University Utrecht

Master Thesis

Improving Sound Event Detection Using Neural Network Trees

Project Supervisor:
Dr. Anja Volk
Utrecht University

Daily Supervisor:
Dr. Jerome Schalkwijk
Intrasonics

Author:
Koen Schalkwijk

Associate Daily Supervisor:
Dr. Peter Kelly
Intrasonics

Second Examiner:
Dr. Peter Kranenburg
Utrecht University

2019

Utrecht University

Intrasonics

Abstract

Sound Event Detection (SED), also known as Audio Event Detection, is the task of labelling audio files with what information is represented in the audio, and when. Most current state-of-the-art SED systems use neural networks. To try and improve on these state-of-the-art methods this work introduces a new method called Branching Neural Networks, that uses multiple neural networks grouped into a tree structure to create a SED.

The properties of this branching neural network are researched by testing the branching neural network method on multiple subsets of a dataset created for testing SED systems during the Detection and Classification of Acoustic Scenes and Events (DCASE) competition in 2017, the DCASE 2017 task 4 dataset.

The branching neural network system seems less negatively influenced by using an non-optimal threshold. Where the threshold is the minimum confidence value for which the SED system accepts that any sound event has occurred. In some subsets the branching neural network system has shown to have significantly better results for all threshold values.

The input to this branching neural network system is the input to a standard SED system, where the classes to train on are not presented as a list, but a tree structure instead.

The tree structure, when used by the branching neural network, has the added benefit of providing intermediate results that have much increased accuracy over the end results. The disadvantage of the tree structure is the requirement for extra tuning, as non-optimal trees can significantly decrease performance when compared to the system with an optimal tree.

Contents

Chapter 1

Introduction

1.1 Motivation

Sound Event Detection (SED), also known as Audio Event Detection, is the task of labelling audio files with what information is represented in the audio, and when. SED has many applications including:

- Detecting vehicle sounds for a smart car system to avoid accidents.
- Detecting gun-shots and screaming in audio recordings in a home security system so the police can be alerted.
- Detecting animal noises to assess which animals occur in an area, and when they are most likely to appear.
- Automatic content recognition in audio/audiovisual media.

Current state of the art research in SED systems use neural networks trained on pre-labelled data. This approach has worked well for Object Detection in images, for which neural networks are now considered the de-facto standard (Ren et al. (2015), He et al. (2016), Szegedy et al. (2017)), and has seen some success for SED systems (Cakir et al. (2016), Jeong et al. (2017)). However, for sounds, these neural networks, working with multiple different label classes, have been less successful than they have been for Object Detection in images. This is illustrated by the Detection and Classification of Acoustic Scenes and Events (DCASE) competition, the biggest competition to find the limits of current state of the art techniques for automatically detecting sound elements. In DCASE 2017, task 4 revolved around creating a SED system to detect dangerous vehicle sounds for a smart car system to avoid accidents. The winner of this competition had an accuracy of 55.6% on 17 classes (Rohan Badlani, 2017). Compare that to the results of the winning system for a similar contest for Object Detection in images, ILSVRC 2017, which had an accuracy of 73.2% on 200 classes (Russakovsky et al., 2015).

In this research project, we propose a completely new method that exploits the human grouping ability by using an ontology of the dataset. An ontology is a set of concepts and categories of a subject area or domain that are both joined by their properties and the relations between them. So for our SED dataset, this is how different sound events relate. This ontology is used to create tree shaped groupings of neural networks used in SED systems. For this thesis the current state of the art Convolutional Gated Recurrent Neural Networks, discussed in section 4, will be used, however theoretically any neural network should work. We use this human grouping as humans are quite capable of differentiating both different sounds and different images, as such this might increase the capabilities of SED systems. The advantages of using these groupings could include:

- When adding a class we only need to retrain a subset of the full network.
- Self checking of the output by requiring that multiple networks agree on the results of the group.
- Providing sub classifications that could be useful, for example knowing that a sound event represents a siren is sometimes more important than knowing whether it represents a police siren or an ambulance siren. In this way the sub classifications alone might be sufficient to decrease the probability of an accident.
- Because the internal neural networks, the networks used for each of the branches, can be changed, this method would improve together with improvements of neural networks for SED systems.

1.2 Sound Event Detection

Although SED systems have been made using **support vector machines** (Temko et al. (2009), Foggia et al. (2015), Elizalde et al. (2016)), **Gaussian Mixture Models** (Cai et al. (2006), Mesaros et al. (2010), Heittola, Mesaros, Eronen, & Virtanen (2013), Vuegen et al. (2013)), **Deep Random Forest** (Yu et al., 2017), **Hidden Markov Models** (Mesaros et al. (2010), Heittola et al. (2011), Heittola, Mesaros, Eronen, & Virtanen (2013), Heittola, Mesaros, Virtanen, & Gabbouj (2013)), or using **Spectral decomposition methods** (Cotton & Ellis (2011), Dikmen & Mesaros (2013), Gemmeke et al. (2013), Heittola, Mesaros, Virtanen, & Gabbouj (2013), Mesaros et al. (2015), Komatsu et al. (2016), Benetos et al. (2016), Benetos et al. (2017)), currently most SED systems are based on **Neural Networks**. There are multiple variants of neural networks, including **Deep Neural Networks** (Cakir et al., 2015), **Convolutional Neural Networks** (CNN) (Cakir et al. (2016), Jeong et al. (2017)), **Recurrent Neural Networks** (RNN) (Parascandolo et al. (2016), Lu & Duan (2017)), and mixes of these like **Convolutional-Recurrent Neural Networks** (CRNN) (Cakir et al. (2017), Adavanne & Virtanen (2017)).

These neural networks and most of the other methods mentioned are learning methods that can be trained on data to produce the desired output. For SED there are two types of data to train on.

Strongly labelled Strongly labelled data means that each audio-file in the dataset has an accompanying annotation containing all sound events in the audio file. Each sound event has a label, a start time and duration. This annotation is the correct output for an SED System.

Weakly labelled For weakly labelled data each audio-file also has an annotation. However, this annotation only contains the labels of each sound event, not the start time and duration. This annotation would be the correct output for a Audio Tagging system, a different kind of system that only concerns itself with the labels present in an audio-file, not their start time and duration.

An example of these 2 data types can be seen in Figure 1.1.

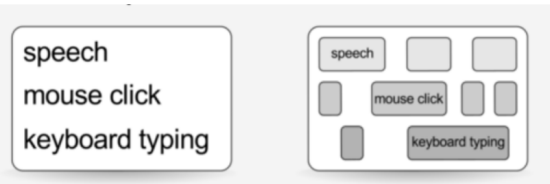


Figure 1.1: Left: Weakly labelled, Right: Strongly labelled. Figure adapted from (Mesaros et al., 2016)

As can be seen in Figure 1.2, strongly labelled SED contains the following steps:

1. Process all audio files and accompanying annotations in the training set by creating a feature-set and annotation matrix, or Event Activity Indicators, for each element in the training set. The feature-set can be created by different methods, section 3 will go into the most popular methods. The annotation matrix is created by dividing the audio file into time steps of between 20-200ms, then the height of the matrix corresponds to all labels that the SED system is training on, and the width the time frames. The elements of the matrix will then indicate if a label is active in a time-frame or not.

2. Train the SED system to output the matrix based on the feature-set on the training set, this training depends on the method used and is more deeply explained in section 4.

3. Process the audio files in the evaluation set by creating their feature-sets.

4. Create annotation matrix for each element of the evaluation set using the SED system.

5. Transform the annotation matrix to the expected form of annotation, by reversing back the time-frames into start time and duration.

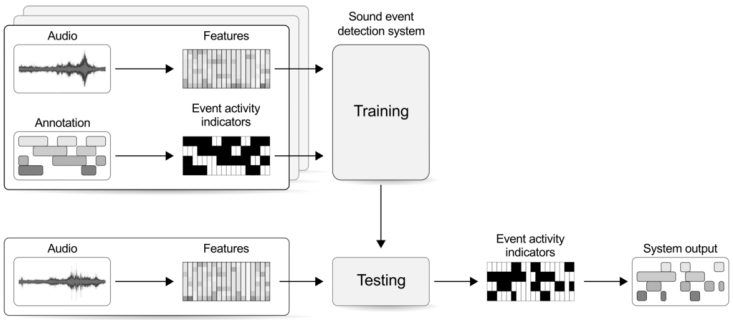


Figure 1.2: SED steps. Figure from (Mesaros et al., 2016)

For weakly labelled datasets the annotations lack start time and duration. This means we cannot make an annotation matrix for a weakly labelled dataset. Therefore an annotation matrix will need to be estimated. A lot of current research is focused on estimating these annotation matrices for SED on weakly labelled datasets (Kumar & Raj (2016), Kumar & Raj (2017), Kong et al. (2017), Su et al. (2017), Chou et al. (2017), Salamon et al. (2017), Wang & Metze (2017), Kumar et al. (2018), Xu et al. (2018)). Because this estimated annotation matrix is estimated differently in each method a more in-depth explanation is given for the current state-of-the-art method in section 4.4.2.

A lot of research is focused on these weakly labelled datasets is because not many strongly labelled datasets are available. It is hard to label sound events exactly because it is very labour-intensive and because it is hard to agree on a specific start time and duration of a certain audio event (Gemmeke et al., 2017). One example of this can be seen when looking at an audio event with the class footsteps. Because of their repetitive nature, there are multiple ways to label these footsteps in an audio file.

- Labelling the presence of a footstep sound so that for each footstep there exists a start and end time.
- Labelling the presence of a series of footsteps so that for each series of footsteps there exists a start and end time.

Even when agreed on one of these definitions, questions still arise. For example, how small does the gap between two series of footsteps need to be before they are considered one series. Weakly labelled data, however, can be created by only rating if a sound-event is present or not in an audio file. This is not only less labour-intensive, but the results are also more easily agreed on.

1.3 Current State-of-the-art Results

The biggest competition for automatic detection of sound elements is the Detection and Classification of Acoustic Scenes and Events (DCASE) competition. This is a yearly event in which current state-of-the-art detection systems are used on public datasets. A requirement for this competition is that each participating team submits a technical report, meaning that current state-of-the-art methods are not only used, but also explained, and weighed against other methods on the same datasets. In the DCASE competition, the evaluation of these state-of-the-art SED systems is on the preferred basis of F1 score and error rate (Mesaros et al., 2016) as a rating. The F1 score measures how many selected items are relevant and how many relevant items are selected in the annotation matrix. The error rate measures how many substitutions, deletions, and insertions are needed to get the correct annotation matrix from the current annotation. The F1 score and error rate are more fully explained in Chapter 2. In this way, the perfect score would be an error rate of 0 and an F1 score of 100%. This weighing is done on a dataset not used for training, called the evaluation set, to ensure the results can be extended to data that the SED system has not seen before. In 2017 (DCASE2017), the DCASE competition had three tasks focusing on SED systems, Task 2, 3 and 4. The task of most interest is task 4.

Task 4 This task focused on a large-scale weakly labelled SED system for smart cars. The dataset was weakly labelled and contained the 17 classes: train horn, Air horn, truck horn, car alarm, reversing beeps, ambulance siren, police siren, fire engine, fire truck siren, civil defense siren, screaming, bicycle, skateboard, car, car passing by, bus, truck, motorcycle, train.

The winner of this task had an error rate of 0.7300 and an F1 score of 51.8% for SED. By using a Convolutional Gated Recurrent Neural network (section 4.4.2).

1.4 The Research Objective

The current study will focus on SED on weakly labelled datasets because weakly labelled datasets are easier to create and usually have the biggest number of classes. This is advantageous for an SED system, because this means more labelling can be done with a single system. The research objective is to formulate and test a new method that uses the ontology of the dataset to shape a network of neural networks. These ontologies are shaped like graphs or trees. In particular, the ontology of the Google Audioset, the superset of the DCASE task 4 dataset, is shaped like a tree. The top 2 layers of this tree can be seen in figure 1.3. This picture shows multiple groupings of the audio-files found in Google Audioset. First the sounds are grouped into: Human sounds, Animal sounds, Natural sounds, Music, Sounds of things, Source-ambiguous sounds and Channel, environment and background sounds. These groupings each also have their own subgroupings, for example Natural Sounds is split into the groups:

Wind, Thunderstorm, Water, Fire. In this way we have a tree that goes from all sounds, into groups, into subgroups of these groups.

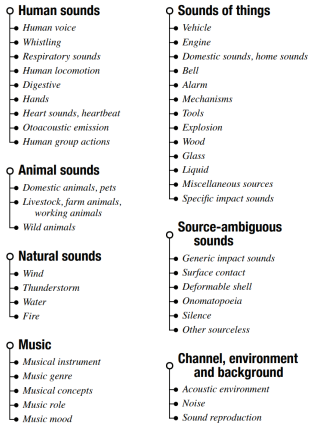


Figure 1.3: Top 2 layers of google AudioSet Tree. Figure from (Gemmeke et al., 2017)

(a) Cheetah, Picture from Buschgardens

(b) Leopard, Picture by Derek Keats

(c) Seal, Picture from National Aquarium

(d) Sea Lion, Picture by Casey Klebba

Figure 1.4: Easily confusable classes

An intuitive example that shows why this tree shape could improve F1 results and simplify training is an image classifier that classifies 4 classes, a cheetah, a leopard, a seal and a sea lion. The hypothesis is as follows: A classifier trained to split all 4 is likely to confuse a cheetah with a leopard, and a seal with a sea lion. If, however, there exist 3 classifiers, one that splits cheetahs and leopards from seals and sea lions, another that splits cheetahs from leopards and another that splits seals from sea lions, these classifiers only need to learn features necessary for their specific task, simplifying training. The classifier that only knows the difference between cheetahs and leopards is less likely to confuse the 2 than the bigger classifier because that is the only thing it is trained to do. Because this would be the true for all 3 networks, this would increase the overall accuracy. A more in-depth explanation of the new method, called branching neural networks, can be found in chapter 5.

