. Compared to the world-wide data available for species found in Estonia from Xeno-Canto database, the amount of recordings from My Nature Sound is small. Also the data might be too noisy for training the machine learning algorithm. The solution of this work is being developed into a mobile applications and therefore the data from My Nature Sound database can be used

as a test set, because it is recorded in similar conditions and with similar distribution as the new mobile application would have. The data gathered from PlutoF database should be split into two data sets - testing and validation data. The validation data would be used to fine-tune hyper-parameters and selecting the final model and testing data would be used for final evaluation of the selected model. Both of these datasets should have similar distribution to the whole dataset gathered from PlutoF, in order to be used in selecting the best model. The audio recordings from My Nature Sound database are being stored in different formats, depending on the mobile device, used for recording the sound. (TÜ Loodusmuuseum, 2015)

### 3.2.3 Species Selection

Making an algorithm which could recognize all the birds in Estonia with good accuracy is currently not a feasible task, due to lack of audio recordings from rare birds. Therefore only a subset of Estonian birds should be included in the machine learning process. Selected bird species should include the species which are most heard in Estonia. My Nature Sound database contains species recorded by the application users and therefore this database is a good option for understanding which species can be heard the most.

The number of species used in the task should be a compromise between having enough species to make the classification task a challenge, but at the same time using few enough species to get good enough results for practical use. Analysis of the Xeno-canto and My Nature Sound databases show that the distribution of the audio recordings per species is different for these data sources. This means that there could be relatively a lot of recordings about one bird species on My Nature Sound database, but in Xeno-canto the number of recordings for that same species is relatively low.

20 species with the highest number of recordings were chosen from My Nature Sound database for this thesis. The recordings from these species make up over 60% of the recordings from My Nature Sound database, while other 90 species correspond to the rest of the 40%.

## 3.3 Classification

As described in section 3.1, (Sprengel et al., 2016) showed that using deep learning algorithms can significantly improve results on previous solutions. The work of (Sprengel et al., 2016) will be used as a basis of this thesis work.

Convolutional neural network used by (Sprengel et al., 2016) uses segments from the spectrogram as an input to the CNN. The approach allows to analyze different parts of the audio recording separately and predict probabilities of bird species being represented in each segment. This can be useful for the cases where several birds can be simultaneously heard in a recording or for implementation where only certain parts of the recording are needed to be analyzed.

The CNN, proposed by Sprengel, is used to classify from 999 different bird species compared to 20 species used in this work. This suggests that the topology of CNN and hyper-parameters used in Sprengel's work would probably not be the optimal ones to use in this work. Therefore a set of experiments are done to find out the optimal hyper-parameters values. These experiments are described in Chapter 5.

The number of possible hyper-parameter combinations grows rapidly with each added hyper-parameter and due to the long training time and limited time resources, it is not feasible to test many solutions. Although there are algorithms for improving the automated hyper-parameters search, suggested by (Bengio, 2012), these are not implemented during this work due to the time limitation, but will be considered in future implementations. By using manual hyper-parameters search, the previous experience and knowledge gained after each performed experiment can be used for finding optimal hyper-parameters with less experiments. Therefore, the decisions made on choosing the parameters for experiments are based on the authors knowledge, experience, intuition and the results of the previous experiments.

Only limited experiments can be done during this work and therefore a limited set of hyper-parameters should be used for fine-tuning the model. The CNN used in this work can probably be a simpler one, compared to the Sprengel's CNN, because of having almost 50 times less species to classify. No batch normalization is being used for initial experiment,

due to the higher number of available training data for each species. The layer sizes are scaled down proportionally by comparing input and output layer dimensions between the Sprengel's CNN and the proposed CNN. Instead of using rectified linear unit (ReLU) as an activation function, the exponential linear unit (ELU) is used which allows faster learning, as described by (Clevert et al., 2015).

The experiments are performed by changing the number of hidden layers, their dimensions, learning rate and momentum of the optimizer. The experiments are done by changing only one hyper-parameter for each training run to see how each value is affecting the accuracy of the model.

The experiments are ran for a fixed number of steps without using early stopping, in order to get better insights on changing the value of each hyper-parameter (Bengio, 2012). The number of steps is chosen in a way that the whole data is used a number of times in the training process. This number, also called epoch, was chosen from a separate experiment, where the mini-batch accuracy and validation data accuracy were compared to find the number of steps where the model started to overfit and leaving a margin, in case the changes in hyper-parameters would postpone overfitting.

## 3.4 Preprocessing

All the downloaded recordings are converted to uncompressed WAV files. The frequency is downsampled to 22050 Hz which allows to analyze frequencies up to 11025 Hz. By downsampling audio recordings, less space is required and faster computations can be achieved. As most of the bird vocalizations are in between 1000 Hz and 8000 Hz, downsampling would not have large effect on the results (The Cornell Lab of Ornithology, 2009). The WAVs are saved with using only one channel to simplify the preprocessing.

All the solutions described in section 3.1 first created the spectrogram from audio recordings, followed by some image processing steps. The spectrogram offers more intuitive understanding of the audio signal, by representing frequency changes in time. (Sprengel et al., 2016), the winner of BirdCLEF 2016 competition, proposed an approach where median values of spectrogram elements were compared, which allowed to separate signal and noise

parts of the spectrogram. Compared to the solution proposed by earlier winners of bird vocalization classifying competitions, this approach required less computational time, which is important if the solution needs to be implemented on devices with less computational power. Sprengel's approach would be used as a basis for segmentation, but simplifications are being made to achieve simpler and faster preprocessing process for the implementation of this work.

(Sprengel et al., 2016) uses morphological dilation and erosion in the process of separating noise and signal from spectrogram, in order to smooth out noise and join sections of signal or noise with small caps between them. The implementation described in this thesis is not using these morphological operations in order to save time and computational resources on preprocessing phase. Therefore, each time frame, where at least one element has value greater than three times the mean of that row and column, would be included into the cleaned spectrogram.

An overlapping sliding window is proposed for dividing the spectrogram into segments. The segment height will be same as segment width and will correspond to number of frequency bins in the spectrogram (height of the spectrogram). The sliding window will extract the first square matrix as a segment from the beginning of the spectrogram and then moves forward certain amount of time frames to extract another one, until it reaches the end. The overlapping allows more accurate predictions, as there are more segments, especially when multiple birds are audible in the recording.

The resizing of segments is suggested to improve the computations speed and reduce loading time of the data. In addition each segment is standardized, which means that data is centered to mean and scaled to unit variance, allowing faster learning on gradient based learning systems (Lecun, Bottou, Bengio, & Haffner, 1998).

Compared to the BirdCLEF 2016 competition, more data is available for each species. The BirdCLEF task had the average of about 25 audio recordings per species, while the dataset used in this work has about 589 recordings per species (see Table 4.1). Therefore, the trained model should be less prone to overfitting. As a result, the data augmentation by generating new samples with the added noise, is not used to keep the development and

preprocessing as simple and fast as possible for the current version of work. With no data augmentation, there is no need for separating the noise parts from the spectrogram and only signal parts will be extracted.

The big amounts of training data and limited random access memory forces to load segments from disk in several iterations, instead of loading it into memory at once. Therefore, training data is spread over several files. A solution is proposed where segments from one species are divided into several files. On training phase one file from each species is loaded into the memory. The training steps are performed until each segment is used in training once, after which another file is loaded from each species. This is repeated until all the segment files are used, after which it will be started over again for predefined number of epochs. After new data is loaded for training, the segments are shuffled into random order to improve convergence, as suggested by (Bengio, 2012).

## 3.5 Evaluation

Different accuracy measures can be calculated in the evaluation phase. It is proposed to measure accuracy and F1 score in this work. The performance can be measured on segments level and on recordings level. When performance is measured on the segments level, the predictions from each segment are compared to its label. On recordings level the means of the predictions, belonging to the recordings, are calculated and therefore the predictions for the whole recordings are compared to the label of the recording.

After each training run, the accuracy measures should be calculated for every saved checkpoint. For users of the final model, the top-3 accuracy would be important measurement, but for choosing the best model it would not be the best option due to the imbalanced classes, as stated by (Japkowicz & Shah, 2011). F1 score takes into account the imbalance of the species and therefore is better metric for choosing the final model. The experiments done by the author showed that although the F1 score on segmentation level and recording level are correlated, it is not always the case that better F1 score on segments level will mean better F1 score. The same goes for top-3 accuracy metric when compared with segment level F1 score. As the validation set is relatively small, making up less than 1% of the training data, the top-3 accuracy or F1 score on recordings level, should not be used for

choosing the model. The small size of validation set means that the segments grouped together into recordings only represent a small amount of available combinations of segments and might not represent the real world connection between the performance on segments and recordings level. It is suggested to use F1 score on segments level, when choosing the final model. Top-3 accuracy can be used, when discussing the accuracy of the final model, but as the testing set is relatively small, this number should be taken with caution.

When experiments are done the decision has to be made, whether the changes improved the results or not, in order to make decision about next experiment. The experiments show that the performance in validation set can vary by several percentage points by going up or down with each measurement, even when trend is improving. The performance metrics are not calculated on each step and as performance is varying, it might happen that by chance, a model which is actually better, will not achieve its highest performance on the steps it is measured, while other model does. It is suggested to look the average of several top measurements to smooth out the change that model which might perform worse in reality, shows better performance by a lucky chance.

## 3.6 Development Tools

There are many deep learning frameworks available. A research was done in late 2015, with modifications in early 2016 by (Tran, 2016) about different deep learning frameworks. Although the ratings can be subjective, the overview can be used for getting an a better understanding of the most popular frameworks. The research is summed up in table 3.1. It can be seen that (TensorFlow, 2015) has good rating in every analyzed aspect. In addition, the author has had experience with creating CNN-s in TensorFlow (Colonna et al., 2016) which would result spending less time on learning to use a new deep learning framework.

(Torch, 2012) had also good ratings, except for model deployment part. The author of this thesis has used Torch to deploy a pre-trained model as a web service. The setup took a lot of time, due to failing dependencies installation, and the deployment needed workarounds which did not follow the good practice. Based on this experience, running the deep learning model with Torch framework on mobile device might be a very complicated task.

**Table 3.1**
Comparison of deep learning frameworks done by (Tran, 2016)

| | Modeling Capability | Interfaces | Model Deploym. | Performance (single GPU) | Archi-tecture | Average Points |
|---|---|---|---|---|---|---|
| **Caffee** | 6 | 6 | 10 | N/A | 6 | 7 |
| **CNTK** | 4 | 5 | 9 | N/A | N/A | 6 |
| **TensorFlow** | 9 | 9 | 9 | 10 | 10 | 9.4 |
| **Theano** | 9 | 8 | 6 | 6 | 6 | 7 |
| **Torch** | 10 | 8 | 6 | 10 | 10 | 8.8 |

TensorFlow offers examples of running trained model on different platforms, such as Android, iOS and Raspberry Pi. It mainly uses Python, but it also has C++ interface, which can be used to make prediction on pre-trained models. One big advantage of TensorFlow is that it is Google's product and has become very popular Open Source project. Therefore it is developing very rapidly. The author of this work had to rewrite this paragraph several times, because several shortcomings of TensorFlow were improved during the thesis writing process. (TensorFlow, 2015)

Most of the solutions from Section 3.1 use Python as development language. Python is good option for solving machine learning tasks. It is easier to use for data processing and machine learning tasks, compared to popular object oriented languages like Java, C#, C++ or C. Also, Python has more mature selection of machine learning and signal processing libraries. Compared to tools made for statistical analysis, like R and Matlab, it is easier to use Python for deployment of the model.

# 4 Implementation

The implementation of the solution developed in this thesis is divided into eight different phases. The first six phase are done to select and create a trained model, which can be later used in classification tasks. These steps are separated in a way, that parameter tuning can be done without the need of running the previous steps. After running each step some data is saved on disk, which would be used in following step. All the data saved on disk, except audio recordings, are saved using Python serialization module - pickle. The last two phases include evaluating the final model and a solution which can make predictions on audio recordings.

The code for this implementation was ran on Ubuntu 14.04 using Tensorflow version r0.12. Intel(R) Core(TM) i7-4702HQ CPU @ 2.20GHz processor was used with 16 GB of Random Access Memory. The settings for segmentation and training phase were chosen by taking the amount of memory into account and therefore segmentation, data preparation and training phase might have to be rewritten, when tried to run on a device with less than 16 GB of memory.

The sections in this chapter correspond to the Python files, found in Appendices, although some of methods might be in the common module (see Appendix 1). In Section 4.1 the implementation of data gathering is explained, whiled code for this can be found in Appendix 2 and 3. The process of converting audio recordings into WAV files is described in Section 4.2 and the code for it is in Appendix 4. Segmentation process is described in Section 4.3 and its code in Appendix 5. Additional preparation of data is further explained in Section 4.4 and the code for it is in Appendix 6. Section 4.5 describes the implementation of CNN and how data is fed to it, while the code can be found in Appendix 7. The experiments

evaluation process is further described in Section 4.6 and the code for it is in Appendix 8. Final model evaluation is briefly explained in Section 4.7 and the code is displayed in Appendix 9. Section 4.8 describes how predictions for audio files are generated, while the code for it is found in Appendix 10. The implementation of the common module, containing important configurations, methods and classes is described in Section 4.9 and the code can be found in Appendix 1.

## 4.1 Downloading Data

The data is downloaded from two different databases, therefore the implementation is also divided into two modules. Subsection 4.1.1 briefly describes the implementation of downloading data for training from Xeno-canto database. Testing and validation data are obtained from My Nature Sound application recordings which are stored in PlutoF database. The process of downloading recordings and meta-data from PlutoF database is described in Subsection 4.1.2

### 4.1.1 Training Data

The meta-data and audio recordings for training data are downloaded from Xeno-Canto by using Xeno-Canto API 2.0 (Jongsma, 2013). A query is made for each species, which returns a list of meta-data about their audio recordings. If there are more than 500 results, they are spread over several pages. If recording meets the quality requirements, marked as "A" or "B", corresponding to the best quality recordings, the meta-data for this recording is saved. All the chosen recordings are then downloaded, unless they do not already exist in the specified path. All the recordings in Xeno-canto are MP3 files.

### 4.1.2 Testing Data

The meta-data and audio recordings for testing data are downloaded from PlutoF database, using both their public and internal API. The internal API is made publicly available, although there could be changes made without prior notice. The internal API is used due to the limited functionality of the public API and it allows the gathering of data with less steps and resources. Different taxonomy groups are presented as taxon objects in PlutoF. The tax-

ons corresponding to the bird species discussed in this work, are needed to be handpicked due to the mismatch of genus and species names in Xeno-canto and PlutoF.

All the observations made by My Nature Sound application are acquired from PlutoF database, where results span over several pages. Observations where the taxon identifier is found in a list of identifiers corresponding to the target species, are selected. From these selected recordings, meta-data and audio file locations are extracted. The collected meta-data is saved into a file and every audio recordings is downloaded. PlutoF recordings come in various formats, including MP4, M4A, WAV, MP3 and OGG file formats.

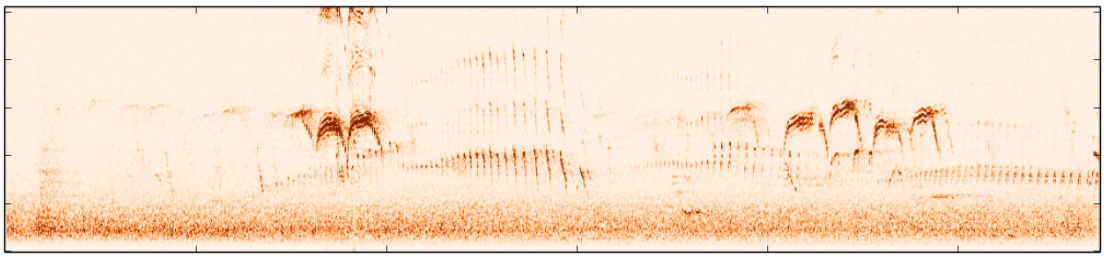
## 4.2 Conversion of Audio Recordings

Downloaded audio recordings are being converted into WAV files by using Pydub library (Robert, 2016). Pydub uses ffmpeg or libav for coding or decoding audio files, therefore necessary codecs have to be installed in the system. All the audio files from a directory are converted into mono channel WAV files, with a sample rate of 22050 samples per second, which allows to calculate frequencies up to 11025 Hz. All audio files are in 16 bit pulse-code modulated (PCM), which means that the amplitude of the analog audio signal is sampled at uniform time, with a resolution of 16 bit per sample.

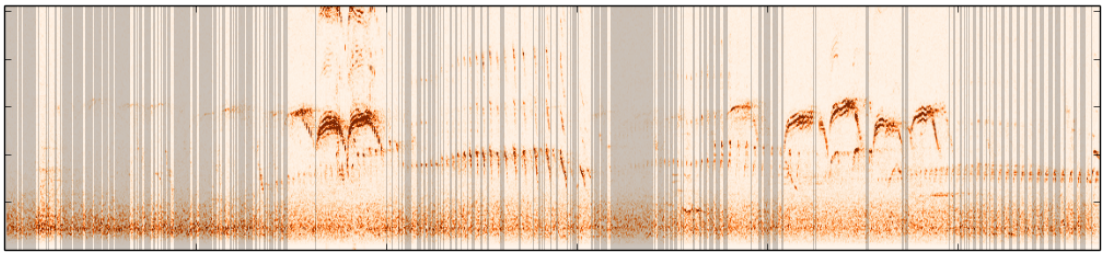
## 4.3 Segmentation

Segmentation phase consists of generating spectrogram for each downloaded audio recording and extracting equal sized segments from that spectrogram. For each audio recording, a file is saved containing all segments from that recording. The segmentation steps are visualized in Figure 4.1.

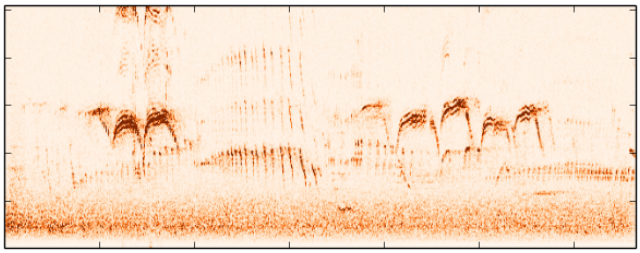
Spectrogram is created by taking absolute value of short-time Fourier transform (STFT) performed on an audio recording. The signal is padded with zeros for not losing any information. Using tools from Numpy library (van der Walt, Colbert, & Varoquaux, 2011), frames are created where each frame contains 512 audio samples and overlaps the previous frame by half. Numpy library is also used for creating the window function, which is applied on each frame, and for taking Fourier transform from these frames. This results



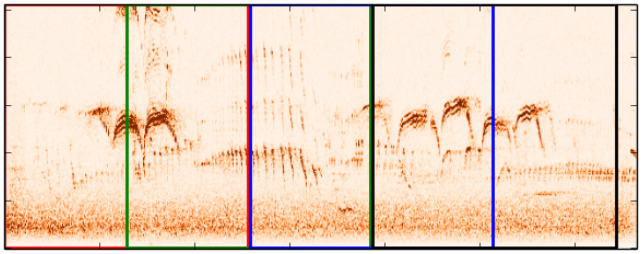
*(a)* Original spectrogram



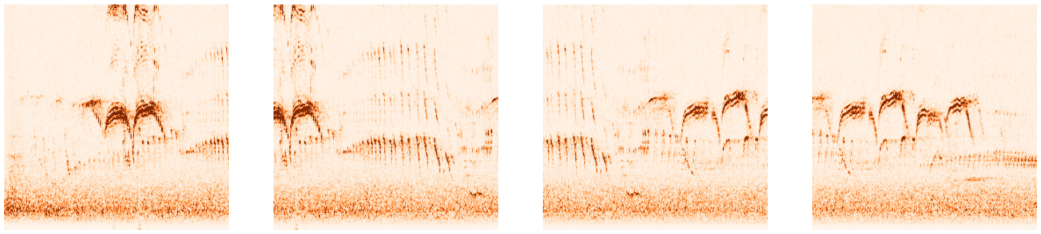
*(b)* Spectrogram where time frames labeled as silent are colored gray



*(c)* Cleaned spectrogram after removal of silent time frames



*(d)* Splitting cleaned spectrogram into segments using sliding window



*(e)* Segments extracted from cleaned spectrogram

*Figure 4.1.* Segmentation process applied to the spectrogram of *Apus apus* recording

in a STFT, which is a complex valued matrix. The frequency axis height of the STFT is 512, but only 256 elements from each time frame are useful, because STFT has symmetric results on its frequency axis. Finally spectrogram is achieved by taking absolute value from each 256 frequency bins of each STFT column. The number of time frames depends on the length of the audio recording.

Spectrogram is cleaned as a next step, leaving out the silent parts of the signal. For this the mean of each row and column is calculated. If some element value of the column exceeds three times the mean value of its column and row, then this column is appended to the clean signal matrix and the algorithm starts iterating over the elements from the next column. The spectrogram showing silent time frames can be seen in Figure 4.1b, while the result of removing silent time frames can be seen in Figure 4.1c.

Cleaned spectrogram has to be divided into equally sized segments, so it can be used by the convolutional neural network algorithm. The dimensions of each segment are chosen in a way, that it would form a square matrix. The order of these square matrices is equal to the number of frequency bins, which is 256 for this implementation. One such segment contains information from about 3 seconds of audio signal. The segments extracted from the cleaned spectrogram are overlapped by half of the width of each segment.

The segment extraction is performed on every downloaded recording and each segment is resized from 256x256 matrix to 64x64 matrix. The segments from each recording are saved as a separate file. The segmentation process took about 8 hours. The number of segments generated for each species can be seen in Table 4.1.

## 4.4 Data Preparation and Balancing

Before starting the training process, extracted training segments are balanced, so each species is having same amount of segments. Each segment is also standardized by centering the data to mean and scaling it to unit variance. The segments from each audio recording are joined together to form files containing training, validation and testing data.

The implementation uses oversampling for balancing data. The highest number of segments

**Table 4.1**
Datasets after segmentation with files and segments count for each species

| Species | Training set | | Validation set | | Testing set | |
|---|---|---|---|---|---|---|
| | **Files** | **Segments** | **Files** | **Segments** | **Files** | **Segments** |
| Fringilla coelebs | 1229 | 23731 | 37 | 230 | 36 | 229 |
| Coloeus monedula | 128 | 1659 | 13 | 43 | 10 | 42 |
| Turdus merula | 1039 | 41633 | 15 | 147 | 16 | 149 |
| Ficedula hypoleuca | 247 | 8650 | 8 | 66 | 7 | 67 |
| Cyanistes caeruleus | 635 | 13536 | 13 | 66 | 14 | 68 |
| Sylvia atricapilla | 888 | 33155 | 8 | 53 | 7 | 53 |
| Erithacus rubecula | 845 | 25211 | 22 | 207 | 21 | 206 |
| Phylloscopus trochilus | 565 | 15590 | 12 | 101 | 13 | 103 |
| Passer domesticus | 384 | 11003 | 12 | 52 | 12 | 53 |
| Sylvia communis | 601 | 15497 | 7 | 35 | 7 | 35 |
| Phylloscopus collybita | 884 | 15306 | 17 | 133 | 17 | 134 |
| Parus major | 1535 | 24205 | 68 | 478 | 68 | 478 |
| Turdus philomelos | 786 | 53562 | 12 | 174 | 13 | 176 |
| Carpodacus erythrinus | 219 | 3714 | 7 | 64 | 6 | 66 |
| Sylvia borin | 392 | 15652 | 12 | 114 | 13 | 116 |
| Corvus cornix | 109 | 1081 | 6 | 19 | 6 | 19 |
| Luscinia luscinia | 264 | 9615 | 10 | 117 | 10 | 119 |
| Apus apus | 94 | 2527 | 6 | 20 | 4 | 19 |
| Emberiza citrinella | 534 | 14969 | 15 | 137 | 14 | 136 |
| Chloris chloris | 400 | 7759 | 12 | 81 | 11 | 80 |
| Total | 11778 | 338055 | 312 | 2337 | 286 | 2348 |

per species is found and taken as the target segments count for each species. As seen in Table 4.1, the highest number of segments for a species is 53562. To achieve the target count for each species, the segments are duplicated until the needed number is achieved.

Due to the amount of data, which could not fit into memory at once, all the training set segments from one species are split into several files and saved to disk. Each represented species therefore has equal number of segment files in training set. Current implementation allows to have the maximum number of 4096 segments in each file, which is chosen based on the available memory of the computer, where the solution was implemented. 14 segments files are saved for each species.

The segments generated from PlutoF recordings are split into testing and validation set. In order to maintain the original distribution as closely as possible, the segments should be split equally, but at the same time the segments from one recording can either be only in

testing or in validation set. For this the recordings from each species are ordered descendingly by segments count. For each species, the segments for each recording are added either into testing or validation set, depending which has less segments before the addition. This algorithm allows both sets to have almost the same distribution as the original set gathered by My Nature Sound application users. Because the small size of testing and validation sets, both files can be saved as a whole without the need to split them. The labels and record id-s are saved into separate files.

The data preparation and balancing took about an hour to complete.

## 4.5 Training

The training is performed by using a convolutional neural networks implemented in Tensorflow. At first the Tensorflow graph of the CNN is defined which includes the necessary CNN topology, variables, loss function and optimizer for minimizing data loss. The graph definition is saved on disk for later use. Experiments were done to find good parameters for solving the problem. The results from these experiments are further described in Section 5.1.

The best solution from the experiments contains three convolutional layers, where each layer is followed by max-pooling layer. These layers are followed by two fully connected layers and a softmax layer, which is also the output layer. For the first two convolutional layers 5x5 kernel is used, while the last convolutional layer uses 3x3 kernel. First convolutional layer has a stride of two, while the following layers have stride of one. The number of kernels used for each convolutional layer are 32, 64 and 128 respectively. All the max-pooling layers have 2x2 sized kernel with the stride value of 2.

After convolutional and max-pooling layers, the data is flattened by re-sizing it to one dimensional array. This array can then be used as an input for the first fully connected layer. The first fully connected layer consists of 256 neurons, while the second one consists of 128 neurons. The fully connected layers are followed by softmax layer. The structure of the CNN can be seen in Figure 4.2

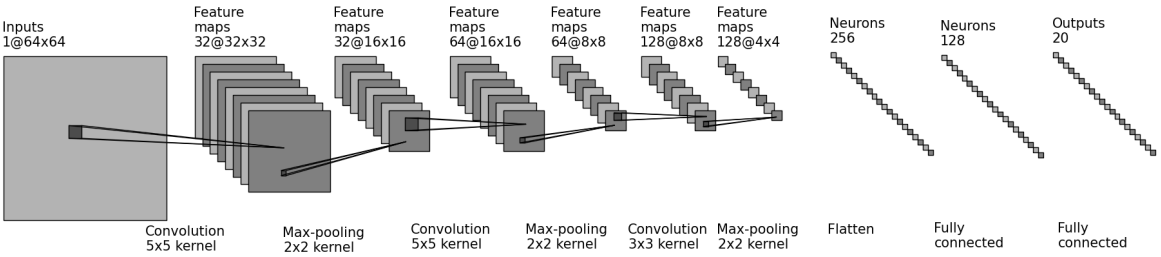



*Figure 4.2.* CNN structure for the selected model

Nesterov accelerated gradient is used as an optimizer for the gradient descent process. The best results were achieved in experiments with the learning rate of 0.0005 and momentum of 0.95.