An advantage of using ontologies to shape the network is that ontologies are usually already created for datasets. In particular, the Google AudioSet, and therefore also its subset DCASE 2017 task 4, contains a very complex ontology. Meaning this additional input is easy to gather.

To test this new branching neural network method the DCASE 2017 task 4 dataset will be used, because it allows us to compare with current-state-of-the-art methods. The data is split into a training set, that is weakly labelled, and a evaluation set, that is strongly labelled. This strongly labelled set exists to test the methods accuracy using the F1 score and error rate explained in chapter 2. Figure 1.5 shows the classes in this dataset, accompanied by the number of instances for each class.

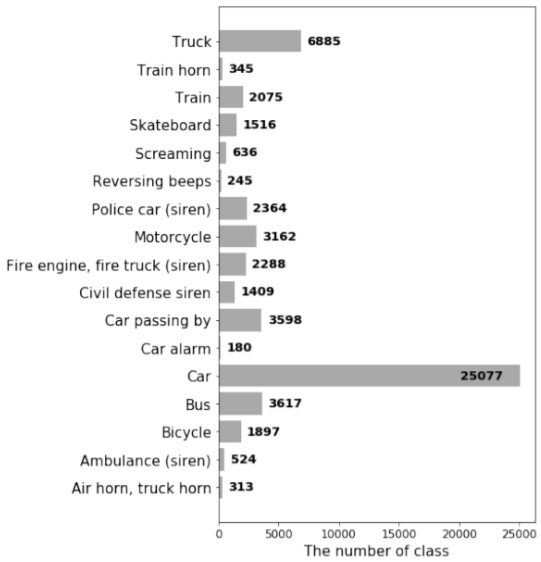


Figure 1.5: DCASE 2017 case 4 challenge labels. Figure from (Lee et al., 2017)

To test the usefulness of this method the following questions will need to be answered:

- Can we successfully build a classifier with this new Branching Neural Network method?
- Does this new method improve in F1 score and error rate on the current winner of the DCASE 2017 task 4 challenge?
- Can current state-of-the-art SED systems be used as the internal tree nodes for the new method? Specifically the Convolutional Gated Recurrent Neural Network?
- What is the training and running time of this method versus the current winning method?
- How much does randomly changing the ontology change the training and running time of the method? Is this change dependant on the shape of the network?
- How much does randomly changing the ontology change the error rate and F1 score?
- Does tuning the ontology based on the number of instances for each class change the results?

Chapter 2

Metrics

To answer the questions we will need a universal way to test the performance of a SED system, which brings us to metrics. This section will go into metrics, by summarizing Mesaros et al. (2016). To determine how well a SED system performs one uses a second set of data besides the training set, called the evaluation set. This set must always be strongly labelled, so that it can be compared with SED output, i.e. the annotation matrix for an audio-file. There are many different types of metrics to evaluate the accuracy of this comparison. The most popular ones are:

True Positive rate: The number of true positives T_p aggregated over the entire evaluation set. A true positive in a segment-based approach, like SED, is when a segment is active, while it is active in the annotation matrix. For the tagging output of an SED, i.e. the labels present without a start-time and duration, this is all labels output from an audio-file, that should be output. See Figure 2.1.

True Negative rate: The number of true negatives T_n aggregated over the entire evaluation set. A true negative in a segment-based approach, like SED, is when a segment is inactive, while it is inactive in the annotation matrix. For the tagging output of an SED this is all the labels that were not output, that should not be output. See Figure 2.1.

False positive rate: The number of false positives F_p aggregated over the entire evaluation set. A false positive in a segment-based approach, like SED, is when a segment is active, while it is inactive in the annotation matrix. For the tagging output of an SED this is all the labels that are output, that should not be output. See Figure 2.1.

False negative rate: The number of false negatives F_n aggregated over the entire evaluation set. A false negative in a segment-based approach, like SED, is when a segment is inactive, while it is active in the annotation matrix. For the tagging output of an SED this is all tags that are not output, that should be output. See Figure 2.1.

Selected items: The number of false and true positives aggregated over the entire evaluation set. See Figure 2.1.

Precision: Precision P is a percentage of how many selected items are relevant. Defined as follows:

$$P = \frac{T_p}{T_p + F_p}$$

See Figure 2.1.

Recall, Sensitivity or Recognition rate: Recall R, also called Sensitivity or Recognition rate, is a percentage of how many relevant items are selected. Defined as follows:

$$R = \frac{T_p}{T_p + F_n}$$

See Figure 2.1.

F-score: F-score is a measure that weighs recall against precision. Defined as follows:

$$F_\beta = (1 + \beta^2) \cdot (\frac{R \cdot P}{(\beta^2 \cdot P) + R}$$

The F_1 score is the harmonic mean of precision and recall. The harmonic mean tends strongly towards the least elements of the list, to mitigate the impact of large outliers and aggravate the impact of small outliers. An F-score with $\beta < 1$ will weigh recall lower than precision, while an F-score with $\beta > 1$ will weigh recall higher than precision.

Specificity: Specificity S_p is a percentage of how many non-selected elements are truly negative. Defined as follows:

$$S_p = \frac{T_n}{F_p + T_n}$$

Error rate: Error rate E measures errors in terms of insertions, deletions and substitutions. For Sound Event Detection this is usually done using segment-based error rate. This segment-based error rate looks at all the segments in an annotation matrix and determines how many segments should be changed from one label to another (substitution S), how many need be deleted (deletions D), and how many need to be added (insertions I).

$$S = \min(F_n, F_p)$$

$$D = \max(0, (F_n - F_p))$$

$$I = \max(0, (F_p - F - n))$$

The error rate is then defined as follows:

$$E = \frac{S + D + I}{T_p + F_p}$$

A lower error rate means, less substitutions, deletions and insertions compared to the True and False Positives, so a better accuracy.

Usually multiple measures are used together to strengthen the accuracy claim. Recognition rate with false positive rate, false negative rate with false positive rate, Precision with Recall, Sensitivity with Specificity, or F-score with error rate. This is done because each accuracy looks at specific way to determine accuracy and on their own they may create a very skewed picture. For example if only Recall is used, how many irrelevant items are selected is never looked at, so if there were many irrelevant items with only a few relevant items in the dataset, recall is more likely to be better just based on the data. Precision does take irrelevant items into consideration, counteracting this effect. For this specific combination F-score was created.

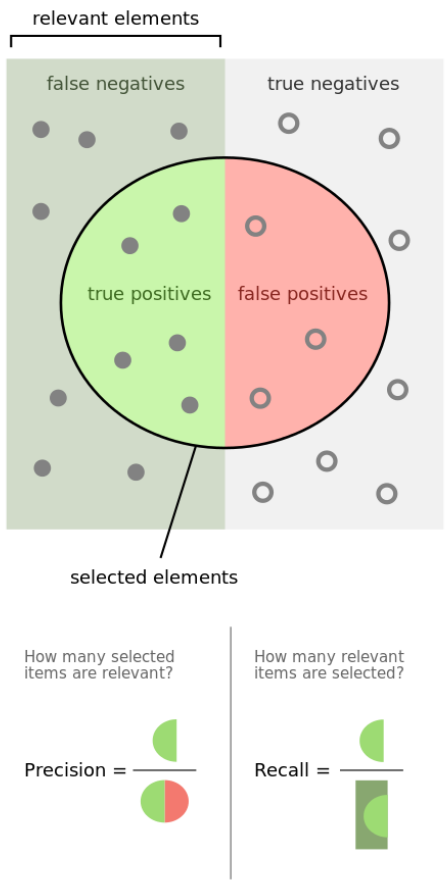


Figure 2.1: True Positives, True Negatives, False Positives, False Negatives, Recall, Precision. Figure from (Walber, n.d.)

Chapter 3

Features

With the metrics defined, we can quantify the performance of a SED system. The first step for a SED system, as can be seen in Figure 1.2, is transforming audio-files into feature-sets. As such, a way to gain features needs to be defined. However before we can discuss how to gain features from audio files, we first need to discuss how to represent audio files in some form of audio representation.

3.1 Waveform

Sound is generated by displacements and oscillations of air molecules (Müller, 2015). This alternating pressure in the air can be perceived as a wave. Sound can then be represented by plotting the pressure-time plot of air in a certain location. The resulting representation is referred to as waveform. This is the simplest audio representation.

3.2 Mel Features

In the past the Fourier transform (Müller, 2015), calculated based on the waveform, was often used to gain features to train SED systems on. However, current methods mostly use features gained by looking at the mel scale, mel coming from melody. This scale is created using a log transformation of the frequency space, because this is most similar to the human perception system. One feature type that uses this mel-scale is the Mel Frequency Cepstral Coefficients (MFCCs, Fayek (2016)). An example of these MFCCs can be seen in Figure 3.3b. MFCCs were created to accurately represent the shape of the human vocal tract, hence their popularity in speech recognition. They, however, used to be very popular for SED systems too.

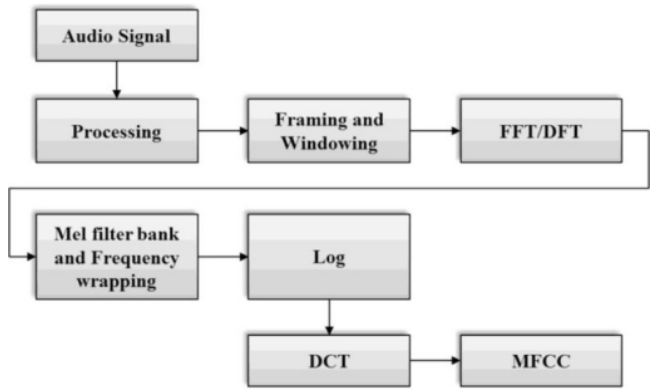


Figure 3.1: MFCCs steps. Figure from (Chung et al., 2013)

The steps for creating MFCCs are as followed, also shown in Figure 3.1:

1. Frame the audio-file into short frames and apply a windowing function to each frame, usually the hamming window $w(n)$, by multiplying each sample in the frame with the window formula. For the hamming window function this is:
$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right)$$
where n is the sample number and N is the amount of samples in the frame.

2. For each frame calculate the periodogram P, also known as the power spectrum, using the formula:
$$P(x_i) = \frac{|F(x_i)|^2}{N}$$
where $F(x_i)$ is the discrete fourier transform, usually calculated with the FFT algorithm, x_i is the signal of frame i and N is the number of samples used for the discrete fourier transform, typically N is 256 or 512.

3. Apply the mel filterbank to each of these periodograms. Applying a mel filterbank means multiplying filter $H_m(k)$ with the frequency k for a set number of filters, for each k in $P(x_i)$, for each x_i. Each filter is given by the formula:
$$H_m(k) = \begin{cases} 0 & k < f(m-1) \\ \frac{k-f(m-1)}{f(m)-f(m-1)} & f(m-1) \leq k < f(m) \\ 1 & k = f(m) \\ \frac{f(m+1)-k}{f(m+1)-f(m)} & f(m) < k \leq f(m+1) \\ 0 & k > f(m-1) \end{cases}$$

Where m is the mel value, $f = 700(10^{m/2595} - 1)$ the formula for transforming mel into frequency, and k is a frequency sample. Typically 40 filters are taken, with m values equally spaced between the minimum and maximum mel value of the periodogram. See Figure 3.2 for an example of these filters and their result on a periodogram. In this Figure (a) shows the full filterbank, with all filters and (b) shows the periodogram after using all these filters on some audio file. Furthermore (c) shows a single filter, filter 8, and (d) shows the periodogram while only using this filter on the audio-file. For (e) and (f) this same thing is shown for filter 20.

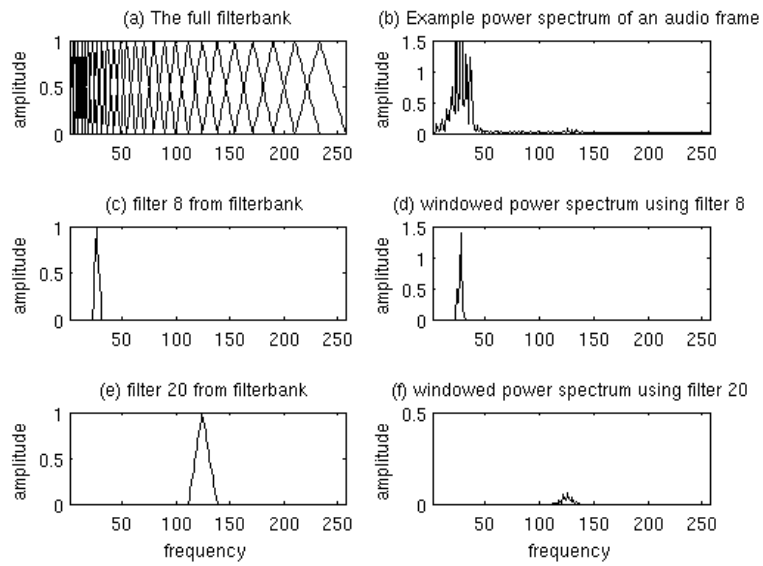


Figure 3.2: Plot of mel filterbank and windowed power spectrum. Figure from (Lyons, 2013)

4. Take the log of each of the resulting elements, creating log filterbank frames.

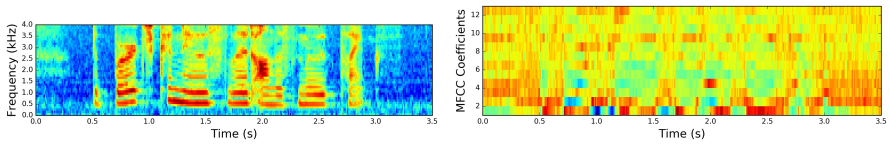
5. Take the Discrete Cosine Transform (DCT) of the log filterbank frames, for coefficient 2 - 13. The Discrete Cosine transform is given by the following formula:

$$C_x(k) = \begin{cases} \sum_{n=0}^{N-1} 2x \cos\left(\frac{\pi}{2N} k(2n+1)\right) & 0 \leq k < N \\ 0 & \text{otherwise} \end{cases}$$

where k is a frequency sample on the log filterbank frames, C_x is coefficient x and N is the max coefficient.