The training process starts by loading the first training segments file to memory from each species. The 4096 segments from 20 files results in 81920 segments held in memory at once. The data is set into the random order and trained by iterating over the segments, using small batches of segments for training at once. The batch size for the implementation is 128.

After 640 steps, when all the segments which are loaded in memory are used in training once, the next set of training segments is loaded into memory. The CNN is trained for 4 epoch, which means that after using all the segments for training, the whole process is repeated for 3 more times. Current implementation only reads the segment files having 4096 segments, while discarding the last file from each species having less segments.

The loss and accuracy for each batch is displayed after every 50 steps. After 250 steps the weights and bias variables are saved as a checkpoints.

## 4.6 Experiments Evaluation

The accuracy measures are calculated after training for every step, where checkpoint file was created. Therefore, the accuracy is measured after every 250 steps. Before the predictions can be made, the variable values from checkpoint file has to be merged with the Tensorflow graph definition file. This process is called freezing the graph, and it allows to create a file which can be later easily used for calculating predictions. For each checkpoint

generated in training phase, the graphs are frozen and saved to disk by using a tool provided by Tensorflow. These files are called frozen graphs or trained models.

The frozen graph is loaded into memory, and predictions are calculated using Tensorflow by iterating over the validation data. For getting faster performance, the predictions are performed on batches, each containing 32 segments. The prediction for each segment is given as a vector, where each element represents a certain species. The value of each element shows the probability of the segment being from the corresponding species. The sum of vector elements is therefore one.

For segment level accuracy measures, the accuracy, F1 score and confusion matrix are being calculated with the help of sklearn library (Pedregosa et al., 2011), using all the segments. The mean of all the segment predictions for each recording is taken which gives the predictions on recording level. The accuracy, F1 score and confusion matrix are being also calculated for recording level, using the same approach as on segments level. In addition Top-3 accuracy is being calculated for recordings for each saved checkpoint.

The accuracy measures are saved to disk together with the step number from training phase for each checkpoint. This allows the possibility for later analysis of the training process and performance, without the need to run the evaluation phase again.

After calculating accuracy measures for each saved checkpoint, the maximum of each accuracy measure is found and displayed together with the corresponding step number. The mean of ten highest F1 scores on segments and recordings level are calculated as is the mean of ten highest top-3 accuracies.

## 4.7 Final Evaluation

The final trained model is selected from the models created during experiment, which had the best F1 score on segments level. As each experiment has several trained models, one for each checkpoint, the selection is made by choosing the best F1 score on segments level.

The selected trained model is loaded and evaluated on testing data, which the algorithm has not used yet in training nor experiments phase. The F1 score and accuracy on segments and recordings level and top-3 accuracy are being calculated in the same way as in experiments phase, and are being displayed. In addition precision, recall and F1 score are also calculated and displayed for each species using confusion matrix.

## 4.8 Classifying Audio Recording Using Final Model

After the final trained model is selected it is possible to make predictions from WAV files. The WAV file given as an input, is being segmented using approach described in Section 4.3. The trained model is loaded into memory and predictions are calculated for each segment using TensorFlow. For each segment the predictions with the highest probability is displayed, together with this probability. The average of these predictions are calculated and the species are sorted by probabilities in the descending order and displayed together with the probabilities.

## 4.9 Common Module

Some object classes, methods and configuration parameters are shared between different Python modules. Therefore a module named *siuts* was created in order to make the code cleaner and more manageable. The module contains configuration setting shared between different modules, such as the species used in this work, segmentation parameters and paths to files and directories used by modules for saving and loading data. It also contains functions for segmentation and reading WAV files. The meta-data about each recordings is saved in a object class, defined in *siuts* module. In addition a class is defined which can hold information about model evaluation, with several performance metrics.

# 5 Results

This chapter provides analysis for the evaluation results of different convolutional neural network structures tested in this work. The results discussed in this chapter should be interpreted with caution, because only small number of audio recordings were available for validation and testing set, making up less than 1% of training data used in the training phase. The analysis of the experiment results is described in Section 5.1. The analysis of the evaluation results for the final selected model are discussed in Section 5.2.

## 5.1 Analysis of Experiments

Several experiments were done by changing the hyper-parameters of convolutional neural network, in order to find best hyper-parameter set. The experiments were done on the validation set. Thirty experiments were performed in total, where the F1 score on segments level and network training time were found for each hyper-parameters setup. The F1 score was found by taking the average of top 10 F1 scores on segments level from the models saved during training phase. The experiments were done by trying out different number of convolutional layers and their kernels count, number of fully connected layers and the number of neurons in them, and the learning rate and momentum of the Nesterov accelerate gradient descent. These experiments are summed up in Table 5.1.

The first set of tested hyper-parameters were chosen similar to the solution which was used by (Sprengel et al., 2016). As the final training set used in this work consisted of segments with size of 64x64, instead of 256x256, only four convolutional layers were used instead of five. First 13 experiments failed to converge, meaning that the algorithm was not able to learn from data. The possible reason might have been learning rate which was too large

**Table 5.1**

The results of the experiments performed by changing the parameters of the CNN

| Ex. Nr | Segments F1 Score[1] | Training time (min) | Convolutional Layers[2] | FC Layers[3] | Learning Rate | Momen-tum |
|---|---|---|---|---|---|---|
| 1 | Did not converge | | 32; 96; 128; 256 | 128 | 0.1 | 0.9 |
| 2 | Did not converge | | 32; 96; 128; 256 | 128 | 0.01 | 0.9 |
| 3 | Did not converge | | 32; 96; 128; 256 | 256; 128 | 0.01 | 0.9 |
| 4 | Did not converge | | 32; 96; 128; 256 | 256; 128 | 0.1 | 0.9 |
| 5 | Did not converge | | 32; 96; 128;192 | 256; 128 | 0.01 | 0.9 |
| 6 | Did not converge | | 32; 96; 128; 512 | 256; 128 | 0.01 | 0.9 |
| 7 | Did not converge | | 32; 96; 128; 256; 256 | 256; 128 | 0.01 | 0.9 |
| 8 | Did not converge | | 32; 96; 128; 192; 256 | 256; 128 | 0.01 | 0.9 |
| 9 | Did not converge | | 32; 96; 128; 160; 192 | 256; 128 | 0.01 | 0.9 |
| 10 | Did not converge | | 32; 96; 128; 160 | 256; 128 | 0.01 | 0.9 |
| 11 | Did not converge | | 32; 96; 128; 192 | 256; 128 | 0.01 | 0.9 |
| 12 | Did not converge | | 32; 96; 128; 512 | 256; 128 | 0.01 | 0.9 |
| 13 | Did not converge | | 32; 96; 160; 256 | 256; 128 | 0.01 | 0.9 |
| 14 | 0.4620 | 345 | 32; 96; 128; 256 | 256; 128 | 0.005 | 0.9 |
| 15 | 0.4593 | 355 | 32; 96; 128; 256 | 256; 128 | 0.001 | 0.9 |
| 16 | 0.4521 | 410 | 32; 96; 128; 256 | 256; 128 | 0.001 | 0.8 |
| 17 | 0.4746 | 377 | 32; 96; 128; 256 | 256; 128 | 0.001 | 0.95 |
| 18 | Did not converge | | 32; 96; 128; 256 | 256; 128 | 0.001 | 1.0 |
| 19 | 0.4650 | 281 | 32; 64; 128; 256 | 256; 128 | 0.001 | 0.95 |
| 20 | 0.4527 | 512 | 32; 96; 128; 192; 256 | 256; 128 | 0.001 | 0.95 |
| 21 | 0.4692 | 397 | 32; 96; 256 | 256; 128 | 0.001 | 0.95 |
| 22 | 0.4751 | 366 | 32; 96; 192 | 256; 128 | 0.001 | 0.95 |
| 23 | Did not converge | | 32; 96; 160 | 256; 128 | 0.001 | 0.95 |
| 24 | 0.4726 | 280 | 32; 64; 160 | 256; 128 | 0.001 | 0.95 |
| 25 | 0.4761 | 278 | 32; 64; 128 | 256; 128 | 0.001 | 0.95 |
| 26 | 0.465 | 480 | 32; 64; 128 | 512; 128 | 0.001 | 0.95 |
| 27 | 0.4736 | 275 | 32; 64; 128 | 256; 64 | 0.001 | 0.95 |
| 28 | 0.4669 | 269 | 32; 64; 128 | 128; 64 | 0.001 | 0.95 |
| 29 | 0.4668 | 249 | 32; 64; 128 | 256; 128 | 0.005 | 0.95 |
| 30 | **0.4829** | 259 | 32; 64; 128 | 256; 128 | 0.0005 | 0.95 |

[1] Average F1 score from top 10 scores from different training steps
[2] Each number presents the number of kernels used in the corresponding convolutional layer
[3] Each number present the number of neurons in corresponding fully connected layers

and made the algorithm to oscillate around the local minimum.

The algorithm started converging on experiment number 14, where smaller learning rate was tested out. Different learning rates and momentum values were tested out through experiments 14 to 18. This showed that out of the tested hyper-parameters, the learning rate of 0.001 and momentum of 0.95 showed the best results. Algorithm failed to converge when momentum coefficient was one, due to the fact that network was completely insensitive to

the local gradient, while performing gradient descent steps.

Different convolutional layer setups were tested by performing experiments 19 to 25. The experiments showed that network was able to learn better if smaller networks were used. The best result was obtained from experiment 25, where three convolution layers were used, in which the number of kernels was being increased by a factor of 2 with each layer. Using five convolutional layers, as in experiment 19, showed to reduce accuracy and increase training time.

Experiments 26, 27 and 28 were performed by changing the number of neurons in fully connected layers. The accuracy was not improved by these experiments and therefore the number of neurons in fully connected layers was left to 256 in the first fully connected layer and 128 in the second one.

The last two experiments were performed by again trying out different learning rates. Decreasing learning rate to 0.0005, showed the best performance.

The experiments showed that although changing the hyper-parameters of CNN training can change the results by some percentage points, the change is not significant. Relying on author's experience, changes in preprocessing process can have more significant influence on the final results. Compared to Sprengel's solution, a simpler CNN showed better results, as there were almost 50 times less species involved.

## 5.2 Analysis of the Selected Model

The final model was selected from the results of experiments and as experiment 30 showed best results, this model was chosen. The weights and biases were chosen from a checkpoint withe the best F1 score, which was calculated on the segments, not taking into account from which audio recording they were from. The final model was evaluated on testing set, which was not used in training or during the experiments.

The F1 score for the selected model on segments level was 0.444, while on evaluation step the same model showed F1 score of 0.4829. The performance on testing data was not as

good as on validation data. This can be due to overfitting to the evaluation set, which often happens on fine tuning hyper-parameters, especially when the validation set is small. The F1 score on recordings level was 0.5529, which is almost 10 percent points less than on validation set (0.6724). Top-3 accuracy of 0.7581 was achieved, meaning that the true label of the species will fall into top-3 predictions about 76% of times.

The performance was further analyzed by calculating precision, recall and F1 score for each species separately. These values can be seen in Table 5.2, where accuracy measures are found on predictions of recordings. It can be seen that depending on the species, the F1 score varies a lot, while lowest score being 0.1928 for *Apus apus* and highest score being 0.7330 for *Erithacus rubecula*. By comparing the results with the available training data for each species (see Table 4.1), it can be seen that the amount of training data is not directly correlated to how accurately the species is predicted by the algorithm. This might be because of the bird vocalization properties. Some species might perform long and complex songs, while others might use short calls instead.

**Table 5.2**
Species specific evaluation of final model on recordings level

| Species Name | Recall | Precision | F1-score |
|---|---|---|---|
| Parus major | 0.6841 | 0.7002 | 0.6921 |
| Coloeus monedula | 0.2381 | 1.0000 | 0.3846 |
| Corvus cornix | 0.3684 | 0.1944 | 0.2545 |
| Fringilla coelebs | 0.6026 | 0.7931 | 0.6849 |
| Erithacus rubecula | 0.6262 | 0.8836 | 0.7330 |
| Phylloscopus collybita | 0.5299 | 0.7245 | 0.6121 |
| Turdus merula | 0.5839 | 0.5337 | 0.5577 |
| Cyanistes caeruleus | 0.5294 | 0.2278 | 0.3186 |
| Emberiza citrinella | 0.3750 | 0.6711 | 0.4811 |
| Chloris chloris | 0.4750 | 0.7755 | 0.5891 |
| Turdus philomelos | 0.3693 | 0.8667 | 0.5179 |
| Phylloscopus trochilus | 0.3689 | 0.8085 | 0.5067 |
| Sylvia borin | 0.2069 | 0.2963 | 0.2437 |
| Apus apus | 0.8421 | 0.1088 | 0.1928 |
| Passer domesticus | 0.4528 | 0.3871 | 0.4174 |
| Luscinia luscinia | 0.8151 | 0.4350 | 0.5673 |
| Sylvia atricapilla | 0.3774 | 0.5882 | 0.4598 |
| Ficedula hypoleuca | 0.4179 | 0.1582 | 0.2295 |
| Sylvia communis | 0.3714 | 0.5000 | 0.4262 |
| Carpodacus erythrinus | 0.3148 | 0.1954 | 0.2411 |

Recall and precision values also vary a lot for some species. For *Coloeus monedula* it can be seen that algorithm was able to only recognize about 24% of the *Coloeus monedula* recordings, but when the algorithm was predicting *Coloeus monedula*, it was correct each time. The opposite happened to *Apus apus*, where about 84% of all *Apus apus* recordings were correctly predicted, but at the same time, the predictions for that species were correct only about 11% of times. This means that algorithm tends to predict *Apus apus* more often than necessary.

The Bird Song Id Automatic UK mobile application has top-3 accuracy of 85% for 135 species, where it was tested with 1000 recordings, which were 30 seconds long (Sunbird Images, 2015). The specifications of the test set used in validation were not available, but according to the manual of the application, the application should be used in quiet environment and as close to the bird as possible, with only a single bird singing. The test set used in this work had less recordings in total and the length of the recordings varied from 1 second to 1 minute, while the median of the lengths was 16 seconds. In addition, some of the recordings in the test set of this work were very noisy, containing many recordings which had the foreground noise from urban environment. Therefore, the evaluation results are probably worse, due to the length and quality of recordings used in this testing set, compared to the testing set of the Bird Song Id Automatic UK application. The accuracy of the proposed solution and the accuracy of the previously mentioned mobile application cannot be directly compared due to the difference in species and testing data, but it can be seen that the accuracy of the proposed solution can be improved.

The model created in this work can be used by everyone on every platform where enough computational power and usage of Python or C++ language is available, without the need to train the CNN proposed in this work. This allows everyone with enough programming knowledge to implement the bird classification as a web service, mobile application, desktop application or even in an embedded systems, if enough computational resources are available.

# 6 Conclusion

The goal of this thesis was to propose and implement a machine learning solution for automatically classifying the most popular bird species in Estonia. For this TensorFlow and several other Python libraries were used. As the first implementation of this solution would be used in a mobile application, the requirements for running the solution on mobile devices were taken into consideration.

The model created in this work will be available to everyone and can be run without the need of doing the whole training process. It can be used in web services, Raspberry Pi, Android or iOS application, or in any device which has enough computing power and can use either Python or C++. The used approach has already shown success on Garage48 Open & Big Data event in 2016, where a prototype Android application was developed, and a second place and best technology award were achieved among 17 other teams. The future release of this model will be implemented in Android and iOS application in 2017, which allows device owners to record and classify bird sounds.

Section 6.1 sums up the work done in this thesis and briefly describes the proposed solution. The ideas and suggestion for future work are described in Section 6.2.

## 6.1 Performed Work

The theory behind different solutions for solving the problem were described in this thesis, as were the importance of classifying bird vocalizations and the scope of the work. One part of the theoretical description concentrated on describing different techniques for processing audio and images, which would be needed in order to preprocess the data. Another part of

the background information was describing a selection of machine learning algorithms, which are relevant to the bird classification task.

Several solutions of the winners of different bird vocalization classifying competitions were described and analyzed in order to find a good solution for the problem of this thesis. Two databases, Xeno-canto and My Nature Sound, were found, described and analyzed. From Xeno-canto, audio recordings were downloaded for training the algorithm. My Nature Sound database offered audio files recorded by mobile devices in Estonia, and therefore the recordings from this database were used for validation and final testing. In addition, My Nature Sound database was used as a basis for choosing 20 bird species, which would be included in classification task.

The winning work of BirdCLEF 2016 competition was taken as a basis for developing the algorithm, while simplifications were proposed after some analysis. Different machine learning frameworks and programming languages were analyzed and it was found that TensorFlow and Python were the most suitable for the task.

The implementation involved downloading data, converting it to WAV formats, segmenting it, preparing the segments, training and evaluating performance. In addition, a program was created which can use the model saved in training phase, and classify birds species from the audio file given as an input. For segmentation, a spectrogram was created out of the raw audio signal. Some parts of the spectrogram were filtered out by median clipping. The sliding window was used to split the cleaned spectrogram into segments. The data was prepeared for training by oversampling species with fewer segments. For training, a convolutional neural network was used with Nesterov accelerated gradient, as an gradient descent optimizer.

Thirty experiments were performed in order to find good hyper-parameters for the training algorithm. Based on these results, the best model was chosen and the performance of that model was evaluated on unseen testing data. The results showed that for audio recorded by mobile devices, the probability that correct species is in top 3 predictions, is about 76%, which shows that the machine learning algorithm was able to learn from the data.

## 6.2 Future Work

The experiments done in this thesis concentrated on finding out how changing hyper-parameters of CNNs affects the performance of the model. In the future, different experiments on segmentation techniques are planned. It includes doing experiments with fast Fourier transform or short-time Fourier transform parameters.

Currently testing data contains a lot of foreground noise, which can be interpreted as bird song in segmentation phase. Another binary classifier can be implemented, which can predict if the segment has bird vocalization in it, or is the signal belonging to some other foreground noise (car signal, footsteps, sound generated from wind or rain).

Morphological operations could help filter out noise. Experiments should be performed to see how much it helps to improve accuracy of the model. Data augmentation can be used by doing shifts in time, adding noise from other recordings and mixing parts of segments together from the same label.

Current approach re-sizes segments in order to get better performance in training phase, but this means that useful data might be lost. Additional experiments should be done in order to see how much re-sizing segments affects the final results. With the current approach, the 256x256 sized segments are re-sized to 64x64 segments. This would increase the size of the dataset 16 times and therefore a new and more efficient data preparation an augmentation technique should also be applied.

If segmentation and preprocessing are made more complex in the future, more efficient methods for preprocessing and data preparation should be used. Parallel processing can be implemented for using all the available computing power which can speed up the process several times, depending on the amount of cores on CPU, hard drive speeds and memory amount. Parallel processing can be also used in the evaluation phase. In addition TensorFlow data loading format could be implemented for having more standardized method of dealing with the data when it does not fit into memory at once. TensorFlow supports the usage of GPU's, which can be used to accelerate training.

# Resümee

## Automaatne linnuhäälte klassifitseerimine konvolutsiooniliste võrkude abil

Antud diplomitöö eesmärgiks oli masinõppe algoritme kasutades leida lahendus ja see kasutusele võtta Eesti populaarseimate linnuliikide automaatseks klassifitseerimiseks lindude helide abil. Lahendus kasutas konvolutsioonilie võrke, mille juures rakendati TensorFlow-d, ning mitut muud Python-i teeki abil. Esimene rakendus antud lahendusele on mobiilirakendus, mis salvestab heli ja ennustab, mis liike võib salvestiselt kuulda olla. Antud diplomitöö tegemisel arvestati mobiilirakendusega seotud piiranguid. Tulemused näitasid, et mobiilidega salvestatud helisalvestiste puhul saavutas algoritm täpsuse umbes 76%. Tingimuseks oli, et salvestisel olev linnuliik langeks kolme esimese ennustuse hulka.

Diplomitöös loodud mudel on kõigile kättesaadav ning igaüks, kellel on piisavalt programmeerimiskogemust, saab seda kasutada veebiteenuses, Androidi või iOS rakenduses või ükskõik millisel platvormil, millel on võimalik kasutada Python-it või C++-i ning millel on piisavalt võimsust ja mälu.

Diplomitöö raames kirjeldati erinevaid tehnikaid, mida on varasemalt kasutatud sarnaste probleemide lahendamisel ning ka neid, mida kasutati selles töös. Samuti käsitleti probleemi tähtsust ning selle töö eesmärki ja ulatust. Diplomitöö teoreetilises osas kirjeldati erinevaid signaalitöötluse tehnikaid ning vajalikku masinõppe teooriat.

Diplomitöös analüüsiti mitmeid linnuliikide klassifitseerimisvõistluste võitjatöid, et leida sobiv lahendus antud probleemi lahendamiseks. Kirjeldati ja analüüsiti Xeno-canto ja Minu Loodusheli andmebaase, mida kasutati treening- ning testimisandmete kogumiseks. Xeno-canto andmebaasi kasutati treeningandmete kogumiseks. Minu Loodusheli andmebaas sisaldas Eestis mobiilidega salvestatud helisalvestisi ning seega kasutati sealsed salvestisi valideerimis- ja testandmetena. Lisaks kasutati Minu Loodusheli andmebaasi ka klassifit-

seerimisprobleemis kasutatava 20 linnuliigi valimiseks. Diplomitöö käigus loodud algoritmi loomisel võeti aluseks 2016. aasta linnuliikude klassifitseerimisvõistluse võitjatöö. Analüüsi käigus selgus, et vajalikud oli sisse viia mõned lihtsustused. Analüüsiti erinevaid tarkvararaamistikke ning programmeerimiskeeli, mille käigus jõuti järeldusele, et TensorFlow teek ning Python-i keel olid kõige sobivaimad antud probleemi lahendamiseks.

Loodud lahendus hõlmas endas andmete allalaadimist, failide konverteerimist, segmenteerimist, nende segmentide ettevalmistamist, masinõppe algoritmi treenimist ning täpsuse hindamist. Segmenteerimiseks loodi helisignaalist spektrogramm, kust mõned osad jäeti välja spektrogrammi mediaanväärtuste hindamise põhjal. Puhastatud spektrogrammist eraldati lühemad segmendid, eraldades järjestikku välja tükke, mis kattusid poole segmendi ulatuses. Eraldatud segmendid valmistati ette treenimiseks, luues duplikaate nende linnuliikide segmentidest, kus alguses leidus vähem segmente. Treenimiseks kasutati konvolutsioonilist võrku koos Nesterov-i kiirendatud gradiendi optimiseerijaga. Treenitavad muutujad salvestati treenimise käigus perioodiliselt, et hiljem oleks võimalik detailselt hinnata igat salvestatud sammu.

Diplomitöö käigus sooritati kokku 30 eksperimenti, et leida head hüperparameetrid töös arendatud masinõppe algoritmi jaoks. Eksperimentide tulemuste põhjal valiti välja parim mudel, mille täpsust hinnati seni nägemata testandmete põhjal.

# References


Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. *CoRR*, *abs/1206.5533*. Retrieved from http://arxiv.org/abs/1206.5533

Böhning-Gaese, K., & Lemoine, N. (2004). Importance of climate change for the ranges, communities and conservation of birds. *Advances in Ecological Research*, *35*, 211–236.

Bracewell, R. (1965). The fourier transform and iis applications. *New York*, *5*.

Briggs, F., Raich, R., Eftaxias, K., Lei, Z., & Huang, Y. (2013). The ninth annual mlsp competition: overview. In *Ieee international workshop on machine learning for signal processing, southampton, united kingdom., sept* (pp. 22–25).

Britz, D. (2015). *Understanding convolutional neural networks for nlp.* Retrieved 2016-11-29, from http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/

Carignan, V., & Villard, M.-A. (2002). Selecting indicator species to monitor ecological integrity: a review. *Environmental monitoring and assessment*, *78*(1), 45–61.

Christie, M., Fazey, I., Cooper, R., Hyde, T., & Kenter, J. O. (2012). An evaluation of monetary and non-monetary techniques for assessing the importance of biodiversity and ecosystem services to people in countries with developing economies. *Ecological economics*, *83*, 67–78.

Clevert, D., Unterthiner, T., & Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). *CoRR*, *abs/1511.07289*. Retrieved from http://arxiv.org/abs/1511.07289

Colonna, J., Peet, T., Ferreira, C. A., Jorge, A. M., Gomes, E. F., & Gama, J. a. (2016). Automatic classification of anuran sounds using convolutional neural networks. In *Proceedings of the ninth international c* conference on computer science & software engineering* (pp. 73–78). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2948992.2949016 doi: 10.1145/2948992.2949016

Cooley, J. W., & Tukey, J. W. (1965). An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, *19*(90), 297–301.

Eyben, F., Wöllmer, M., & Schuller, B. (2010). Opensmile: the munich versatile and fast open-source audio feature extractor. In *Proceedings of the 18th acm international conference on multimedia* (pp. 1459–1462).

Fisher, R., Perkins, S., Walker, A., & Wolfart, E. (1994). Hypermedia image processing reference. *Department of Artificial Intelligence, University of Edinburgh*.

Fodor, G. (2013). The ninth annual mlsp competition: first place. In *2013 ieee international workshop on machine learning for signal processing (mlsp)* (pp. 1–2).

Glosser.ca. (2013). *Artificial neural network with layer coloring.* Retrieved 2016-11-29, from `https://commons.wikimedia.org/wiki/File:Colored_neural_network.svg`

Goëau, H., Glotin, H., Vellinga, W.-P., Planqué, R., & Joly, A. (2016, September). LifeCLEF Bird Identification Task 2016: The arrival of Deep learning. In *Working Notes of CLEF 2016 - Conference and Labs of the Evaluation forum* (pp. 440–449). Evora, Portugal. Retrieved from `https://hal.archives-ouvertes.fr/hal-01373779`

Goëau, H., Glotin, H., Vellinga, W.-P., Planqué, R., Rauber, A., & Joly, A. (2014). Lifeclef bird identification task 2014. In *Clef2014*.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning.* Retrieved from `http://www.deeplearningbook.org` (Book in preparation for MIT Press)

ImageCLEF. (2016). *Imageclef - the clef cross language image retrieval track | imageclef / lifeclef - multimedia retrieval in clef.* Retrieved 2016-11-25, from `http://imageclef.org/`

Japkowicz, N., & Shah, M. (2011). *Evaluating learning algorithms: a classification perspective*. Cambridge University Press.

Joly, A., Goëau, H., Glotin, H., Spampinato, C., Bonnet, P., Vellinga, W.-P., ... others (2015). Lifeclef 2015: multimedia life species identification challenges. In *International conference of the cross-language evaluation forum for european languages* (pp. 462–483).

Jongsma, J. (2013). *Xeno-canto api 2.0.* Retrieved 2016-11-29, from `http://www.xeno-canto.org/article/153`

Lasseck, M. (2013). Bird song classification in field recordings: Winning solution for nips4b 2013 competition, chap. In *Proc. of 'neural information processing scaled for bioacoustics: from neurons to big data-nip4b', joint to nips conf., issn* (pp. 979–10).