6. Join all DCT frames into one feature-set, where each DCT frame is one vertical row. The result is an image similar to Figure 3.3b, where the amplitude is now given by a color.

Currently, however, SED systems most often use log-mel energies, mel-filterbanks, mel-spectrograms, or mel band energies. The difference between MFCCs and these others is that one of the steps, called the Discrete Cosine Transform (DCT) step, is skipped. This results in a feature-set more similar to Figure 3.3a. In the past this step would not be skipped because the most prevalent learning algorithms like Gaussian Mixture Models and Hidden Markov Models had better results on the MFCCs, however, neural networks work just as well on both versions. So skipping this step increases training speed and decrease running time of the SED system.



(a) Mel Spectogram

(b) MFCCs

Figure 3.3: Mel-spectogram and MFCCs of same audio-file. Figure from (Fayek, 2016)

Chapter 4

Current Methods for SED

All current state-of-the-art SED methods and especially those with the highest accuracy ratings in the DCASE challenges, rely on neural networks. Because of this the basics of neural networks will be introduced here. Section 4.4.1 will explain the baseline method given for DCASE2017 task 4 as a comparison method. Section 4.4.2 will explain the winning method of DCASE2017 task4.

4.1 Neural Network

This section summarizes Karn (2016b). (artificial) neural networks are a learning algorithm inspired by the human brain. The human brain consists of millions of neurons connected to each other. To mimic this behavior in computers a neural network consists of a network of artificial neurons or nodes, however not millions of them. At the simplest level one of these neurons consists of one or more input, a function that acts on those input, and one or more output, as shown in Figure 4.1.

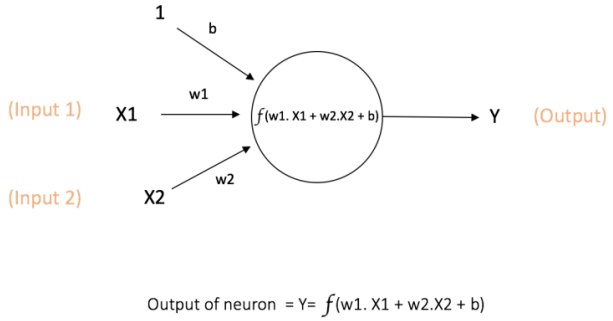


Figure 4.1: A single artificial Neuron. Figure from (Karn, 2016b)

Each of the input and output has a corresponding weight that mimics the strength of a neuron connection. All input values are multiplied by their weight and added together before they are used in the function. There are multiple different function, called activation functions, that are usually used in these neurons. The 3 most popular functions are seen in Figure 4.2:

Sigmoid Takes any value as input and outputs a value between 0 and 1, as such we know that sudden very high or low inputs will not greatly influence the entire network.

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

Tanh The sigmoid function extended from -1 to 1, makes it so negative values can be output, causing more complex interactions between neurons.

$$\tanh(x) = 2\sigma(2x) - 1$$

ReLu Outputs the input when it is higher than 0, has higher convergence than Tanh, but is more fragile because high inputs will greatly influence the entire network.

$$f(x) = \max(0, x)$$

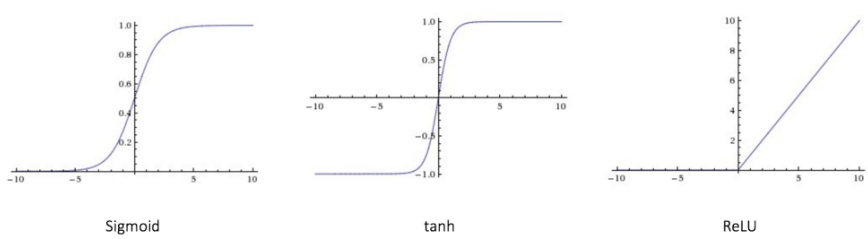
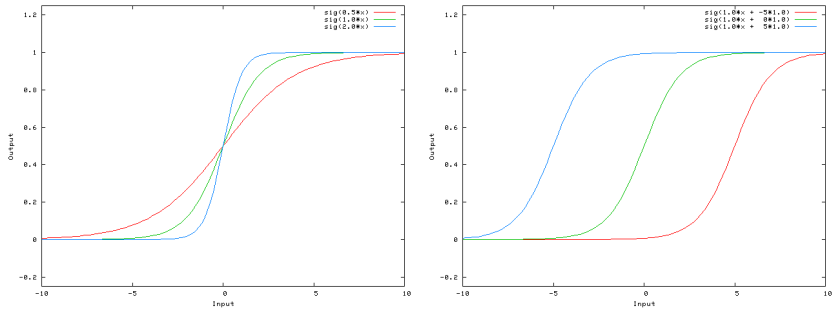


Figure 4.2: popular activation functions. Figures from (Karn, 2016b)

Usually a constant input is added with weight b, called the bias, so the activation function can be shifted to the left or the right depending on this bias, as shown in Figure 4.3.



(a) Effect on output with 3 different weights

(b) Effect on output with 3 different biases, same weights.

Figure 4.3: The use of bias. Figure from (Kohl, 2010)

Neurons are connected to each other in layers to form a network. They are usually divided into 3 categories:

Input node The neurons on the first layer only have 1 in-going connection, one of the input features, hence the name input nodes. Their output go towards the next layers.

Output node The neurons in the last layer only have 1 outgoing connection, all output from the neurons on the last layer combined are the output of the neural network. These last neurons are therefore also called output nodes. Their input comes from previous layers.

Hidden node Any node in the intermediate layers, between the input and the output layer, are called hidden nodes. This is because they are only connected to other neurons.

A neural network in which every layer is only connected to the previous and next layer is called a feedforward neural network. This was the first and simplest type of neural network. There can be any number of layers. An example of a feedforward neural network can be seen in Figure 4.4. A feedforward network with only 2 layers, the input layer and the output layer, is called a Single Layer Perceptron. If there are one or more layers containing Hidden nodes, it is called a Multi-Layer Perceptron. If a neural network has many layer it is called a deep neural network (DNN). How many layers are necessary before it is called a deep neural network depends on who you are talking to and the application of the neural network. However conventionally it is at least more than 3 layers.

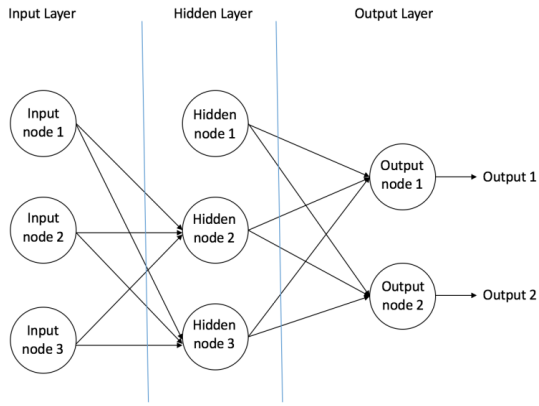
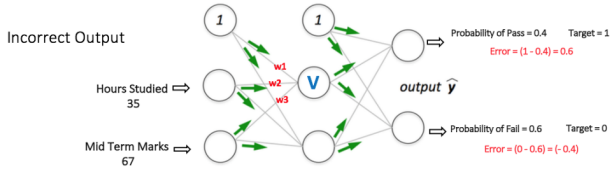
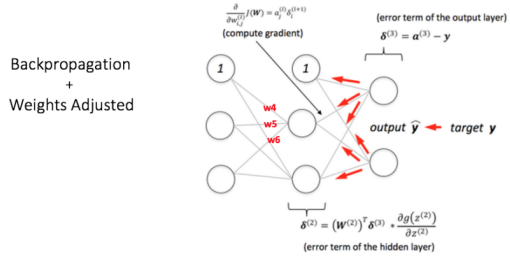


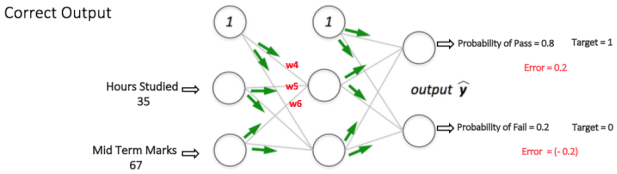
Figure 4.4: Feedforward Neural Network. Figure from (Karn, 2016b)



(a) Incorrect output



(b) Backpropagation



(c) Correct output

Figure 4.5: Neural Network Learning. Figures from (Karn, 2016b)

To train a neural network a training algorithm is used to update the neural network to output the correct data given the input. The simplest algorithm that does this is called back-propagation. This algorithm works as follows:

1. Create all nodes and connections with randomized weights.

2. Input features into the input layer of the neural network. Then use the weights and functions in the neurons to calculate output. This process is known as forward propagation.

3. Calculate total error in all output nodes compared to correct output, using a loss function. There are multiple loss functions L, the simplest being:

$$L = \frac{1}{N} \sum_{n=1}^{N} (p_n - a_n)$$

   where p_n is the nth predicted output and a_n is the nth actual output. This loss function is called the Mean Bias Error (MBE) and finds the average distance between the prediction and the actual observation.

4. Adjust all weights in the network in such a way that total error is reduced. This can for example be done with gradient descent: Calculate the derivative of the output in terms of the weights, adjust weights based on the derivative. The most popular method for calculating these derivatives is known as back-propagation. The exact math for calculating this derivative with back-propagation can be found in Lucas Schuermann (2016) and an example can be seen in section 7.1.

5. Repeat step two to four for all elements in the training-set.

An example of this process can be seen in Figure 4.5. In this figures (a) shows step 2, (b) shows step 3 and 4, and (c) shows step 2 in the second loop.

Some popular loss functions used in step 3 include:

Mean Squared Error (MSE)

$$L = \frac{1}{N} \sum_{n=1}^{N} (p_n - a_n)^2$$

Finds the average squared distance between the prediction p_n and the actual observation a_n, in this way predictions with big differences are more heavily penalized.

Mean Absolute Error (MAE)

$$L = \frac{1}{N} \sum_{n=1}^{N} |p_n - a_n|$$

Finds the average absolute distance between the prediction and the actual observation, in this way predictions with big differences are more heavily penalized, however it is more robust to outliers as the difference is not squared.

Mean Squared Log Error (MSLE)

$$L = \frac{1}{N}\sum_{n=1}^{N}(\log(a_n + 1) - \log(p_n + 1))^2$$

Finds the average squared log distance between the prediction and the actual observation, in this way predictions with big differences are less penalized than for MSE or MAE.

Cross Entropy Loss

$$L = -\frac{1}{N}\sum_{n=1}^{N}(a_n \log(p_n + 1) + (1 - a_n)\log(1 - p_n))$$

Penalizes predictions for how far they are from actual observations with values 0 or 1, as such it is used for tasks where the output should be either 0 or 1, like SED systems.

Hinge loss

$$L = \frac{1}{N}\sum_{n=1}^{N}\max(0, m - p_n a_n)$$

Where m is some customized margin. Penalizes predictions from how far they are from actual observations with value $-m$ or m, in such a way that $|p_n|$ goes to ∞. In this way the hinge loss creates predictions with high levels of certainty, because it is less likely that $-m < p_n < m$.

4.2 Convolutional Neural Network

Convolutional Neural Networks (CNNs) are a type of neural networks invented specifically for Object Detection: Finding the position of an object in an image. However, they have also found their use in Sound Event Detection. this section summarizes Karn (2016a) to explain these CNNs. The difference between feed-forward and convolutional neural networks is the ability for convolutional neural networks to find important groups or combinations in the input that improve the performance of the neural network, removing the need for manual selection of these combinations. This is done by having three operations which a feed-forward neural network does not have, before continuing on like a feedforward neural network. resulting in four operations.