Lasseck, M. (2014). Large-scale identification of birds in audio recordings. In *Clef (working notes)* (pp. 643–653).

Lasseck, M. (2015). Improved automatic bird identification through decision tree based feature selection and bagging. In *Working notes of clef 2015 conference.*

Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998, Nov). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278-2324. doi: 10.1109/5.726791

Lee, C. H., Han, C. C., & Chuang, C. C. (2008, Nov). Automatic classification of bird species from their sounds using two-dimensional cepstral coefficients. *IEEE Transactions on Audio, Speech, and Language Processing*, *16*(8), 1541-1550. doi: 10.1109/TASL.2008.2005345

NIPS4B. (2016). *[nips4b] neural information processing scaled for bioacoustics.* Retrieved 2016-11-29, from `http://sabiod.univ-tln.fr/nips4b/challenge1.html`

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... others (2011). Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, *12*(Oct), 2825–2830.

Raspberry Pi. (2016). *Raspberry Pi.* Retrieved 2016-11-29, from `https://www.raspberrypi.org/`

Robert, J. (2016). *Pydub.* Retrieved 2016-12-18, from `http://pydub.com/`

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *CoRR*, *abs/1609.04747*. Retrieved from `http://arxiv.org/abs/1609.04747`

Smith, S. W., et al. (1997). The scientist and engineer's guide to digital signal processing.

Sprengel, E., Martin Jaggi, Y., & Hofmann, T. (2016). Audio based bird species identification using deep learning techniques. *Working notes of CLEF*.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, *15*(1), 1929–1958.

Sunbird Images. (2015). *Sunbird Apps and eBooks: Nature Apps.* Retrieved 2016-12-18, from `http://sunbird.tv/sunbird-apps-ebooks/`

Sutton, R. S. (1986). Two problems with backpropagation and other steepest-descent learning procedures for networks. In *Proc. 8th annual conf. cognitive science society* (pp. 823–831).

TensorFlow. (2015). *Tensorflow.* Retrieved 2017-01-02, from `https://www.tensorflow.org/`

The Cornell Lab of Ornithology. (2009). *Do bird songs have frequencies higher than humans can hear?* Retrieved 2016-12-18, from `https://www.allaboutbirds.org/do-bird-songs-have-frequencies-higher-than-humans-can-hear/`

The Xeno-canto Foundation. (2016). *Xeno-canto sharing bird sounds from around the world.* Retrieved 2016-10-13, from http://www.xeno-canto.org/

Torch. (2012). *Torch - a scientific computing framework for luajit.* Retrieved 2017-01-02, from http://torch.ch/

Tran, K. (2016). *Evaluation of deep learning toolkits.* Retrieved 2016-11-29, from https://github.com/zer0n/deepframeworks

TÜ Loodusmuuseum. (2012). *PlutoF biodiversity platform.* Retrieved 2016-12-18, from https://plutof.ut.ee/

TÜ Loodusmuuseum. (2015). *Helivaatlused.* Retrieved 2016-12-18, from https://plutof.ut.ee/#/citizen-science-projects/loodusheli

TÜ Loodusmuuseum. (2016). *Minu Loodusheli.* Retrieved 2016-12-18, from http://www.loodusmuuseum.ee/app

van der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, *13*(2), 22-30. Retrieved from http://aip.scitation.org/doi/abs/10.1109/MCSE.2011.37 doi: 10.1109/MCSE.2011.37

Warblr LTD. (2015). *Warblr: the birdsong recognition app.* Retrieved 2016-12-18, from https://warblr.net/

Weisstein, E. W. (2002). *Discrete fourier transform.* Wolfram Research, Inc. Retrieved 2016-12-05, from http://mathworld.wolfram.com/DiscreteFourierTransform.html

# Appendices

The code used in this work is displayed in the following appendices. It can be also accessed on GitHub by following this link: `https://github.com/tpeet/siuts-thesis`

## Appendix 1 Code for Common Library

```
1  import os
2  import wave
3  import pylab
4  import numpy as np
5  from numpy.lib import stride_tricks
6  from sklearn.preprocessing import scale
7  import scipy.misc
8  import pickle
9
10 #
11 # General settings for selecting species and pre-processing properties
12 #
13
14 # List of species used in classification task. The index of the list is
       the label for each species
15 species_list = ['Parus_major', 'Coloeus_monedula', 'Corvus_cornix', '
       Fringilla_coelebs',
16                 'Erithacus_rubecula', 'Phylloscopus_collybita', '
                      Turdus_merula', 'Cyanistes_caeruleus',
17                 'Emberiza_citrinella', 'Chloris_chloris', '
                      Turdus_philomelos', 'Phylloscopus_trochilus',
18                 'Sylvia_borin', 'Apus_apus', 'Passer_domesticus', '
                      Luscinia_luscinia', 'Sylvia_atricapilla',
19                 'Ficedula_hypoleuca', 'Sylvia_communis', '
                      Carpodacus_erythrinus']
20
21 # PlutoF species ID-s had to be handpicked, because the names didn't
       always correspond to the ones in Xeno-Canto.
22 # Each ID in this list corresponds to the species in species_list
23 plutoF_taxon_ids = [86560, 48932, 110936, 60814, 57887, 89499, 107910,
       86555, 56209, 43289, 107914, 89514, 102321,
24                     36397, 86608, 72325, 102319, 60307, 102323, 43434]
25
26 # Xeno-canto quality A is the best
27 acceptable_quality = ["A", "B"]
28
```

```
29 # Sample rate of the wave files
30 wav_framerate = 22050
31
32 # Frame size of the Fourier transform
33 fft_frame_size = 512
34
35 # Final segment size
36 resized_segment_size = 64
37
38 # Batch size in valuation phase
39 test_batch_size = 32
40
41 # How many samples of training data in one file. If lower than 16GB of
       RAM, a lower number should be used
42 samples_in_file = 4096
43
44 # Overlap by half of the segment size when performing segmentation
45 segmentation_hop_size = fft_frame_size / 4
46
47 # Directory path containing audio files, segments, etc...
48 data_dir = "data/"
49 # Directory where Xeno-canto audio files are downloaded
50 xeno_dir = data_dir + "xeno_recordings/"
51 # Directory where PlutoF audio file are downloade
52 plutoF_dir = data_dir + "plutof_recordings/"
53 # File path to the meta-data of xeno-canto recordings
54 xeno_metadata_path = data_dir + "xeno_metadata.pickle"
55 # File path to the meta-data of PlutoF recordings
56 plutof_metadata_path = data_dir + "plutof_metadata.pickle"
57 # Directory where xeno-canto converted WAV files are saved
58 xeno_wavs_dir = data_dir + "xeno_wavs/"
59 # Directory where PlutoF converted WAV files are saved
60 plutof_wavs_dir = data_dir + "plutof_wavs/"
61 # Directory where Xeno-canto segments are saved
62 xeno_segments_dir = data_dir + "xeno_segments/"
63 # Directory where PlutoF segments are saved
64 plutof_segments_dir = data_dir + "plutof_segments/"
65
66 # Directory where the input files for training and evaluation are saved
67 dataset_dir = data_dir + "dataset/"
68 # File path to the joined testing segments
69 testing_data_filepath = dataset_dir + "testing_data.pickle"
70 # File path to the joined testing labels
71 testing_labels_filepath = dataset_dir + "testing_labels.pickle"
72 # File path to the joined testing recording id's
73 testing_rec_ids_filepath = dataset_dir + "testing_rec_ids.pickle"
74 # File path to the joined validation segments
75 validation_data_filepath = dataset_dir + "validation_data.pickle"
76 # File path to the joined validation labels
77 validation_labels_filepath = dataset_dir + "validation_labels.pickle"
78 # File path to the joined validation recording id's
79 validation_rec_ids_filepath = dataset_dir + "validation_rec_ids.pickle"
80
81 # Directory where graph and checkpoint files are saved
82 checkpoints_dir = "checkpoints/"
83 # Directory where frozen graphs are located
84 frozen_graphs_dir = checkpoints_dir + "frozen_graphs/"
```

```
85 # File path to the accuracies objects for each step, where checkpoint
       was saved
86 accuracies_filepath = checkpoints_dir + "accuracies.pickle"
87
88
89 class Recording:
90   segments_count = None
91
92   def __init__(self, identifier, gen, sp, label, file_url):
93     self.id = identifier
94     self.gen = gen
95     self.sp = sp
96     self.label = label
97     self.file_url = file_url
98
99   def __repr__(self):
100    return "id: {0}, name: {1}_{2}, label: {3}".format(self.id, self.gen
           , self.sp, self.label)
101
102  def get_name(self):
103    """Return the scientific name - <genus_species>"""
104    return "{0}_{1}".format(self.gen, self.sp)
105
106  def get_filename(self):
107    """Return the filename withoud extension - <genus_species-id>"""
108    return "{0}_{1}-{2}".format(self.gen, self.sp, self.id)
109
110
111 class Accuracy:
112   def __init__(self):
113     pass
114
115   step = None
116   seg_acc = None
117   seg_auc = None
118   seg_f1 = None
119   seg_conf_matrix = None
120   file_acc = None
121   file_auc = None
122   file_f1 = None
123   file_conf_matrix = None
124   top3_acc = None
125
126
127 def create_dir(path):
128   (dirname, _) = os.path.split(path)
129   if not os.path.isdir(dirname):
130     os.makedirs(dirname)
131
132
133 def get_wav_info(wav_file):
134   wav = wave.open(wav_file, 'r')
135   frames = wav.readframes(-1)
136   sound_info = pylab.fromstring(frames, 'Int16')
137   frame_rate = wav.getframerate()
138   wav.close()
139   return sound_info, frame_rate
140
```

```
141
142 def stft(sig, frame_size, overlap_fac=0.5, window=np.hanning):
143   win = window(frame_size)
144   hop_size = int(frame_size - np.floor(overlap_fac * frame_size))
145
146   # zeros at beginning (thus center of 1st window should be for sample
        nr. 0)
147   samples = np.append(np.zeros(int(np.floor(frame_size / 2.0))), sig)
148   # cols for windowing
149   cols = np.ceil((len(samples) - frame_size) / float(hop_size)) + 1
150
151   samples = np.append(samples, np.zeros(frame_size))
152
153   frames = stride_tricks.as_strided(samples, shape=(cols, frame_size),
154                                     strides=(samples.strides[0] *
                                              hop_size, samples.strides[0])).
                                              copy()
155   frames *= win
156
157   return np.fft.fft(frames)
158
159
160 def clean_spectrogram(transposed_spectrogram, coef=3):
161   row_means = transposed_spectrogram.mean(axis=0)
162   col_means = transposed_spectrogram.mean(axis=1)
163
164   cleaned_spectrogram = []
165
166   for col_index, column in enumerate(transposed_spectrogram):
167     for row_index, pixel in enumerate(column):
168       if pixel > coef * row_means[row_index] and pixel > coef *
            col_means[col_index]:
169         cleaned_spectrogram.append(transposed_spectrogram[col_index])
170         break
171   return np.array(cleaned_spectrogram)
172
173
174 def scale_segments(segments):
175   segment_size1 = len(segments[0])
176   segment_size2 = len(segments[0][0])
177   segment_count = len(segments)
178   segments = np.reshape(segments, (segment_count, segment_size1 *
        segment_size2))
179   scaled_segments = scale(segments, axis=1, with_mean=True, with_std=
        True, copy=True)
180   return scaled_segments.reshape(segment_count, segment_size1,
        segment_size2, 1).tolist()
181
182
183 def segment_wav(wav_path):
184   signal, fs = get_wav_info(wav_path)
185   transposed_spectrogram = abs(stft(signal, fft_frame_size))[:, :
        fft_frame_size / 2]
186   cleaned_spectrogram = clean_spectrogram(transposed_spectrogram)
187   segments = []
188   if cleaned_spectrogram.shape[0] > fft_frame_size / 2:
189     for i in range(int(np.floor(cleaned_spectrogram.shape[0] /
          segmentation_hop_size - 1))):
```

```
190         segment = cleaned_spectrogram[i * segmentation_hop_size:i *
                segmentation_hop_size + cleaned_spectrogram.shape[1]]
191         resized_segment = scipy.misc.imresize(segment, (
                resized_segment_size, resized_segment_size), interp='nearest')
192         segments.append(resized_segment)
193     return segments
194
195
196 def load(location):
197     with open(location, 'rb') as opened_file:
198         return pickle.load(opened_file)
199
200
201 def reformat_labels(labels):
202     labels = (np.arange(len(species_list)) == labels[:, None]).astype(np.
            float32)
203     return np.array(labels)
204
205
206 def accuracy(predictions, labels):
207     return float((100.0 * np.sum(np.argmax(predictions, 1) == np.argmax(
            labels, 1)) / len(predictions)))
```

## Appendix 2 Code for Downloading Xeno-canto Data

```
import json
import urllib2
import urllib
import pickle
import os
import siuts
from siuts import create_dir, Recording


def main():
  create_dir(siuts.data_dir)
  species_set = siuts.species_list
  acceptable_quality = siuts.acceptable_quality

  all_recordings = []
  for species_name in species_set:
    species_split = species_name.split("_")
    genus = species_split[0]
    species = species_split[1]
    url = "http://www.xeno-canto.org/api/2/recordings?query={0}%20{1}".
        format(genus, species)
    json_data = json.load(urllib2.urlopen(url))
    recordings = []

    # if data is divided to several pages, then include them all
    page = int(json_data["page"])
    while int(json_data["numPages"]) - page >= 0:
      # creates list of Recordings objects from recordings of each page
      quality_recordings = [Recording(x['id'], x['gen'], x['sp'],
          species_set.index("{0}_{1}".format(x["gen"], x["sp"])), x["file
          "]) for x in json_data["recordings"] if x["q"] in
          acceptable_quality]
      recordings = recordings + quality_recordings
      page += 1
      if int(json_data["numPages"]) - page >= 0:
        json_data = json.load(urllib2.urlopen(url + "&page=" + str(page)
            ))
    all_recordings = all_recordings + recordings

  with open(siuts.xeno_metadata_path, 'wb') as f:
    pickle.dump(all_recordings, f, protocol=-1)
  print "Finished downloading and saving training meta-data"

  path = siuts.xeno_dir
  create_dir(path)
  recordings_count = len(all_recordings)

  i = 0
  for rec in all_recordings:
    file_path = path + rec.get_filename() + ".mp3"
    i += 1
    if i % 100 == 0:
      print "{0}/{1} downloaded".format(i, recordings_count)

    if not os.path.isfile(file_path) or os.stat(file_path).st_size == 0:
      urllib.urlretrieve(rec.file_url, file_path)

```

```
53
54 if __name__ == "__main__":
55     main()
```

## Appendix 3 Code for Downloading My Nature Sound Data

```
import json
import urllib2
import pickle
import urllib
import math
import os
import siuts
from siuts import create_dir, Recording


def main():
  files_url_prefix = "https://files.plutof.ut.ee/"

  create_dir(siuts.plutoF_dir)

  taxon_ids = siuts.plutoF_taxon_ids

  taxon_url_temp = "https://api.plutof.ut.ee/v1/taxonomy/taxonnodes/{0}/"
  taxon_urls = [taxon_url_temp.format(x) for x in taxon_ids]

  recordings = []
  counter = 0
  url = "https://api.plutof.ut.ee/v1/public/taxonoccurrence/observation/observations/?mainform=15&page={}&page_size=100"
  json_data = json.load(urllib2.urlopen(url.format(1)))
  number_of_pages = int(math.ceil(float(json_data["collection"]["count"]) / 100))

  print "Starting to download audio recordings"

  for page in range(1, number_of_pages + 1):
    json_data = json.load(urllib2.urlopen(url.format(page)))
    print
    print "Downloading from page {}".format(page)
    items = json_data['collection']['items']
    for item in items:
      links = item['links']
      taxon_url = [x['href'] for x in links if 'rel' in x and x['rel'] == 'taxon_node'][0]

      # if species is part of our classification task
      if taxon_url in taxon_urls:
        audio_urls = [x['href'] for x in links if 'format' in x and 'audio' in x['format']]
        if len(audio_urls) > 0:
          audio_url = audio_urls[0].replace("/public/", "/")
          audio_data = json.load(urllib2.urlopen(audio_url))
          file_url = files_url_prefix + audio_data["public_url"]

          rec_id = audio_data["id"]
          label = taxon_urls.index(taxon_url)
          sp_name = siuts.species_list[label]
          gen = sp_name.split("_")[0]
          sp = sp_name.split("_")[1]

```

```
52            # use same file format as with xeno-canto recordings: <
                  genus_species-id>
53            fname = "{}-{:06d}".format(sp_name, audio_data["id"])
54            file_path = "{}/{}.m4a".format(siuts.plutoF_dir, fname)
55
56            if not os.path.isfile(file_path) or os.stat(file_path).st_size
                   == 0:
57              urllib.urlretrieve(file_url, file_path)
58
59            recordings.append(Recording(rec_id, gen, sp, label, file_url))
60            counter += 1
61            if counter % 10 == 0:
62              print "Downloaded␣{}␣files".format(counter)
63
64    with open(siuts.plutof_metadata_path, 'wb') as f:
65      pickle.dump(recordings, f, protocol=-1)
66
67    print ""
68    print "Finished␣saving␣meta-data␣and␣downloaded␣{0}␣recordings".format
        (len(recordings))
69
70
71  if __name__ == "__main__":
72    main()
```

## Appendix 4 Code for Converting Audio to WAV files

```
import datetime
import os
import pickle
import time
from pydub import AudioSegment
import siuts
from siuts import create_dir


def main():
  start = time.time()
  create_dir(siuts.xeno_wavs_dir)
  create_dir(siuts.plutof_wavs_dir)

  print "Starting to convert training data to wav files"
  recordings = siuts.load(siuts.xeno_metadata_path)
  recordings_count = len(recordings)

  i = 0
  skipped_files = 0
  converted_files = 0
  for rec in recordings:
    if i % 1000 == 0:
      print "{0}/{1} | {2}".format(i, recordings_count, str(datetime.
          datetime.now()))
    i += 1
    file_path = "{0}{1}.wav".format(siuts.xeno_wavs_dir, rec.
        get_filename())

    if not os.path.isfile(file_path) or os.stat(file_path).st_size == 0:
      try:
        sound = AudioSegment.from_mp3("{0}{1}.mp3".format(siuts.xeno_dir
            , rec.get_filename()))
        sound = sound.set_frame_rate(siuts.wav_framerate).set_channels
            (1)
        sound.export(file_path, format="wav")
        converted_files += 1
      except:
        print "Error on converting {0}".format(rec.get_filename())
    else:
      skipped_files += 1

  end = time.time()
  print "Converting training data from to wav files took {0} seconds. {
      1} recordings out of {2} were converted.".format(end - start, i,
      recordings_count)
  print "{0} files were already converted.".format(skipped_files)

  start = time.time()
  print ""
  print "Starting to convert testing data to wav files"
  recordings = siuts.load(siuts.plutof_metadata_path)
  recordings_count = len(recordings)

  i = 0
  skipped_files = 0
  converted_files = 0
```

```
52    for rec in recordings:
53      if i % 100 == 0:
54        print "{0}/{1} | {2}".format(i, recordings_count, str(datetime.
              datetime.now()))
55
56      file_path = "{0}{1}.wav".format(siuts.plutof_wavs_dir, rec.
            get_filename())
57      i += 1
58      if not os.path.isfile(file_path) or os.stat(file_path).st_size == 0:
59        try:
60          sound = AudioSegment.from_file("{0}{1}.m4a".format(siuts.
                plutoF_dir, rec.get_filename()))
61          sound = sound.set_frame_rate(siuts.wav_framerate).set_channels
                (1)
62          sound.export(file_path, format="wav")
63          converted_files += 1
64        except:
65          print "Error on {0}".format(rec.get_filename())
66      else:
67        skipped_files += 1
68
69    end = time.time()
70    print "Converting testing data from to wav files took {0} seconds. {1}
          recordings out of {2} were converted.".format(end - start,
          converted_files, recordings_count)
71    print "{0} files were already converted.".format(skipped_files)
72
73
74  if __name__ == "__main__":
75    main()
```

## Appendix 5 Code for Segmentation

```
import os
import pickle
import time
import siuts
from siuts import create_dir


def segment_wavs(recordings_file, segments_dir, wavs_dir):
  create_dir(segments_dir)
  recordings = siuts.load(recordings_file)
  recordings_count = len(recordings)
  for counter, rec in enumerate(recordings):
    fname = rec.get_filename()
    pickle_path = segments_dir + fname + ".pickle"
    if not os.path.isfile(pickle_path):
      wav_path = "{0}{1}.wav".format(wavs_dir, fname)
      if os.path.isfile(wav_path):
        segments = siuts.segment_wav(wav_path)
        if len(segments) > 0:
          with open(pickle_path, 'wb') as f:
            pickle.dump(segments, f, protocol=-1)
    if counter % 100 == 0:
      print "{0}/{1} file segmented".format(counter, recordings_count)


def main():
  print "Starting training data segmentation"
  start = time.time()
  segment_wavs(siuts.xeno_metadata_path, siuts.xeno_segments_dir, siuts.
      xeno_wavs_dir)
  print "Training data segmentation took {0} seconds".format(time.time()
      - start)

  print "Starting testing segmentation"
  start = time.time()
  segment_wavs(siuts.plutof_metadata_path, siuts.plutof_segments_dir,
      siuts.plutof_wavs_dir)
  print "Testing data segmentation took {0} seconds".format(time.time()
     - start)


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

## Appendix 6 Code for Balancing and Preparing Data

```
import time
import siuts
from os import listdir
from os.path import isfile, join
import pickle
import numpy as np
import warnings
import sklearn.utils.validation
import random

import operator

warnings.simplefilter('ignore', sklearn.utils.validation.
    DataConversionWarning)


def load_pickled_segments_from_file(filename, label, rec_id):
  segments = siuts.load(filename)
  segments_number = len(segments)

  if segments_number == 0:
    return np.empty([0]), np.empty([0]), np.empty([0])
  labels = [label] * segments_number
  rec_ids = [rec_id] * segments_number
  return segments, labels, rec_ids


def join_segments(selected_recordings, segments_dir, data_filepath,
    labels_filepath, rec_ids_filepath):
  selected_recordings_count = len(selected_recordings)

  all_segments = []
  all_labels = []
  all_rec_ids = []
  segments_count = {}
  file_count = {}

  if not isfile(data_filepath):
    for counter, rec in enumerate(selected_recordings):
      fname = rec.get_filename()
      label = rec.label
      rec_id = rec.id
      rec_segments, labels, rec_ids = load_pickled_segments_from_file(
          segments_dir + fname + ".pickle", label, rec_id)
      if len(rec_segments) > 0 and len(labels) > 0:
        processed_segments = siuts.scale_segments(rec_segments)
        all_segments = all_segments + processed_segments
        all_labels = all_labels + labels
        all_rec_ids = all_rec_ids + rec_ids

        specimen = rec.get_name()
        if specimen in segments_count:
          segments_count[specimen] += len(processed_segments)
          file_count[specimen] += 1
        else:
          segments_count[specimen] = len(processed_segments)
          file_count[specimen] = 1
```

```
55        if counter % 100 == 0:
56          print "{0}/{1}".format(counter, selected_recordings_count)
57
58      with open(data_filepath, 'wb') as f:
59        pickle.dump(np.array(all_segments), f, protocol=-1)
60
61      with open(labels_filepath, 'wb') as f:
62        pickle.dump(np.array(all_labels), f, protocol=-1)
63
64      with open(rec_ids_filepath, 'wb') as f:
65        pickle.dump(np.array(all_rec_ids), f, protocol=-1)
66      print "File count: " + str(file_count)
67      print
68      print "Segments count: " + str(segments_count)
69
70
71  def main():
72    plutof_recordings = siuts.load(siuts.plutof_metadata_path)
73
74    # count segments for each recording in testing data
75    for rec in plutof_recordings:
76      segments_path = siuts.plutof_segments_dir + rec.get_filename() + ".
            pickle"
77      if isfile(segments_path):
78        rec.segments_count = len(siuts.load(segments_path))
79
80    # separate testing and validation dataset
81    valid_recordings = []
82    test_recordings = []
83    segments_count = 0
84    for specimen in siuts.species_list:
85      recordings = sorted([x for x in plutof_recordings if x.get_name() ==
            specimen and x.segments_count >= 2], key=operator.attrgetter('
          segments_count'))
86      recordings.reverse()
87      sp_valid_recordings = []
88      sp_test_recordings = []
89      sp_valid_segments_count = 0
90      sp_test_segments_count = 0
91      for rec in recordings:
92        segments_count += rec.segments_count
93        if sp_valid_segments_count < sp_test_segments_count:
94          sp_valid_recordings.append(rec)
95          sp_valid_segments_count += rec.segments_count
96        else:
97          sp_test_recordings.append(rec)
98          sp_test_segments_count += rec.segments_count
99
100     valid_recordings = valid_recordings + sp_valid_recordings
101     test_recordings = test_recordings + sp_test_recordings
102
103   siuts.create_dir(siuts.dataset_dir)
104
105   training_segments_dir = siuts.xeno_segments_dir
106   testing_segments_dir = siuts.plutof_segments_dir
107
108   start = time.time()
109   print "Starting to join testing segments"
```

```
110   print
111   plutof_filenames = [x.split(".")[0] for x in listdir(
          testing_segments_dir) if isfile(join(testing_segments_dir, x))]
112   selected_testing_recordings = [x for x in test_recordings if x.
          get_filename() in plutof_filenames]
113
114   join_segments(selected_testing_recordings, testing_segments_dir, siuts
          .testing_data_filepath, siuts.testing_labels_filepath, siuts.
          testing_rec_ids_filepath)
115   print
116   print "Joining␣testing␣segments␣took␣{0}␣seconds".format(time.time() -
           start)
117   print
118
119   start = time.time()
120   print
121   print "Starting␣to␣join␣validation␣segments"
122
123   selected_validation_recordings = [x for x in valid_recordings if x.
          get_filename() in plutof_filenames]
124
125   join_segments(selected_validation_recordings, testing_segments_dir,
          siuts.validation_data_filepath, siuts.validation_labels_filepath,
          siuts.validation_rec_ids_filepath)
126   print
127   print "Joining␣validation␣segments␣took␣{0}␣seconds".format(time.time
          () - start)
128   print
129
130   start = time.time()
131   max_segments = 0
132   species_segments_count = {}
133   species_files_count = {}
134
135   print
136   print "Finding␣species␣from␣training␣set␣which␣has␣the␣maximum␣number␣
          of␣segments"
137   train_filenames = [x.split(".")[0] for x in listdir(
          training_segments_dir) if isfile(join(training_segments_dir, x))]
138   species = siuts.species_list
139   training_recordings = siuts.load(siuts.xeno_metadata_path)
140   for specimen in species:
141     specimen_files = [x for x in training_recordings if x.get_name() ==
            specimen and x.get_filename() in train_filenames]
142     species_files_count[specimen] = len(specimen_files)
143     for rec in specimen_files:
144       fname = rec.get_filename()
145       segs = siuts.load(siuts.xeno_segments_dir + fname + ".pickle")
146       if specimen in species_segments_count:
147         species_segments_count[specimen] += len(segs)
148       else:
149         species_segments_count[specimen] = len(segs)
150     if species_segments_count[specimen] > max_segments:
151       max_segments = species_segments_count[specimen]
152   print "Species␣files␣count"
153   print species_files_count
154
155   print "Species␣segments␣count:"
```