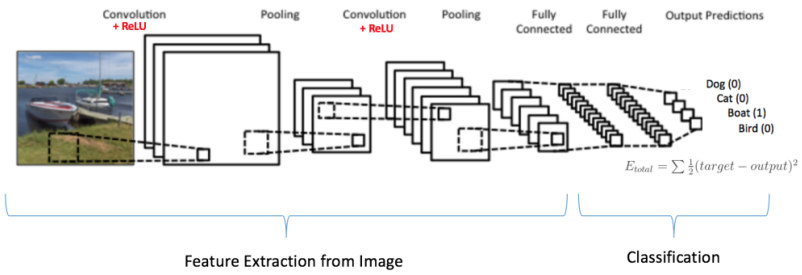


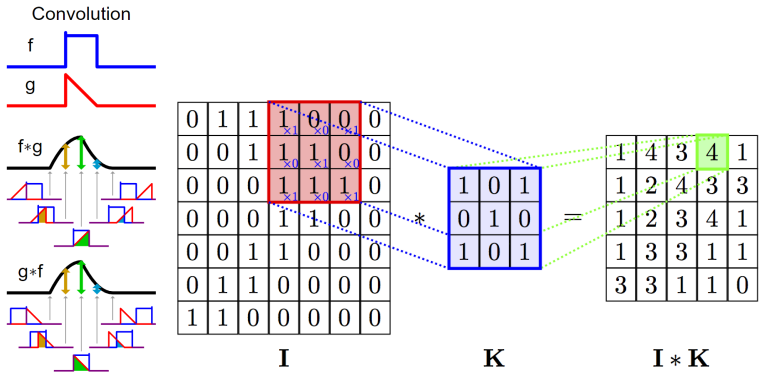
Figure 4.6: Convolutional Neural Network. Figure from (Karn, 2016a)

The four operations of a CNN, shown in Figure 4.6 are:

Convolution The convolution operation, from which CNNs gained their name, is based on the mathematical convolution:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

where $f, g : \mathbb{R} \to \mathbb{R}$ are two random functions and $t \in \mathbb{R}$.



(a) On functions

(b) On a feature-set

Figure 4.7: Convolution. Figure from (Karn, 2016a)

The effect of this operator on a two random functions can be seen in Figure 4.7a. In CNNs this step is mimicked by having the feature-set be $f(t)$ and a smaller set, called a filter, kernal or feature detector, be $g(t)$. Then this filter is passed over the feature-set and the dot product of the two is output in one pixel in a new Feature Map, also called convolved feature map or

activation map. This effect can be seen in Figure 4.7b, where $f(t) = I$ and $g(t) = K$, an animation of this can be found in Karn (2016a). The size of the steps taken when the filter is passed over the feature-set is called the stride. Because the size of the feature map is smaller than the original, some convolutional filters use zero-padding to pad the feature map using zeros at the edges. Some standard filters used for image manipulation can be seen in Figure 4.8

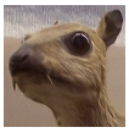
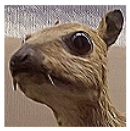
Original	*Gaussian Blur*	*Sharpen*	*Edge Detection*
$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$
			

Figure 4.8: Some Standard 3x3 Filters. Figure from (Karn, 2016a)

The idea of these filters is that the feature map, the output of the filter set, captures some features of the input. For example the edge detection filter, seen in Figure 4.8, will bring to focus the edges in an image. CNNs learn similar filters during the training process. However some parameters must be given to the CNN. These parameters include the number of filters, called depth, the filter size, and the layout of the entire network. Usually more filters result in better, but slower, recognition, because more features need to be extracted.

Non Linearity (ReLU) The non-linearity operation, tries to introduce non-linearity into the CNN. This non-linearity is introduced by using an activation function, for example the Sigmoid, Tanh or ReLu functions seen in Figure 4.2. The chosen function is used on every element of a feature map, resulting in a map reffered to as the rectified feature map. The most used activation function is the ReLU function, because this has been found to have the best performance. Therefore this operation is sometimes also called the ReLU operation.

Pooling or Sub Sampling The pooling operation reduces dimensionality of a feature map, but retains the most important information. This is needed to solve what is known as the curse of dimensionality: The higher the dimensionality of the problem, the more data is needed to learn the correct model. This operation also reduces the likelihood of over-fitting, makes the network resistant to small distortions and helps create an almost scale

invariant representation of the input. Over-fitting happens when a learning method corresponds too closely to an exact data-set and may therefore fail to fit to additional data-sets. Scale invariance means the features do not change with the size of the input. Reducing the dimensionality is done using spatial pooling, also known as sub-sampling or down-sampling. Spatial pooling is done by defining a window, sliding it over a feature map and outputting the result in a new pooling map. However the function of the window is different for spatial pooling, in the convolution operation a dot product was used, while for spatial pooling one of the following functions is used:

Max pooling Take the max element in the window.

Average pooling Take the average of all the elements in the window.

Sum pooling Take the sum of all the elements in the window.

An example of max pooling is given in Figure 4.9. Just as in the convolution operation, these windows have different sizes and the step size for sliding is called the stride. The pooling is done for each of the rectified feature maps gained from the previous step.

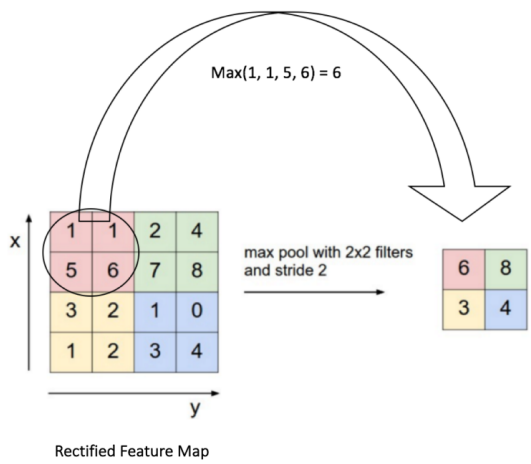


Figure 4.9: Max Pooling. Figure from (Karn, 2016a)

Classification The classification operation uses the output of the pooling layers as input of a fully connected feedforward neural network. A feedforward neural network is called fully connected when every neuron in each layer is connected to all neurons in the next layer. This means this operation works by using the feedforward neural network explained in section 4.1, but with a different input.

Figure 4.6 shows the first CNN used for object detection, the LeNet CNN (LeCun et al., 1998). The steps for this CNN where as followed:

1. Initialize all filters parameters and weights with random values.
2. Use forward propagation on an element in the training set. By doing the steps: convolution, ReLU, pooling, convolution, ReLU, pooling, classification.
3. Calculate total error of the output
4. Use back-propagation to adjust all weights and filters to reduce total error of the output.
5. Repeat Step 2 to 4 with all elements in the training-set.

These steps are very similar to the steps of a standard neural network, this is because the same forward and back-propagation steps can be taken. The difference is the math involved for these propagation steps. The math to get the derivatives related to the convolution and pooling steps can be seen in Agarwal (2017). In newer CNNs the layer ordering and depth have changed compared to the LeNET CNN, however because they use the same operations forward and back-propagation are still used to train them. Just as with standard neural networks, at the end of this training a CNN should output the correct results for most elements in the training-set and on data that is similar to it.

4.3 Recurrent Neural Network

This section summarizes Banerjee (2018). One big problem with Standard Neural networks is that these only act on instantaneous input. If a network is trained to identify parts of a sentence, the network will not use the knowledge of the last sentence for the new sentence, unless they are both input at the same time. Recurrent Neural Networks (RNNs) try to fix this context problem by allowing information to persist in the network. The way this is done is by looping a hidden state back as partial input for the next input. Then this network can be seen as a recurring network of neural networks, as seen in Figure 4.10, where a RNN is seen as a looping neural network on the left side and multiple connected copies of a neural network on the right. The representation of an RNN as a recurring neural network is also called an unrolled RNN. This recurrence makes them very useful for data that is dependant on context, such as sequences of events, like in audio-files used for Sound Event Detection. Another big advantage of these RNNs is that they are not constrained by fixed size input. This is because the input can be split up into multiple smaller files, as long as the data can be put in a sequence. For audio files used in SED system this sequencing is performed by framing the files into multiple time steps.

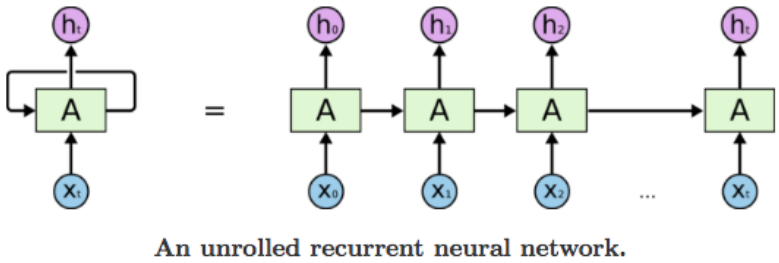


Figure 4.10: RNN unrolled. Figure from (Banerjee, 2018)

RNNs are divided into the following categories, based on their input and output:

One to One A standard neural network, not recurrent.

One to Many A neural network that goes from one input to many output, for example from an audio-file containing speech, to the words spoken.

Many to One A neural network that goes from multiple input to one output, for example from a sentence, to a value representing its hostility.

Many to Many A neural network that goes from multiple input to multiple output, for example from an English Sentence to a Dutch one.

Synced Many to Many A neural network that goes from multiple input to multiple output, in such a way that each input has an output, for example from an audio-file split in frames to a list of sound events that happened on each frame.

As seen in Figure 4.11, where blue is the output, red is the input and green is a part of an unrolled RNN.

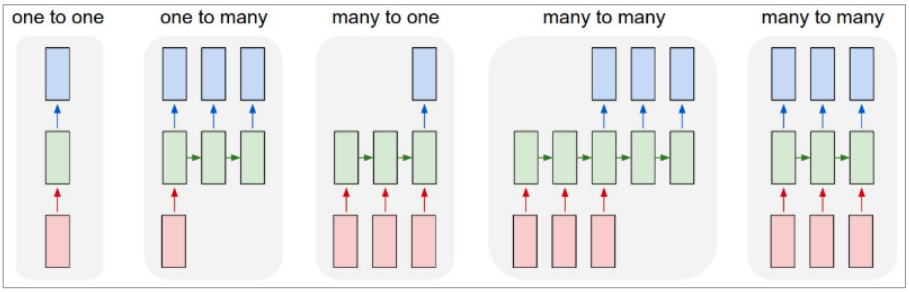


Figure 4.11: RNN types. Figure from (Karpathy, 2015)

The most important part for the RNN itself is how the hidden state h_t is created, how it is updated based on input and how this hidden state influences output. In the simplest case this hidden state is a single vector. This vector is initialized with the zero vector $h_0 = \overrightarrow{0}$ and updated on the input with the tanh function such that $h_t = \tanh(W_{hh}h_{t-1} + W_{hx}x_t)$, where W_{hh} is a matrix of hidden weights, W_{xh} a matrix of input weights and x_t the input vector at step t. Then the output at step t, y_t, is calculated with $y_t = W_{yh}h_t$ where W_{yh} is a matrix of output weights. These weights are updated using a back-propagation algorithm:

1. Unroll the RNN until a standard feedforward neural network is obtained, an example of which is shown in Figure 4.10.
2. Set the input of this network to be the initial hidden state and the input at every time-step.
3. Perform forward and backward propagation as with a standard neural network
4. Average all gradients at each layer, so the weight change is the same at each recurrent step.
5. Repeat steps 1 to 4 for all elements in the sequence made from all data in the training set.

A problem standard RNNs have is that the information most important to the hidden state is always the most recent input. However the most important information for the sequence might be less recent. To solve this a variant of RNNs called Long Short Term Memory Networks (LSTM) was created. Because this variant solves many problems with the original RNNs, LSTM models are used more often than the original and sometimes the terms for RNN and LSTM are used interchangeably. The only difference between a standard RNN and a LSTM is the update function. The update function is changed to:

$$h_t = o_t * \tanh(C_t)$$

using

$$o_t = \sigma(x_t W_{xo} + h_{t_i} W_{ho})$$
$$C_t = \sigma(f_t * C_{t-1} + i_t * \tilde{C}_t)$$
$$f_t = \sigma(x_t W_{xf} + h_{t_i} W_{hf})$$
$$i_t = \sigma(x_t W_{xi} + h_{t_i} W_{hi})$$
$$\tilde{C}_t = \tanh(x_t W_{xc} - h_{t_i} W_{hc})$$

where the weights are given by the W matrices, x_t signifies the input at time t, h_t the hidden values at time t, $*$ is the operator for element-wise multiplication, and σ is the sigmoid function given by $\sigma(x) = \frac{e^x}{e^x+1}$. i, f and o are called the input, forget and output gate respectively. This is because the σ-function has

a value between 0 and 1. This means these input, forget and output functions gate how much information is let through. These gates have the same dimension as the hidden state. Because the update steps is the only change in this type of RNN, the steps to train an LSTM stay the same as for an RNN, only the exact math changes.

4.4 Current State-of-the-Art

Now the basics have been discussed the current state-of-the-art methods can be described in detail. These details are necessary because the branching method proposed will use current state-of-the-art methods as part of its internal structure.

4.4.1 Baseline Method for DCASE2017 Task 4

The baseline for DCASE2017 task 4 is a feedforward neural network (FFNN, section 4.1). As input the audio-files are transformed to log mel spectograms with frames of 40ms. This neural network is trained on each frame as a separate input, losing context, but making it possible to work with different length audio files. As such each frame uses a 50% overlap to get back a bit of context. The mel-spectrogram uses 40 mel bands covering the frequency range 0 to 22050Hz. The FFNN consists of 2 fully connected layers, with 50 neurons each. These layers both had a 20% dropout rate, which means that during training a random 20% of the neurons where not used to decrease overfitting (Srivastava et al., 2014). The Adam algorithm is used as its gradient-based optimizer (Kingma & Ba, 2014) for back-propagation. Training is performed for 200 epochs using a learning rate of 0.001 (Mesaros et al., 2017). The SED results for this FFNN were gained by running the FFNN over each frame of an audio file and smoothing out the results by using a running window median function because of the 50% overlap. The tagging results were gained by using the maximum output for each tag from this smoothed output.