```
156   print species_segments_count
157   print
158
159   print "Max segments for species: " + str(max_segments)
160   print
161
162   # join training segments
163   for specimen in species:
164     print ""
165     print "Joining training segments for {}".format(specimen)
166     specimen_files = [x for x in training_recordings if x.get_name() ==
          specimen and x.get_filename() in train_filenames]
167     specimen_files_count = len(specimen_files)
168
169     all_segments = np.empty
170     all_labels = []
171     all_rec_ids = []
172
173     filepath_prefix = "{0}{1}_".format(siuts.dataset_dir, specimen)
174     labels_fname = filepath_prefix + "labels.pickle"
175     rec_ids_fname = filepath_prefix + "rec_ids.pickle"
176     rec_segments, labels, rec_ids = [], [], []
177     if not (isfile(labels_fname) and isfile(rec_ids_fname)):
178       processed_segments = []
179       for counter, rec in enumerate(specimen_files):
180         fname = rec.get_filename()
181         label = rec.label
182         rec_id = rec.id
183         rec_segments, labels, rec_ids = load_pickled_segments_from_file(
184           siuts.xeno_segments_dir + fname + ".pickle", label, rec_id)
185         if len(rec_segments) > 0 and len(labels) > 0:
186           processed_segments = np.array(siuts.scale_segments(
                rec_segments))
187
188           all_labels = all_labels + labels
189           all_rec_ids = all_rec_ids + rec_ids
190           if counter == 0:
191             all_segments = processed_segments
192           else:
193             all_segments = np.vstack((all_segments, processed_segments))
194
195         if counter % 100 == 0:
196           print "{0}/{1}".format(counter, specimen_files_count)
197
198       del rec_segments
199       del processed_segments
200       print "Saving joined files to disk"
201
202       random.shuffle(all_segments)
203       nr_samples = len(all_segments)
204       # duplicating data in minority classes
205       if nr_samples < max_segments:
206         data_to_append = np.copy(all_segments)
207         for j in range(int(np.floor(max_segments / nr_samples)) - 1):
208           all_segments = np.concatenate((all_segments, data_to_append))
209         all_segments = np.concatenate((all_segments, data_to_append[:(
              max_segments - len(all_segments))]))
```

```
210         nr_of_files = int(np.ceil(float(max_segments) / siuts.
                samples_in_file))
211
212         # save segments into splitted files
213         for i in range(nr_of_files):
214           with open("{0}/{1}-training_{2}.pickle".format(siuts.dataset_dir
                  , specimen, i), 'wb') as f:
215             pickle.dump(all_segments[i * siuts.samples_in_file:(i + 1) *
                    siuts.samples_in_file], f, protocol=-1)
216         print specimen + " segments saved"
217
218         with open(labels_fname, 'wb') as f:
219           pickle.dump(np.array(all_labels), f, protocol=-1)
220
221         with open(rec_ids_fname, 'wb') as f:
222           pickle.dump(np.array(all_rec_ids), f, protocol=-1)
223
224     print "Joining training segments took {0} seconds".format(time.time()
          - start)
225
226
227 if __name__ == "__main__":
228     main()
```

## Appendix 7 Code for Training

```
import sys
import time
import numpy as np
import tensorflow as tf
from sklearn.cross_validation import train_test_split
import siuts
import glob

num_epochs = 4
batch_size = 128


def main():
  start = time.time()
  num_channels = 1
  num_labels = len(siuts.species_list)
  image_size = siuts.resized_segment_size
  num_files = len(glob.glob1(siuts.dataset_dir, "{}-training*".format(
      siuts.species_list[0])))

  graph = tf.Graph()
  with graph.as_default():
    tf.set_random_seed(1337)
    tf_train_dataset = tf.placeholder(
      tf.float32, shape=(batch_size, image_size, image_size,
          num_channels), name="train_dataset_placeholder")
    tf_train_labels = tf.placeholder(tf.float32, shape=(batch_size,
        num_labels), name="train_labels_placeholder")
    tf_test_dataset = tf.placeholder(tf.float32, shape=(siuts.
        test_batch_size, image_size, image_size, num_channels), name="
        test_dataset_placeholder")
    tf_one_prediction = tf.placeholder(tf.float32, shape=(1, image_size,
         image_size, num_channels), name="tf_one_prediction")

    def conv2d(name, data, kernel_shape, bias_shape, stride=1):
      with tf.variable_scope(name) as scope:
        weights = tf.get_variable("weights", kernel_shape, initializer=
            tf.random_normal_initializer(0.0, 0.05))
        biases = tf.get_variable("biases", bias_shape, initializer=tf.
            constant_initializer(1.0))

        conv = tf.nn.conv2d(data, weights, [1, stride, stride, 1],
            padding='SAME', name="conv")
        pre_activation = tf.nn.bias_add(conv, biases)
        activation = tf.nn.elu(pre_activation, name="elu")
        return activation

    def fully_connected(name, data, weights_shape, bias_shape, dropout):
      with tf.variable_scope(name) as scope:
        weights = tf.get_variable("weights", weights_shape, initializer=
            tf.random_normal_initializer(0.0, 0.05))
        biases = tf.get_variable("biases", bias_shape, initializer=tf.
            constant_initializer(1.0))
        activation = tf.nn.elu(tf.nn.bias_add(tf.matmul(data, weights),
            biases), name="elu")
        return tf.nn.dropout(activation, dropout, name="dropout")

```

```
46      def model(data, input_dropout, fc_dropout):
47        data = tf.nn.dropout(data, input_dropout)
48        # Conv1
49        print data
50        conv = conv2d("conv1", data, [5, 5, 1, 32], [32], 2)
51        pool = tf.nn.max_pool(conv, ksize=[1, 2, 2, 1], strides=[1, 2, 2,
              1], padding='SAME', name='pool1')
52        print pool
53
54        # Conv2
55        conv = conv2d("conv2", pool, [5, 5, 32, 64], [64])
56        pool = tf.nn.max_pool(conv, ksize=[1, 2, 2, 1], strides=[1, 2, 2,
              1], padding='SAME', name='pool2')
57        print pool
58
59        # Conv3
60        conv = conv2d("conv3", pool, [3, 3, 64, 128], [128])
61        pool = tf.nn.max_pool(conv, ksize=[1, 2, 2, 1], strides=[1, 2, 2,
              1], padding='SAME', name='pool3')
62        print pool
63
64        # Fully connected 1
65        shape = pool.get_shape().as_list()
66        reshaped_layer = tf.reshape(pool, [shape[0], shape[1] * shape[2] *
               shape[3]])
67
68        fc = fully_connected("fc1", reshaped_layer, [shape[1] * shape[2] *
               shape[3], 256], [256], fc_dropout)
69
70        # Fully connected 2
71        fc = fully_connected("fc2", fc, [256, 128], [128], fc_dropout)
72
73        # output layer
74        return fully_connected("output", fc, [128, num_labels], [
              num_labels], 1)
75
76      logits = model(tf_train_dataset, 1, 0.8)
77      loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits
            + 1e-50, tf_train_labels))
78      optimizer = tf.train.MomentumOptimizer(0.0005, 0.95, use_locking=
            False, name='Momentum', use_nesterov=True).minimize(loss)
79
80      tf.get_variable_scope().reuse_variables()
81      train_prediction = tf.nn.softmax(model(tf_train_dataset, 1, 1), name
            ="sm_train")
82      test_prediction = tf.nn.softmax(model(tf_test_dataset, 1, 1), name="
            sm_test")
83      one_prediction = tf.nn.softmax(model(tf_one_prediction, 1, 1), name=
            "sm_one")
84
85    checkpoint_path = siuts.checkpoints_dir + "model.ckpt"
86
87    with tf.Session(graph=graph) as session:
88      writer = tf.summary.FileWriter(siuts.checkpoints_dir, session.graph)
89      tf.global_variables_initializer().run()
90
91      saver = tf.train.Saver(tf.global_variables(), max_to_keep=200)
```

```
92      tf.train.write_graph(session.graph_def, siuts.checkpoints_dir, "
            graph.pb", False)  # proto
93
94      train_dataset = np.empty
95      train_labels = np.empty
96      current_file = 0
97      current_epoch = 1
98      step = 0
99      while True:
100       if (step * batch_size) % (siuts.samples_in_file * num_labels -
              batch_size) == 0:
101         if current_epoch > num_epochs:
102           break
103         del train_dataset
104         del train_labels
105         sys.stdout.write("Loading␣datasets␣nr␣" + str(current_file))
106         sys.stdout.flush()
107         counter = 0
108
109         train_dataset = np.empty
110         train_labels = np.empty
111         for specimen in siuts.species_list:
112           new_data = siuts.load("{0}{1}-training_{2}.pickle".format(
                siuts.dataset_dir, specimen, current_file))
113           new_labels = np.empty(new_data.shape[0])
114           new_labels.fill(siuts.species_list.index(specimen))
115           if counter == 0:
116             train_dataset = new_data
117             train_labels = new_labels
118           else:
119             train_dataset = np.vstack((train_dataset, new_data))
120             train_labels = np.concatenate((train_labels, new_labels))
121           counter += 1
122
123           sys.stdout.write(".")
124           sys.stdout.flush()
125
126         print
127         current_file += 1
128         if current_file >= num_files - 1:
129           current_file = 0
130           current_epoch += 1
131         train_dataset, _, train_labels, _ = train_test_split(
              train_dataset, siuts.reformat_labels(train_labels), test_size
              =0, random_state=1337)
132       offset = (step * batch_size) % (num_labels * siuts.samples_in_file
              - batch_size)
133       sys.stdout.write(".")
134       sys.stdout.flush()
135
136       batch_data = train_dataset[offset:(offset + batch_size), :, :, :]
137       batch_labels = train_labels[offset:(offset + batch_size), :]
138       feed_dict = {tf_train_dataset: batch_data, tf_train_labels:
              batch_labels}
139       _, l, predictions = session.run([optimizer, loss, train_prediction
              ], feed_dict=feed_dict)
140       if step % 25 == 0:
141         batch_acc = siuts.accuracy(predictions, batch_labels)
```

```
142
143         if step % 250 == 0:
144           saver.save(session, checkpoint_path, global_step=step)
145
146         print '%d - Minibatch loss: %f | Minibatch accuracy: %.1f%%' % (
                step, l, batch_acc)
147       step += 1
148
149     saver.save(session, checkpoint_path, global_step=step)
150
151   print "Training took " + str(time.time() - start) + " seconds"
152
153
154 if __name__ == "__main__":
155   main()
```

## Appendix 8 Code for Training Evaluation

```
import operator
import pickle
import numpy as np
import tensorflow as tf
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score
from sklearn.metrics import roc_auc_score
from tensorflow.python.platform import gfile
from tensorflow.python.tools import freeze_graph
import siuts

siuts.create_dir(siuts.frozen_graphs_dir)

# How many results from saved checkpoints are used in calculating the mean accuracy metrics
nr_in_mean = 10

# Location to input graph
input_graph = siuts.checkpoints_dir + "graph.pb"

# TensorFlow saver file to load
input_saver = ""

# Whether the input files are in binary format
input_binary = True

# The name of the output nodes, comma separated
output_node_names = "sm_one,tf_one_prediction,sm_test,test_dataset_placeholder"

# The name of the master restore operator
restore_op_name = "save/restore_all"

# The name of the tensor holding the save path
filename_tensor_name = "save/Const:0"

# Whether to remove device specifications.
clear_devices = True

# comma separated list of initializer nodes to run before freezing
initializer_nodes = ""


def get_accuracies(graph_path, validation_data, validation_labels, recording_ids):
  acc_obj = siuts.Accuracy()
  with tf.Session() as persisted_sess:
    with gfile.FastGFile(graph_path, 'rb') as opened_file:
      graph_def = tf.GraphDef()
      graph_def.ParseFromString(opened_file.read())
      persisted_sess.graph.as_default()
      tf_test_dataset = tf.placeholder(tf.float32, shape=(siuts.test_batch_size, 64, 64, 1))
      test_predictions_op = tf.import_graph_def(graph_def, input_map={"test_dataset_placeholder:0": tf_test_dataset}, return_elements=['sm_test:0'])

```

```
52      testing_predictions = np.empty
53      for i in range(validation_data.shape[0] / siuts.test_batch_size):
54        start = i * siuts.test_batch_size
55        end = (i + 1) * siuts.test_batch_size
56        if i == 0:
57          testing_predictions = test_predictions_op[0].eval(feed_dict={
              tf_test_dataset: validation_data[start:end]})
58        else:
59          testing_predictions = np.concatenate((testing_predictions,
              test_predictions_op[0].eval(feed_dict={tf_test_dataset:
              validation_data[start:end]})))
60
61      validation_labels = validation_labels[:testing_predictions.shape[0]]
62
63    predictions = np.argmax(testing_predictions, 1)
64    labels = np.argmax(validation_labels, 1)
65    recording_ids = recording_ids[:testing_predictions.shape[0]]
66
67    acc_obj.seg_acc = siuts.accuracy(testing_predictions,
        validation_labels)
68    acc_obj.seg_auc = roc_auc_score(validation_labels, testing_predictions
        , average="weighted")
69    acc_obj.seg_f1 = f1_score(labels, predictions, average='weighted')
70    acc_obj.seg_conf_matrix = confusion_matrix(labels, predictions)
71
72    file_predictions = []
73    file_labels = []
74    for rec_id in recording_ids:
75      rec_predictions = []
76      test_label = -1
77      for i in range(len(recording_ids)):
78        if recording_ids[i] == rec_id:
79          rec_predictions.append(np.array(testing_predictions[i]))
80          test_label = validation_labels[i]
81      if len(rec_predictions) > 0:
82        file_predictions.append(np.array(rec_predictions))
83        file_labels.append(test_label)
84
85    file_predictions_mean = []
86    for prediction in file_predictions:
87      prediction = np.array(prediction)
88      file_predictions_mean.append(np.asarray(np.mean(prediction, axis=0))
          )
89
90    total = 0
91    for i in range(len(file_predictions_mean)):
92      if np.argmax(file_predictions_mean[i]) == np.argmax(file_labels[i]):
93        total += 1
94
95    acc_obj.file_acc = float(total) / len(file_predictions_mean)
96    file_predictions_mean = np.array(file_predictions_mean)
97    file_labels = np.array(file_labels)
98
99    rec_predictions = np.array([np.argmax(pred) for pred in
        file_predictions_mean])
100   rec_labels = np.argmax(file_labels, 1)
101
```