4.4.2 Convolutional Gated Recurrent Neural Network

The SED system by Xu et al. (2018), uses a Convolutional Gated Recurrent Neural Network (CGRNN) trained on weakly labelled data to create a SED system. In Task 4 of the DCASE2017 contest it came in first place overall because of its audio tagging results, but second place in SED with an error rate of 0.7300 and an F1 score of 51.8%.

To get these results, first the audio files are split into frames of $41.7ms$ and transformed to log mel spectrograms. Then these spectrograms are fed into three gated convolutional neural network blocks (section 4.2) created using gated linear units (GLUs, (Dauphin et al., 2016), as can be seen in Figure 4.12. These GLUs, similar to the gates in the LSTM network (section 4.3), control the amount of information that flows to the next layer. As such GLUs increase

the performance of a neural network by reducing the vanishing gradient problem (Nielsen, 2018), a problem in which earlier layers learn more slowly compared to neurons on the later layers. This is due to the distance the gradient of the earlier layers have from the output compared to the later layers. This problem decreases prediction accuracy on deeper neural networks. These GLU blocks are followed by a bi-directional recurrent neural network (Bi-RNN, Schuster & Paliwal (1997)), which is a variant of RNN that uses context from both before and after the frame it is currently analyzing. The final step consists of two FFNNs (section 4.1, one that uses a sigmoid activation function and another that uses softmax as the activation function. These two FFNNs together predict the tags present in each frame, which created the estimated annotated matrix. These annotations are then averaged for each class to get the predicted audio-tags. To train the neural networks, stochastic gradient descent (SGD) is used for the back-propagation with the cross-entropy loss function, seen in section 4.1. This loss function is calculated over the tag prediction of the entire file. To compensate SGD's fluctuating performance while training, system results fusion (SRF) was used to stabilize and improve results. SRF fuses the results of multiple systems so as to create one superior system. In CGRNN, SRF is used in two ways. First the results of the same system are fused during different steps in the training process, then the posteriors of systems with different configurations are averaged to ensure a more stable solution.

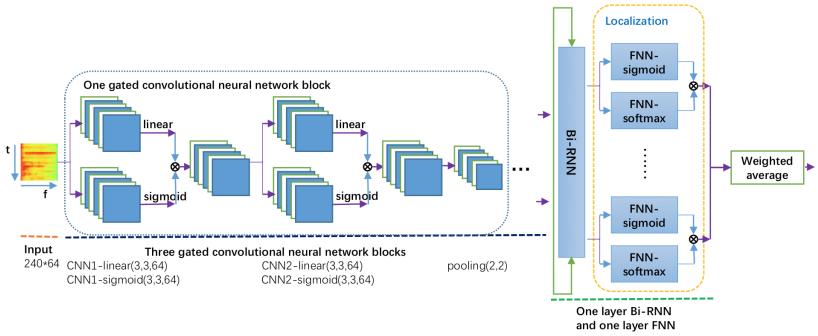


Figure 4.12: Convolutional Gated Recurrent Neural Network. Figure from (Xu et al., 2018)

Chapter 5

A New Method: Branching Neural Networks

At the moment humans are still better at SED than state-of-the-art SED methods (Kim & Pardo, 2018). This implies that we might be able to improve SED methods by using human knowledge.

Humans tend to group sounds together by similarity, known as the law of Prägnanz. This law states that humans tend to group things together in a way that is regular, orderly, symmetrical and simple. This manifests itself with a tendency to group objects together when humans have to deal with a great number of objects. Examples of this are the ontologies that humans often create for sounds. These ontologies are networks of different groupings with information on how groupings relate, such as which group is a subgroup of another and which elements belong to which grouping. In this way these ontologies contain information about which elements we find similar and why. We might use the human grouping ability, used to create these ontologies, to improve SED results.

The method proposed in this report consists of training multiple neural networks, used as a SED system, in a tree shape inspired by the ontology. Each neural network would then focus on a subtask of the full system in such a way that each neural network decides the best next neural network to use. This works by having the same input format for each neural network as a standard SED system, but a different classification task depending on its position in the tree. An example of one an ontology-inspired tree can be seen in Figure 5.1. For this example the method would consist of a first neural network that differentiates *Vehicle* from *Alarm* and a second neural network trained on differentiating the *Vehicle* category into *MotorVehicle*, *NonMotorizedVehicle* and *Train*. This would continue until each split in the tree had its own neural network. Henceforth, we will refer to this method as a branching neural network.

An advantage of such a branching neural network is that it could be created using any SED system as the internal nodes, because the input and output

would be the same between the different SED systems. If necessary, multiple types of SEDs could be used in the same tree if it turns out, for example, that one SED is more suited for a certain subsection and another SED is more suited for another subsection.

In this way the proposed method behaves as a decision tree built of different groupings, in which neural networks make complex choices on which groups follow from which.

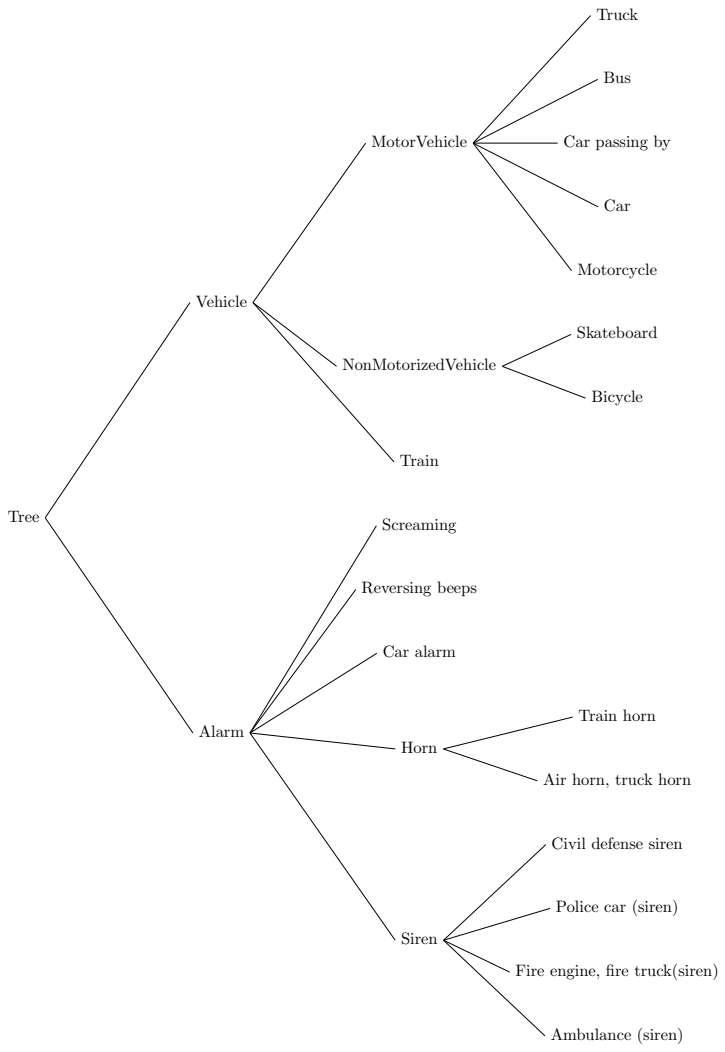


Figure 5.1: DCASE 2017 tree.

Chapter 6

The Basic Branching Neural Network

Branching neural networks are based on the idea of combining neural networks with decision trees in such a way that each decision in the tree is made by a neural network. At each node in the tree in the branching neural network, a decision is made as to which neural network should be used next. As such we can look at decision trees to decide how to create these trees and make these decisions.

The simplest way that trees are formed for decision trees is manually. Variants of decision trees exist that are formed automatically, using techniques like induction (Quinlan, 1986). But using hand-made decision trees is an easier way to test the viability of this hybrid method. Therefore, we will use the latter.

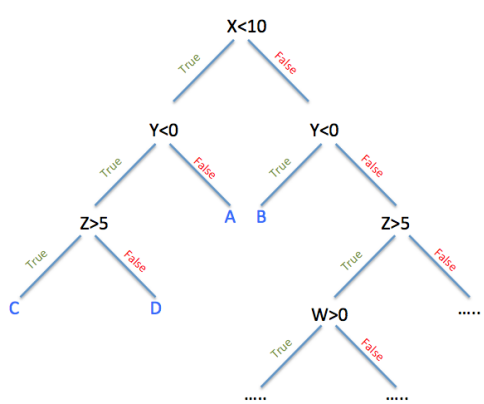


Figure 6.1: Example of a Decision Tree. Figure from (Cavaioni, 2017)

For decision trees a dataset is split each time by a logical decision. An example of a decision tree can be seen in Figure 6.1. In this decision tree, the first decision is of the form $x < 10$ that splits a dataset into data that contains elements where the variable $x \geq 10$ and elements where $x < 10$. The advantage of using neural networks for these decisions is that the decision can be more complicated, or needs to be less well-defined. An example of this is the decision $x = High$ or Low, where $High$ and Low are determined by human labeling of data. As another example, for the classes sea lion, seal, cheetah and leopard, one might imagine the tree in Figure 6.2, in which the grouping is made by similarity according to some underlying criteria (i.e. visuals).

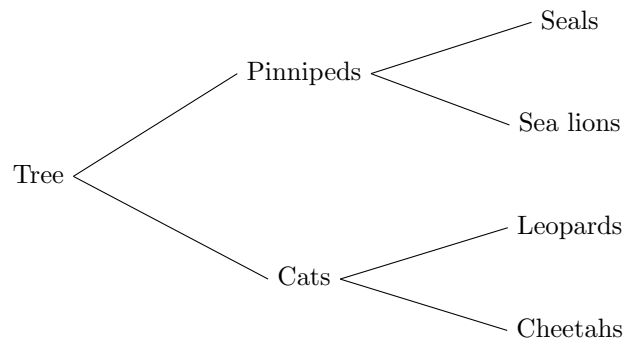


Figure 6.2: An example tree.

6.1 Intuitive Reasoning

As branching neural networks are derived from a combination of decision trees and neural networks, the intuitive reason behind why they work can also be derived from these methods. The intuitive reasons inherited from decision trees are the following:

- As each split is done to find the dissimilarities in the most dissimilar groupings, each split should be the easiest task possible.
- As each split in the tree is a simpler task than the whole SED task, a neural network trained to do these simpler task should be more accurate.
- If one task can be shown to have bad results, we can change the shape of the tree to remove this task completely or partially.

An example that illustrates these reasons is the animal example shown in Figure 6.2. In this example the easiest split is the task that separates Pinnipeds and Cats, because their visual dissimilarities are greatest. This split is intuitively easier than splitting all 4 at the same time. A second split, that splits Cheetahs

and Leopards, is again intuitively easier than splitting all 4 at the same time, as it is a subtask already. If it turns out that the Cats Pinnipeds split was not easy, we could reshape the tree into for example the tree in picture 6.3, showing the third intuitive reason stated.

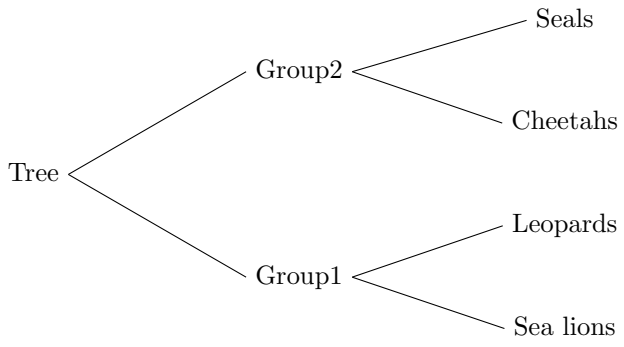


Figure 6.3: An example tree.

However the advantages of a branching neural network are not limited to the advantages given by decision trees. Due to its composition the method also has the advantages of a neural network. One example is that the decision needed for the tree can be made even if it is complex or non-linear, which is not possible in decision trees. Because neural networks extend well to unseen data, branching neural networks might also extend well to unseen data, an advantage decision trees normally do not have. Additionally, there are many different types of neural networks used for many different applications, such as convolutional neural networks, recurrent neural networks and feedforward neural networks explained in section 4.1. As such by changing the internal neural network, the branching neural network might be able to generalize to many different applications and input data types. It could even be possible to have many different neural networks inside the tree itself, making it so that each decision can be tuned to fit to the decision type and its input data type.

6.2 How it Works

The basic branching neural network consists of multiple neural networks that each make a decision based on a predefined tree. Similar to a standard neural network, this branching neural network will need a list of outputs and a set of labelled data to train on. Unlike a neural network a tree also needs to be created, as this tree is not automatically generated. The tree divides the output into multiple sub-decisions. The simplest tree is the tree with only one decision, rendering it a standard neural network. An example of this simple variant for the more complex tree given in Figure 6.2, can be seen in Figure 6.4. By adding

decisions, this tree becomes more complex.

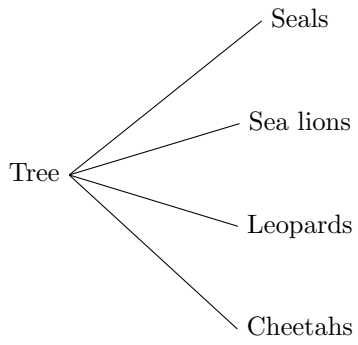


Figure 6.4: Tree for a standard neural network.

Each of these neural networks is trained like the standard neural network, using inferred labels from the existing labelled data. For example, data with the label Seal would have an output of Pinnipeds in the tree neural network in Figure 6.2, a label of Seal in the Pinnnipeds neural network, and an empty label in the Cats neural network.

6.3 Testing the Method

To test the branching neural network it is used as an SED system in DCASE 2017 task 4. In DCASE 2017 task 4, a baseline method to compare to is given and the winning method has published their code on github. The DCASE 2017 task 4 dataset consists of 17 classes: screaming, bus, motorcycle, car, car alarm, car passing by, ambulance (siren), "fire engine, fire truck (siren)", police car (siren), civil defense siren, truck, "air horn, truck horn", reversing beeps, train, train horn, bicycle, and skateboard.

These things make this task a good dataset to test the branching neural network, as we have multiple state of the art SED systems to compare it to and to use as internal decision systems. To test branching neural networks, multiple subsets from this dataset were used, as can be seen in the final comparison in section 8. To show each consecutive variant of the branching neural networks, however, only one subset will be used, as to illuminate the results of each step of the step-by-step process, without making the comparisons more complex than they need to be. This subset consists of 4 classes: "Air horn, truck horn", Train horn, Ambulance (siren) and Police car (siren). The tree for this subset is given in Figure 6.5. Hereafter this shall be referred to as the HornSiren dataset. This subset was chosen because it keeps the branching neural network to its core: 4 classes, split into 2 groups that are easily split by humans. The HornSiren dataset consists of 2 partitions. The training set, with which the neural network

and internal neural networks for the branching neural network are trained as explained in chapter 4, and the evaluation set, a strongly labelled dataset with which the F1 score is calculated as discussed in chapter 2. The HornSiren dataset training set is a balanced dataset. This means that for each class there exist the same amount of labelled audiofiles as for each other class. For the HornSiren training set there are 313 files for each class. These files are picked from the larger trainingset given by the DCASE2017 task 4 dataset. For each class the first 313 files that contain the label, sorted alphabetically, are used from the larger training set. The evaluation set is not a balanced dataset. Instead as much data from the DCASE2017 task 4 evaluation set was used by using all files containing the classes: "Air horn, truck horn", Train horn, Ambulance (siren) or Police car (siren). This was done for all subsets shown in this thesis. For the hornsiren dataset this evaluation set contained 46 "Air horn, truck horn" files, 49 Train horn files, 33 Ambulance (siren) files and 36 Police car (siren) files. Table 6.1 shows these numbers.

	Air horn, Truck horn	Train horn	Ambulance (siren)	Police car (siren)
Training set	313	313	313	313
Evaluation set	46	49	33	36

Table 6.1: Number of files in the HornSiren subset

For each of the Internal nodes, in this case *Tree*, *Siren* and *Horn*, the baseline method seen in section 4.4.1 is used to choose which branch to take. For this reason the only parameters changed between the baseline method and the branching method is the tree structure given, and which data to train on. This means that the features used are the same as the features used for this baseline method, and are gained by using the settings described in 4.4.1 on the audiofiles in the subset, for both the training set and evaluation set.

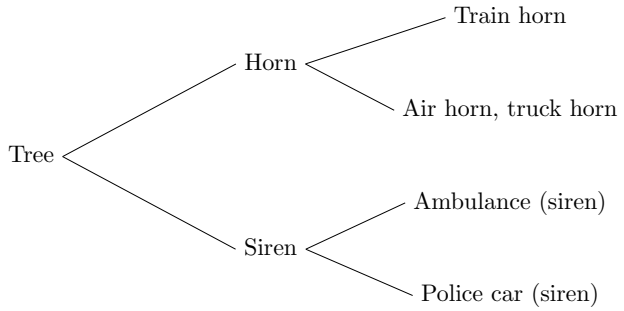


Figure 6.5: HornSiren

SED systems have 2 outputs:

- All classes present in an audio-file, called the tags.
- When exactly each of the classes is present.

Using the tags internally to determine which neural network to go to next for each of the neural networks is much simpler than using the full SED output. This is because with the tagging output we can focus on one output for each next neural network, not multiple outputs. This means we do not have to combine all of the decisions for all the SED output, as such decreasing the amount of tuning necessary. As the focus is to improve SED performance, SED results will still be shown in section 8, however to find the best branching structure all results will be tested using the F1 score of the tagging output.

6.4 From Neural Network to Decision

In a decision tree each logical decision has an output of True or False, however neural networks output a real number. To use this neural network as a decision, we must transform this real number into a True or False value. The standard way to do this in tasks that require a True or False values is to use a threshold value t. The output o of the neural network is than compared to this threshold value t in a function such that

$$d(o) = \begin{cases} \text{False} & \text{if } o < t \\ \text{True} & \text{if } o >= t \end{cases}$$

creating a decision d which has a True or False value. Because this same method is used in audio tagging, we can look at audio tagging to see how this threshold value is picked. In audio tagging there are 2 ways to pick this t value:

1. Pick the t value for which the neural network has the best accuracy on the training set
2. Pick the t value for which the neural network has the best accuracy on a evaluation set on which the neural network has not been trained.

It should be clear that training on a unseen evaluation set is the most robust method for determining this t value. The standard way to create this evaluation set is by using n-fold cross-validation, where the training set is split into n folds, and for each fold $\frac{n-1}{n}$ of the data is used in the training set and $\frac{1}{n}$ in the evaluation set. Afterwards the best t value over all these n folds is used, after which the neural network is retrained on the full training set. However as this increases training time by about n times, the easier, albeit less accurate, method using the training set was used instead. In this method the most accurate t value on the training set is used. Another way to create the evaluation set is by picking a random subset of the data for the evaluation set. However because this adds extra randomness into already complicated procedure of evaluation a new method, this random evaluation set method was not chosen to determine the threshold values.

6.5 How to Divide the Dataset

Now that we can create decisions from neural networks and we have a tree structure for the decision tree, we need only determine how to train the neural networks. Although for each neural network we have a set of labels for each dataset, this does not mean we need to use all data in each decision. In fact there are multiple ways to filter the data based on the decision.

Only correct data We could only use the correct data for each decision. For example we could use only the data with the label *Police car (siren)* or an *Ambulance (siren)* in the *Siren* decision. This greatly decreases training times, as we use less data in the decisions as we go deeper into the network. However, intuitively this would perform worse on badly labelled data in previous decisions, as we never train on data that ends up in the wrong decision.

All data We could use all data for each decision. For example if we have data with the *Train horn* label, we would label it as empty for the *Siren* neural network. Although this would increase performance of data that ends up in the wrong decisions, training on all data will increase the training time relative to the other methods.

Only important data As a middle ground, we could use earlier decisions to decide which data ends up where. For example data with the label *Train horn*, would be trained like normal in the *Tree* and *Horn* neural networks. However, it would be used in the training of the *Siren neural* network, if the *Tree* neural network would label the data as *Siren*. In this way we would decrease training time, but also keep the performance up by also training on the data that is most likely to end up in each network.

The basic branching neural network has been tested on all 3 of these training methods on the HornSiren dataset. As a comparison, the baseline neural network from which the internal neural networks were made is also tested. The test consists of running all methods 10 times and plotting their mean and standard deviation in F1 score for their tagging results on a F1/threshold plot. The lines in the figure represent the mean F1 score over the 10 runs, and the area around each line, in its respective color, represents a standard deviation for the 10 runs. In other words, the shaded area represents where the actual mean is most likely to be.

The table shows the F1 mean and F1 standard deviation for each method, followed by a p-value when compared to the baseline method, for thresholds 0.10, 0.20, ..., 0.90. The p-value in the table is used to determine if the difference between 2 methods is significant. This value is calculated by performing a Mann-Whitney U test (McKnight & Najab, 2010), a statistical test between 2 random independent samples that can be used on non-normally distributed data. After all, it is not a given that the data is normally distributed. All test were performed using a single-sided test, when the mean is *lower,given in red*,

the test was done to check if the mean was **significantly lower**, given in a bold red. When the mean was *higher*, given in blue the test was done to check if the mean was **significantly higher**, given in bold blue. The difference was considered significant if $p < 0.05$ (or $p < 5\%$), meaning that there is less than a 5% chance that they belong to a distribution with the same mean given the results. In this table, and all following tables, the highest F1 mean value of all methods is **bolded** for each threshold value.

As can be seen in Figure 6.6 and its accompanying table the Correct Data, All Data and Important Data branching methods have very different properties:

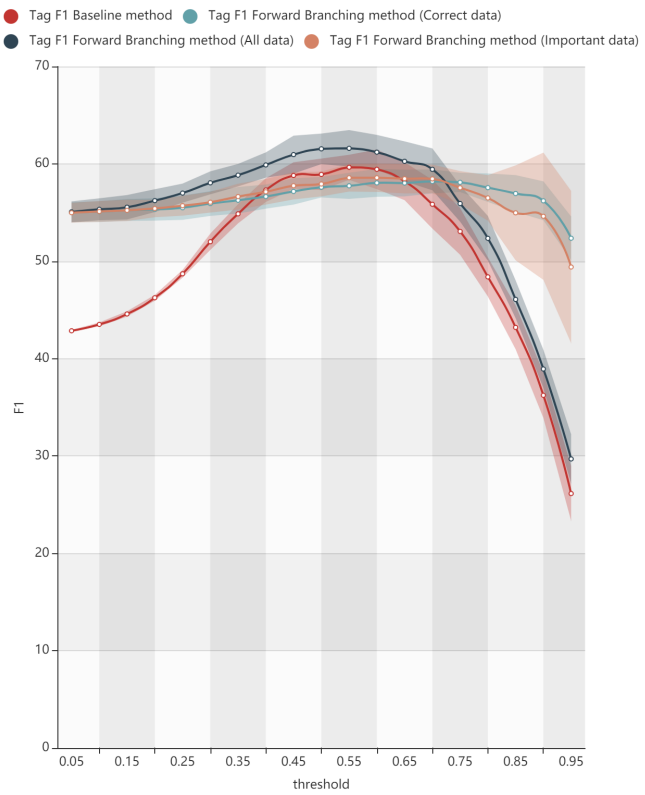
Shown in the figure is that Training with all data results in a shape most similar to the baseline method, but with a higher F1 score mean for all thresholds.

training with only correct results in a shape most similar to training with important data, however training with important data has a higher standard deviation. This shows that although this method has great potential for success, it also has a high risk of failure.

Shown in the table is that if we want the method least influenced by the threshold, the best choice would be training with only correct data, as the F1 score mean of Correct Data changes the least over the entire threshold range while maintaining a low standard deviation. Similarly, if we want a method that works best on high thresholds, which equates to high confidence ratings, the correct data method would be most preferred. However, if we wish to have the highest F1 score for some threshold, the method that uses all the data wins out.

Some general conclusions we can draw from this data is that:

- Training on all data indicates potential for success.
- All methods are more robust than the baseline method: less sensitive to picking an incorrect threshold. Because for all methods the lower threshold values have significantly better results than the baseline method.



Threshold	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
Baseline F1 mean	43.5	46.3	52.0	57.3	58.9	59.4	55.8	48.4	36.2
Correct Data F1 mean	55.1	55.3	55.9	56.6	57.6	58.0	58.2	**57.6**	**56.2**
All Data F1 mean	**55.3**	**56.2**	**58.1**	**59.9**	**61.5**	**61.2**	**59.4**	52.4	38.9
Important Data F1 mean	55.1	55.4	56.1	57.1	57.9	58.6	58.5	56.5	54.6
Baseline F1 Standard Deviation	0.2	0.3	0.8	1.2	1.6	2.0	2.5	2.0	2.3
Correct Data F1 Standard Deviation	1.1	1.1	1.3	1.2	1.0	1.4	1.2	1.5	2.0
All Data F1 Standard Deviation	1.1	1.2	1.2	1.3	1.6	1.8	2.1	2.3	2.0
Important Data F1 Standard Deviation	1.1	0.9	1.1	1.3	1.3	1.5	1.5	2.3	6.5
Correct Data p-value	**0.0%**	**0.0%**	**0.0%**	*10.6%*	**3.2%**	*8.1%*	**1.9%**	**0.0%**	**0.0%**
All Data p-value	**0.0%**	**0.0%**	**0.0%**	**0.0%**	**0.1%**	**3.8%**	**0.5%**	**0.2%**	**1.3%**
Important Data p-value	**0.0%**	**0.0%**	**0.0%**	*21.4%*	*14.5%*	*19.2%*	**1.3%**	**0.0%**	**0.1%**

Figure 6.6: Tagging F1 graph for HornSiren: Baseline vs Correct Data, All Data and Important Data Branching methods

Chapter 7

Backward Propagation in a Branching Neural Network

All 3 variants of the basic branching neural network can be seen as forward propagating methods. First the trunk of the tree is trained after which the deeper layers are trained, based on the output of the previous layers. This is similar to how a decision tree is formed. However, neural networks usually work the other way around. Because the network shape is set in advance, similar to our tree, back-propagation can used instead. This would entail changing shallow layers based on how wrong the deeper layers were. If we could do this for our branching neural network, we would in effect be creating a deeper neural network. It is known that deeper neural networks can do more complex things than shallow neural networks, although they have their own downsides. As such if we want to mimic a deeper neural network with the branching method we will need to look into these downsides and the math behind a neural network more closely.