```
102   acc_obj.file_auc = roc_auc_score(file_labels, file_predictions_mean,
          average="weighted")
103   acc_obj.file_f1 = f1_score(rec_labels, rec_predictions, average='
          weighted')
104
105   rec_conf_matrix = confusion_matrix(rec_labels, rec_predictions)
106   acc_obj.file_conf_matrix = rec_conf_matrix
107
108   file_predictions_top = []
109   for i in range(len(file_predictions_mean)):
110     top_3 = []
111     pred = np.copy(file_predictions_mean[i])
112     for j in range(3):
113       index = np.argmax(pred)
114       top_3.append(index)
115       pred[index] = -1.0
116     file_predictions_top.append(top_3)
117
118   TPs = 0
119   for i in range(len(file_predictions_mean)):
120     if rec_labels[i] in file_predictions_top[i]:
121       TPs += 1
122   acc_obj.top3_acc = float(TPs) / len(file_predictions_mean)
123   return acc_obj
124
125
126 def main():
127   data = siuts.load(siuts.validation_data_filepath)
128   labels = siuts.reformat_labels(siuts.load(siuts.
          validation_labels_filepath))
129   rec_ids = siuts.load(siuts.validation_rec_ids_filepath)
130   output_path = siuts.frozen_graphs_dir + "frozen_graph-{}.pb"
131   accuracies_list = []
132   for checkpoint in tf.train.get_checkpoint_state(siuts.checkpoints_dir)
          .all_model_checkpoint_paths:
133     step = checkpoint.split("-")[1]
134     print "Evaluating␣for␣step␣{0}".format(step)
135     freeze_graph.freeze_graph(input_graph, input_saver, input_binary,
            checkpoint, output_node_names, restore_op_name,
            filename_tensor_name, output_path.format(step), clear_devices,
            initializer_nodes)
136     accuracies = get_accuracies(output_path.format(step), data, labels,
            rec_ids)
137     accuracies.step = step
138     accuracies_list.append(accuracies)
139
140   with open(siuts.accuracies_filepath, 'wb') as f:
141     pickle.dump(accuracies_list, f, protocol=-1)
142
143   print
144   print "Highest␣segments␣level␣F1␣scores"
145   accuracies_list.sort(key=operator.attrgetter('seg_f1'), reverse=True)
146   for acc in accuracies_list[:nr_in_mean]:
147     print "{:6}␣{:1.4f}␣".format(acc.step, acc.seg_f1)
148   print "Mean␣of␣{0}␣segment␣level␣F1␣scores:␣{1}".format(nr_in_mean, np
          .mean([x.seg_f1 for x in accuracies_list[:nr_in_mean]]))
149
150   print
```

```
151   print "Highest␣recording␣level␣F1␣scores:"
152   accuracies_list.sort(key=operator.attrgetter('file_f1'), reverse=True)
153   for acc in accuracies_list[:nr_in_mean]:
154     print "{:6}␣{:1.4f}␣".format(acc.step, acc.file_f1)
155   print "Mean␣of␣{0}␣file␣level␣F1␣scores:␣{1}".format(nr_in_mean, np.
        mean([x.file_f1 for x in accuracies_list[:nr_in_mean]]))
156
157   print
158   print "Highest␣top␣3␣accuracies"
159   accuracies_list.sort(key=operator.attrgetter('top3_acc'), reverse=True
        )
160   for acc in accuracies_list[:nr_in_mean]:
161     print "{:6}␣{:1.4f}␣".format(acc.step, acc.top3_acc)
162   print "Mean␣of␣{0}␣highest␣top-3␣accuracies:␣{1}".format(nr_in_mean,
        np.mean(
163     [x.top3_acc for x in accuracies_list[:nr_in_mean]]))
164   print
165
166
167 if __name__ == "__main__":
168   main()
```

## Appendix 9 Code for Final Evaluation

```
import numpy as np
import tensorflow as tf
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score
from sklearn.metrics import roc_auc_score
from tensorflow.python.platform import gfile
import siuts
import sys


def main(graph_path):
  num_classes = len(siuts.species_list)
  testing_data = siuts.load(siuts.testing_data_filepath)
  test_labels = siuts.reformat_labels(siuts.load(siuts.
      testing_labels_filepath))
  recording_ids = siuts.load(siuts.testing_rec_ids_filepath)

  print "Getting predictions..."

  with tf.Session() as persisted_sess:
    with gfile.FastGFile(graph_path, 'rb') as opened_file:
      graph_def = tf.GraphDef()
      graph_def.ParseFromString(opened_file.read())
      persisted_sess.graph.as_default()
      tf_test_dataset = tf.placeholder(tf.float32, shape=(siuts.
          test_batch_size, 64, 64, 1))
      test_predictions_op = tf.import_graph_def(graph_def, input_map={"
          test_dataset_placeholder:0": tf_test_dataset}, return_elements
          =['sm_test:0'])

    testing_predictions = np.empty
    for i in range(testing_data.shape[0] / siuts.test_batch_size):
      start = i * siuts.test_batch_size
      end = (i + 1) * siuts.test_batch_size
      if i == 0:
        testing_predictions = test_predictions_op[0].eval(feed_dict={
            tf_test_dataset: testing_data[start:end]})
      else:
        testing_predictions = np.concatenate((testing_predictions,
            test_predictions_op[0].eval(feed_dict={tf_test_dataset:
            testing_data[start:end]})))

    test_labels = test_labels[:testing_predictions.shape[0]]

  predictions = np.argmax(testing_predictions, 1)
  labels = np.argmax(test_labels, 1)
  recording_ids = recording_ids[:testing_predictions.shape[0]]

  print
  print "----Results on segments level----"

  print "Accuracy: " + str(siuts.accuracy(testing_predictions,
      test_labels))
  print "AUC score (weighted): " + str(roc_auc_score(test_labels,
      testing_predictions, average="weighted"))
  print "F1 score (weighted): " + str(f1_score(labels, predictions,
      average='weighted'))
```

```
48
49  seg_conf_matrix = confusion_matrix(labels, predictions)
50  print
51  print "Labels     Species name      Recall  Precis.  F1-score"
52  print "-------------------------------------------------------"
53  for i in range(num_classes):
54    TP = seg_conf_matrix[i][i]
55    SUM = np.sum(seg_conf_matrix[i])
56    if SUM == 0:
57      recall = 0
58      precision = 0
59      f1 = 0
60    else:
61      recall = float(TP) / SUM * 100
62      try:
63        precision = float(TP) / np.sum(seg_conf_matrix[:, i]) * 100
64      except ZeroDivisionError:
65        precision = 0
66
67      try:
68        f1 = 2 * (recall * precision) / (recall + precision)
69      except ZeroDivisionError:
70        f1 = 0
71
72    print "{:2d}  {:^25}  {:05.2f}  |  {:05.2f}  |  {:05.2f}".format(i, siuts.
        species_list[i], recall, precision, f1)
73  print
74
75  print "----Results on recordings level----"
76  file_predictions = []
77  file_labels = []
78  for rec_id in recording_ids:
79    rec_predictions = []
80    test_label = []
81    for i in range(len(recording_ids)):
82      if recording_ids[i] == rec_id:
83        rec_predictions.append(np.array(testing_predictions[i]))
84        test_label = test_labels[i]
85    if len(rec_predictions) > 0:
86      file_predictions.append(np.array(rec_predictions))
87      file_labels.append(test_label)
88
89  file_predictions_mean = []
90  for prediction in file_predictions:
91    prediction = np.array(prediction)
92    file_predictions_mean.append(np.asarray(np.mean(prediction, axis=0))
        )
93
94  total = 0
95  for i in range(len(file_predictions_mean)):
96    if np.argmax(file_predictions_mean[i]) == np.argmax(file_labels[i]):
97      total += 1
98
99  file_acc = float(total) / len(file_predictions_mean)
100 print "Accuracy: " + str(file_acc)
101
102 file_predictions_mean = np.array(file_predictions_mean)
103 file_labels = np.array(file_labels)
```

```
104   rec_predictions = np.array([np.argmax(pred) for pred in
          file_predictions_mean])
105   rec_labels = np.argmax(file_labels, 1)
106
107   print "AUC score (weighted): " + str(roc_auc_score(file_labels,
          file_predictions_mean, average="weighted"))
108   print "F1 score (weighted): " + str(f1_score(rec_labels,
          rec_predictions, average='weighted'))
109
110   rec_conf_matrix = confusion_matrix(rec_labels, rec_predictions)
111   print
112   print "Prediction accuracy on recordings level"
113   print "Labels      Species name        Recall Precis. F1-score"
114   print "-------------------------------------------------------"
115   for i in range(num_classes):
116     TP = rec_conf_matrix[i][i]
117     SUM = np.sum(rec_conf_matrix[i])
118     try:
119       recall = float(TP) / SUM * 100
120     except ZeroDivisionError:
121       recall = 0
122
123     try:
124       precision = float(TP) / np.sum(rec_conf_matrix[:, i]) * 100
125     except ZeroDivisionError:
126       precision = 0
127
128     try:
129       f1 = 2 * (recall * precision) / (recall + precision)
130     except ZeroDivisionError:
131       f1 = 0
132     print "{:2d} {:^25} {:6.2f} | {:6.2f} | {:6.2f}".format(i, siuts.
            species_list[i], recall, precision, f1)
133
134   print
135
136   file_predictions_top = []
137   for i in range(len(file_predictions_mean)):
138     top_3 = []
139     pred = np.copy(file_predictions_mean[i])
140     for j in range(3):
141       index = np.argmax(pred)
142       top_3.append(index)
143       pred[index] = -1.0
144     file_predictions_top.append(top_3)
145
146   TPs = 0
147   for i in range(len(file_predictions_mean)):
148     if rec_labels[i] in file_predictions_top[i]:
149       TPs += 1
150   top3_acc = float(TPs) / len(file_predictions_mean)
151   print "Top-3 accuracy: " + str(top3_acc)
152
153
154 if __name__ == "__main__":
155   if len(sys.argv) > 1:
156     main(sys.argv[1])
157   else:
```

```
158         print "You␣have␣to␣add␣path␣to␣the␣frozen␣graph␣as␣a␣command␣line␣
                argument"
```

## Appendix 10 Code for Classifying With Final Model

```
import tensorflow as tf
from tensorflow.python.platform import gfile
from operator import itemgetter
import numpy as np
import sys
import os
import warnings
import siuts

warnings.filterwarnings('ignore')


def reshape_segments(segments):
  segments = np.array(segments)
  return np.reshape(segments, [segments.shape[0], segments.shape[1],
      segments.shape[2], 1])


def main(wav_path, graph_path="siuts_model.pb"):
  if not os.path.isfile(graph_path):
    print "No model file found. Please specify trained model path as a
        second command line argument"
    return

  segments = reshape_segments(siuts.segment_wav(wav_path))
  with tf.Session() as persisted_sess:
    with gfile.FastGFile(graph_path, 'rb') as opened_file:
      graph_def = tf.GraphDef()
      graph_def.ParseFromString(opened_file.read())
      persisted_sess.graph.as_default()
      tf_sample = tf.placeholder(tf.float32, shape=(1, 64, 64, 1))
      predictions_op = tf.import_graph_def(graph_def, input_map={"
          tf_one_prediction:0": tf_sample}, return_elements=['sm_one:0'])

    predictions = np.empty
    for i in range(len(segments)):
      if i == 0:
        predictions = predictions_op[0].eval(feed_dict={tf_sample: [
            segments[i]]})
      else:
        predictions = np.concatenate((predictions, predictions_op[0].
            eval(feed_dict={tf_sample: [segments[i]]})))

  first_predictions = np.argmax(predictions, 1)
  print "Prediction for each segment:"
  for i, prediction in enumerate(first_predictions):
    print "{:^25} ({:05.2f}%)".format(siuts.species_list[prediction].
        replace("_", " "), max(predictions[i]) * 100)

  averaged_predictions = np.mean(predictions, axis=0)
  predictions_dict = {}
  for i, prediction in enumerate(averaged_predictions):
    predictions_dict[siuts.species_list[i]] = prediction
  print
  print "Predictions:"
  for species, probability in sorted(predictions_dict.items(), key=
      itemgetter(1), reverse=True):
```

```
51      print "{:^25} ({:05.2f}%)".format(species.replace("_", " "),
            probability * 100)
52
53
54 if __name__ == "__main__":
55   if len(sys.argv) > 2:
56     main(sys.argv[1], sys.argv[2])
57   elif len(sys.argv) > 1:
58     main(sys.argv[1])
59   else:
60     print "You have to add path to the *.wav file as a command line
           argument"
```