7.1 back-propagation in Neural Networks

The most popular method to train a neural network, back-propagation, works by calculating the error gradient of the weights and then adjusting the weights based on this gradient, nudging the neural network towards a better set of weights. Take y_n to be the output vector at layer n, then the formula for y_n is given by:

$$y_n = \begin{cases} x & \text{if } n = 0 \\ f_n(w_{n-1}y_{n-1}) & \text{if } n > 0 \end{cases}$$

where x is the input, f_n is the set of functions for the neurons in layer n, and w_n is the set of weights for layer n. For a network with N layers y_N is thus the output. Assume $\hat{y_x}$ is the correct output for input x. Then the error of the network is defined by $e = \frac{1}{2}(y_N - \hat{y_x})^2$. This means that the error gradient,

d_{N-1}, of weights $N-1$ is given as $d_{N-1} = \frac{\delta e}{\delta w_{N-1}}$. We can then give this formula as a function of the outputs of the neural network by simplifying d_{N-1}:

$$\begin{aligned}
d_{N-1} &= \frac{\delta e}{\delta w_{N-1}} \\
&= \frac{\delta}{\delta w_{N-1}} \frac{1}{2}(y_N - \hat{y_x})^2 \\
&= \frac{\delta}{\delta w_{N-1}} \frac{1}{2}(f_N(w_{N-1}y_{N-1}) - \hat{y_x})^2 \\
&= (f_N(w_{N-1}y_{N-1}) - \hat{y_x}) \frac{\delta f_N}{\delta w_{N-1}} \\
&= (y_N - \hat{y_x}) y_{N-1} f'_N
\end{aligned}$$

For weights $N-2$, d_{N-2} is given by $d_{N-2} = \frac{\delta e}{\delta w_{N-2}}$. This can, similar to d_{N-1} then be given as a function of the outputs of the neural network:

$$\begin{aligned}
d_{N-2} &= \frac{\delta e}{\delta w_{N-2}} \\
&= \frac{\delta}{\delta w_{N-2}} \frac{1}{2}(y_N - \hat{y_x})^2 \\
&= \frac{\delta}{\delta w_{N-2}} \frac{1}{2}(f_N(w_{N-1}y_{N-1}) - \hat{y_x})^2 \\
&= (f_N(w_{N-1}y_{N-1}) - \hat{y_x}) \frac{\delta f_N(w_{N-1}f_{N-1}(w_{N-2}y_{N-2}))}{\delta w_{N-1}} \\
&= (f_N(w_{N-1}y_{N-1}) - \hat{y_x}) \frac{\delta f_N}{\delta f_{N-1}} \frac{\delta f_{N-1}}{\delta w_{N-2}} \\
&= (y_N - \hat{y_x}) f'_N w_{N-1} y_{N-2} f'_{N-1} \\
&= d_{N-1} \frac{y_{N-2}}{y_{N-1}} w_{N-1} f'_{N-1}
\end{aligned}$$

or in general, with the assumption that $d_{M-1} = d_{M-1} \frac{y_{M-2}}{y_{M-1}} w_{M-1} f'_{M-1}$ holds for all weights $M > n$, with induction we can show that for all $0 < n < N-1$:

$$\begin{aligned}
d_n &= d_{n+1} \frac{y_n}{y_{n+1}} w_{n+1} f'_{n+1} \\
&= (y_N - \hat{y_x}) y_N \prod_{m=n}^{N-1} \frac{y_m}{y_{m+1}} w_{m+1} f'_{m+1} \\
&= (y_N - \hat{y_x}) y_N \frac{y_n}{y_N} \prod_{m=n}^{N-1} w_{m+1} f'_{m+1} \\
&= (y_N - \hat{y_x}) y_n \prod_{m=n+1}^{N} w_m f'_m
\end{aligned}$$

This means that we can turn branching neural networks into a deeper neural network by having a function that connects each of the layers. A connecting

function however, does exist for branching neural networks, the decision function $f(x)$:

$$f(x) = \begin{cases} 0 & \text{if } x < t \\ 1 & \text{if } x >= t \end{cases} \tag{7.1}$$

where 0 represents False and 1 represents True. However, as we can see in the back-propagation formula, this function also needs a derivative. If we look at the derivative of our decision function we get:

$$f'(x) = \begin{cases} 0 & \text{if } x < t \\ \text{undefined} & \text{if } x = t \\ 0 & \text{if } x > t \end{cases} \tag{7.2}$$

Which means our derivative is partially undefined and as such we can not use it for backwards propagation. We could change this function into a function that has a derivative, such as the σ function, as is standard in a neural network. However, this would remove the decision and create instead a standard deeper neural network.

The reasons not to use this a different can be seen by looking at the 2 big problems a standard deeper neural network has.

Vanishing Gradient Problem When calculating the gradients used to adjust the neural network, if these gradients are lower than one, the gradients tend to go towards zero as we go deeper in the neural network. This is because these gradients are multiplied with each other. With a gradient of zero, the network does not adjust and therefore learns nothing. This causes deeper layers to not adjust, making it so the entire network does worse, as there exist layers that turn useful data into useless data.

Exploding Gradient Problem When calculating the gradients used to adjust the neural network, if these gradients are higher than one, the gradients tend to go towards infinity as we go deeper in the neural network. This is because these gradients are multiplied with each other. With a gradient of infinity, the network adjusts completely every time an adjustment takes place. This makes it so that deeper layers will not converge to a useful state, as such there exist layers that turn useful data into useless data.

The origin of these problems can be seen by looking at d_n.

$$\lim_{N>\infty} d_0 = \lim_{N>\infty} (y_N - \hat{y_x}) y_0 \prod_{m=1}^{N} w_m f'_m$$

so if $w_n f'_n < 1$ for multiple n d_0 goes towards 0, called a vanishing gradient. If $w_n f'_n > 1$ for multiple n d_0 goes towards ∞, called an exploding gradient.

If we look again at our decision function, function 7.1, we can see that an exploding gradient would not be an issue with enough decision functions,

because our derivative, function 7.2, is always 0 where it is defined. For this same reason, for the decision function, the vanishing gradient problem would always be a problem. Normally this would mean that backward propagation would be impossible, however if we create a step inspired by backward propagation that would solve the vanishing gradient problem, we will have created a deeper neural network without its 2 biggest problems by using these new decision layers at the correct depths.

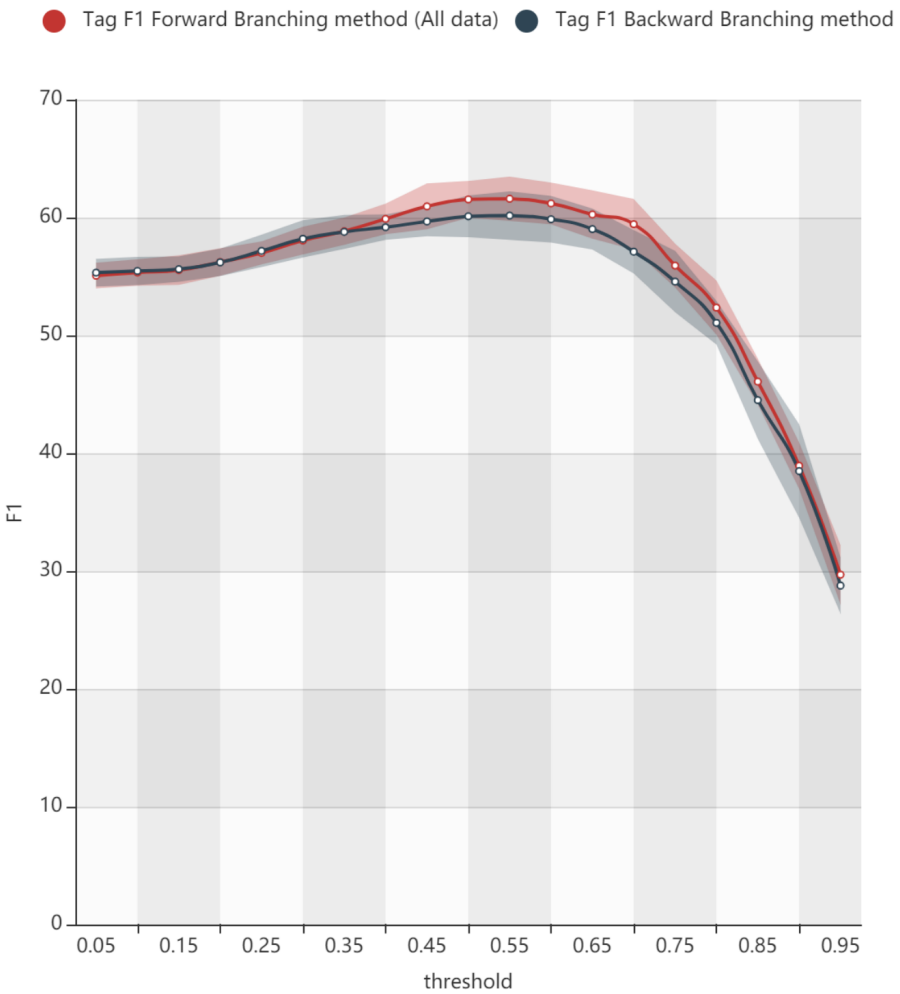
7.2 Propagating Loss

In the basic branching neural network we trained each of the neural networks separately. However, this solution does not keep in mind backward propagation, or vanishing gradients, a big remaining issue with this method as this could lead to networks that turn useful data into useless data. If however we adjust the method to take a gradient from the later decisions in such a way that we can train the earlier decisions in the tree, we will have a solution that mimics backward propagation and mimics a deeper neural network without vanishing gradients. The reason this solution would not be the same is that we would need a connected network with a fully defined derivative to actually perform backward propagation. A fully defined derivative function however, would make it so the method is no longer a branching neural network, but instead a deeper neural network, as seen in section 7.1.

To determine the gradient for our mimicking of backward propagation we can take a closer look at the decision function. The error of a the decision function can be calculated by looking at what will happen in the next networks. In the case the next neural network has the correct output, the decision was correct, and its error would be 0. If however the neural network has the incorrect output, the error equals this output. As such we now have a loss l of $l = max(\text{wrong output})$. The way to represent this loss is by having a separate output neuron in the neural network that trains with this loss value as its actual value, in this way adjusting all the weights in the neural network during training. As the loss of a decision is determined by the output of the next neural networks and later decisions, we will need to train the later decisions first, before training earlier decisions in the tree. This extra loss however also removes the option of only training on important data or correct data in each network, as all data is now important to determine this loss. However, if this could increase the performance of the system as a whole, by mimicking a deeper neural network, without the vanishing gradient and exploding gradient problems, this might be worth this loss of options.

The results on the HornSiren subset are shown in Figure 7.1 and its accompanying table. These tables contain the same type of information as the tables seen in section 6.5, however the p-value here is calculated between the forward and backward method. Just as in this section, the values are all for the tagging output of the SED systems. These tables show that no significant difference between the forward and backward propagation was found on most thresholds

values, as only $t = 070$ has a p-value of $p < 5\%$, on this $t = 0.70$, however, backward propagation did significantly worse.



Threshold	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
Forward F1 mean	55.3	**56.2**	58.1	**59.9**	**61.5**	**61.2**	**59.4**	**52.4**	**38.9**
Backward F1 mean	**55.5**	**56.2**	**58.2**	59.2	60.1	59.9	57.1	51.1	38.5
Forward F1 Standard Deviation	1.1	1.2	1.2	1.3	1.6	1.8	2.1	2.3	2.0
Backward F1 Standard Deviation	1.2	1.2	1.6	1.1	1.8	2.0	1.8	1.9	4.0
p-value	*48.5%*	*54.5%*	*39.6%*	*15.4%*	*7.0%*	*8.1%*	**1.3%**	*10.6%*	*41.0%*

Figure 7.1: Tagging F1 graph for HornSiren: Forward vs Backward branching method

7.3 Different Loss Functions

One reason for the absence of significant differences between the backward and forward propagation could be the loss function used to train the neural network. Because we want to propagate loss from one network to another, this loss function plays a great role, as it determines how the loss is calculated and used. For that reason we need to look not only at the loss function currently used internally for the baseline, but also why this loss function is used and if it makes sense to use it for the perpetuation of our new loss.

The loss function function currently used in the baseline implementation is *Binary Cross-entropy*:

$$L(y, \hat{y}) = \begin{cases} -\log(y) & \text{if } \hat{y} = 1 \\ -\log(1-y) & \text{else} \end{cases}$$

Where y is the output of a neural network and $\hat{y}$ the true observation. As the name implies this loss function is designed for when the true observation is either 0 or 1. As the loss gained by backward propagation is the output of a neural network, this value is a real number. As such it might make sense to change the loss function for a function that is designed for real numbers, not binary values. The lack of a significant difference might then be explained by this loss function. Therefore we should test different loss functions, to see if changing these loss functions change our results. To do this we need to analyze different loss functions to find the most theoretically valid loss functions. The most popular loss functions are (Changhau, 2017):

Mean Squared Error (MSE)

$$L(y, \hat{y}) = \frac{1}{N} \sum_{n=1}^{N} (\hat{y_n} - y_n)^2$$

The simplest loss function. This loss function is for real values, as such this loss function could be suitable for our loss propagation. One big problem this loss function suffers from is slow convergence speed when used in conjuncture with a sigmoid activation function. As the last layer of the baseline method uses this sigmoid activation function, MSE will most likely suffer from this problem in the baseline implementation.

Mean Squared Logarithmic Error (MSLE)

$$L(y, \hat{y}) = \frac{1}{N} \sum_{n=1}^{N} (\log(\hat{y_n} + 1) - \log(y_n + 1))^2$$

A varient of MSE that is designed to work with larger numbers. As the expected loss values for our purposes are likely to be between 0 and 1, MSE is more likely to be useful than MSLE.

Mean Absolute Error (MAE)

$$L(y, \hat{y}) = \frac{1}{N} \sum_{n=1}^{N} |\hat{y_n} - y_n|$$

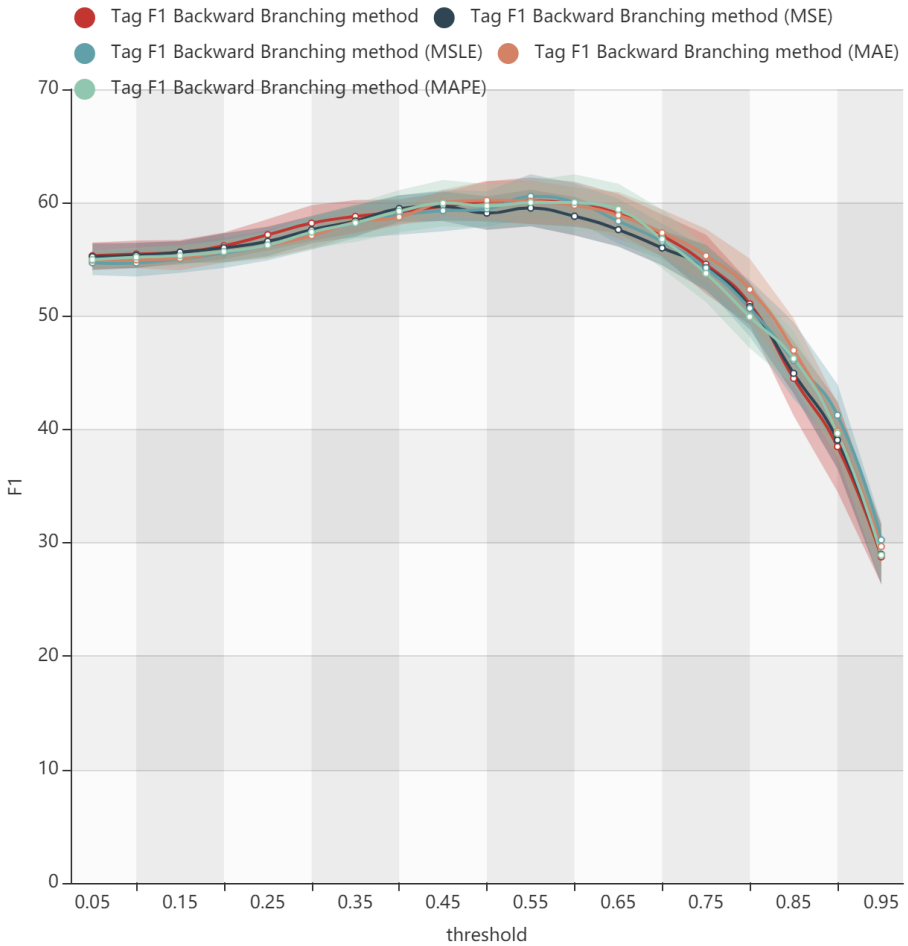
A variant on MSE that is less influenced by outliers in the data because of its linear nature. This however, also means it is less influenced by large errors. As such this loss function is most useful when more concerned with noisy data than with large errors. As our loss can be exactly calculated, we are less concerned with noisy data for our purposes, so most likely the MSE will perform best.

Mean Absolute Percentage Error (MAPE)

$$L(y, \hat{y}) = \frac{1}{N} \sum_{n=1}^{N} \left| \frac{\hat{y_n} - y_n}{\hat{y_n}} \right| \cdot 100$$

A variant on MAE which uses the ratio of the difference in percentages. This loss function works well with many different classifications because of its relative nature, however in cases where $\hat{y} = 0$ its value is undefined. As this is the case for a big part of our dataset, because $\hat{y} = 0$ whenever we send data to a correct node, this loss function is most likely inappropriate for our use case, if we do not adjust this definition.

There exist more loss functions, like the Kullback Leibler Divergence, Poisson and Cosine proximity loss functions, however, as these are not meant for other purposes than multiple real-valued inputs, they will not be investigated. MSE, MSLE, MAE and MAPE are plotted in Figure 7.2 against using the default *Binary Cross-entropy (BCE)*. This figure and its accompanying table show that no significant positive difference is found between the different loss functions. This can not only be seen by the fact that for no p-value $p < 5\%$, but can also be seen in the Figure by the fact that all lines and shaded areas overlap, making it impossible to tell which method is which. Therefore we most conclude that changing loss function will not create a significant difference between backward and forward propagation and the default will be used henceforth.



Threshold	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
BCE (Original)	**55.5**	**56.2**	**58.2**	59.2	60.1	59.9	57.1	51.1	38.5
MSE	55.4	56.0	57.6	**59.5**	59.1	58.8	56.0	50.8	39.1
MSLE	54.7	55.6	57.3	58.9	59.5	**60.0**	56.6	50.7	**41.3**
MAE	55.0	55.7	57.1	58.7	**60.2**	59.8	**57.3**	**52.3**	39.8
MAPE	55.2	55.7	57.4	59.2	59.7	**60.0**	56.8	49.9	39.7
BCE Standard Deviation	1.2	1.2	1.6	1.1	1.8	2.0	1.8	1.9	4.0
MSE Standard Deviation	1.1	1.3	1.2	1.2	1.5	1.7	1.5	2.0	2.5
MSLE Standard Deviation	1.2	1.3	1.4	1.7	1.6	1.6	1.5	2.5	2.7
MAE Standard Deviation	0.7	1.0	1.1	0.7	1.7	1.6	2.1	2.7	2.5
MAPE Standard Deviation	0.8	1.0	1.4	1.9	1.9	2.5	2.6	2.8	2.1
MSE p-value	*42.5%*	*33.9%*	*20.3%*	*31.2%*	*17.2%*	*10.6%*	*15.3%*	*50.0%*	*33.9%*
MSLE p-value	*12.1%*	*13.7%*	*12.1%*	*44.0%*	*36.7%*	*54.5%*	*28.5%*	*44.0%*	*9.9%*
MAE p-value	*23.6%*	*10.6%*	*5.6%*	*13.7%*	*33.9%*	*50.0%*	*26.0%*	*17.2%*	*28.5%*
MAPE p-value	*33.9%*	*15.4%*	*12.1%*	*68.9%*	*39.6%*	*45.5%*	*51.5%*	*12.0%*	*31.2%*

Figure 7.2: Tagging F1 graph for HornSiren: Different Loss functions in Backward propagation

7.4 Additional Output

The backward branching neural network is now trained in such a way that each network has an output that shows how wrong the next network is likely to be: the output that is trained to propagate our loss from the next network. As such we have extra data that could tell us which network is best to go to. Therefore we can experiment with this additional data to find how to best use it. Could it be that this extra input changes the backward method to be significantly better than the forward method? This question could be answered by changing the output in such a way that only when the label was originally given and the network doesn't think the next step will go wrong, the label will now be given. As such the following method were devised to use this new data, where $\hat{o}$ is our new output, o the original output of the network, and p this data symbolizing how wrong the network is:

Multiplication

$$\hat{o_i} = o_i \cdot (1 - p_i)$$

Subtraction

$$\hat{o_i} = o_i - p_i$$

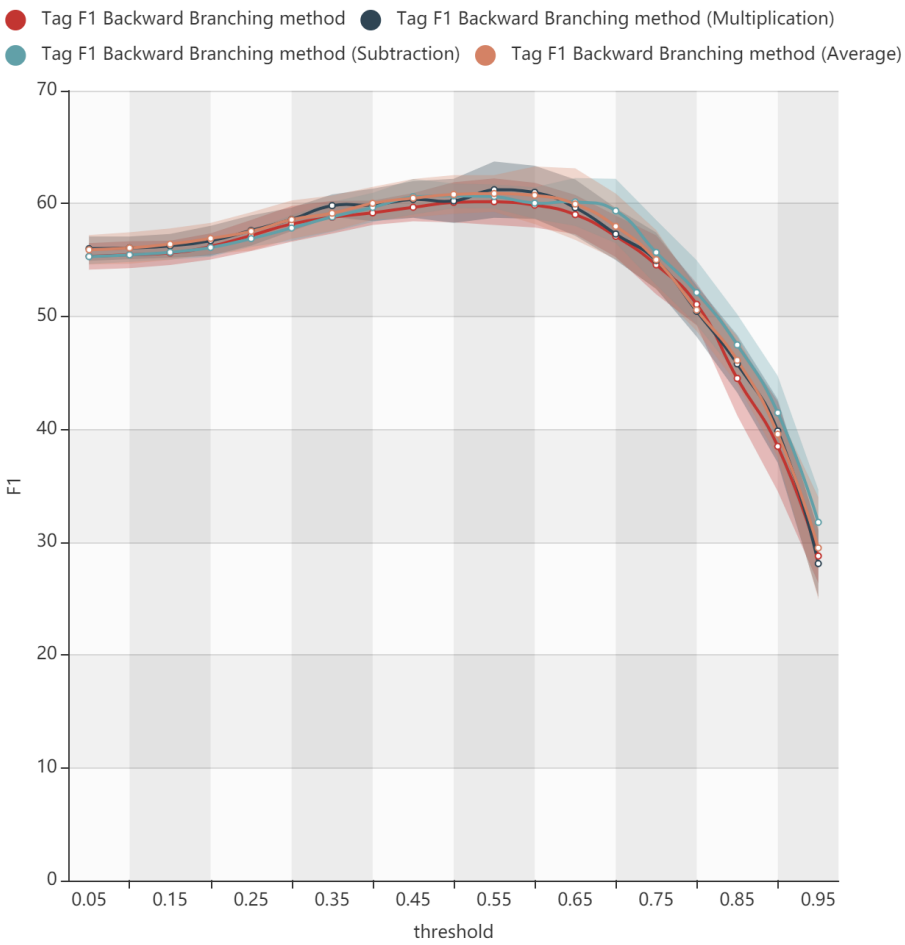
Average

$$\hat{o_i} = \frac{o_i + (1 - p_i)}{2}$$

Each of these functions are a different way of interpreting how o_i and p_i interact. For all it is the case that the highest output would be when $p_i = 0$ and $o_i = 1$ and the lowest output when $p_i = 1$ and $o_i = 0$, however, for all thee functions the values in between follow a slightly different pattern. Multiplication weights high p_i values more heavily than subtraction and average is a compromise between multiplication and subtraction.

The results of these different output functions can be seen in Figure 7.3 and its accompanying table. On most thresholds no significant difference is found between using an output function or not using an output function, as can be seen from the p-values. However for Subtraction and Average there is a thresholds that does significantly better, $t = 0.70$, and that thresholds overlaps with the thresholds where backwards propagation did worse than forward propagation. As Subtraction showed the most significant change at $t = 0.70$, since there the p-value is the lowest, this function will be used in further results.

It seems as if the backwards branching method is not any better than the forward branching method at this point, as we have not been able to use the extra information in a way to increase the performance of the backwards branching method on all thresholds, as we have not been able to create a method with significant p-values over all thresholds when compared to the original.



Threshold	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
Original	55.5	56.2	58.2	59.2	60.1	59.9	57.1	51.1	38.5
Multiplication	**56.1**	56.7	**58.6**	59.9	**60.2**	**61.0**	57.3	50.4	39.8
Subtraction	55.5	56.1	57.8	59.7	60.7	60.0	**59.4**	**52.1**	**41.5**
Average	**56.1**	**56.9**	**58.6**	**60.0**	**60.8**	60.7	58.0	50.6	39.6
Original Standard Deviation	1.2	1.2	1.6	1.1	1.8	2.0	1.8	1.9	4.0
Multiplication Standard Deviation	1.0	1.3	1.0	1.4	2.0	2.3	2.3	2.3	2.8
Subtraction Standard Deviation	0.6	0.8	1.1	1.3	1.1	1.4	2.8	2.9	3.3
Average Standard Deviation	1.4	1.4	1.7	1.5	1.7	2.5	2.9	1.9	2.0
Multiplication Diff	0.6	0.5	0.4	0.7	0.1	1.1	0.2	-0.6	1.4
Subtraction Diff	-0.0	-0.1	-0.4	0.5	0.6	0.2	2.3	1.0	3.0
Average Diff	0.6	0.7	0.4	0.8	0.7	0.9	0.9	-0.5	1.1
Multiplication p-value	*17.2%*	*28.5%*	*42.5%*	*17.2%*	*44.0%*	*22.5%*	*56.0%*	*28.5%*	*26.0%*
Subtraction p-value	*36.7%*	*23.6%*	*31.2%*	*23.6%*	*15.4%*	*51.5%*	**2.1%**	*17.2%*	*7.0%*
Average p-value	*12.0%*	*17.2%*	*32.5%*	*7.0%*	*18.2%*	*23.6%*	**4.4%**	*42.5%*	*35.3%*

Figure 7.3: Tagging F1 graph for HornSiren: Different output functions in Backward propagation

Chapter 8

Testing on Multiple Subsets

To test how successful the forward and backward versions of branching neural networks are, they have been tested on multiple subsets of the DCASE 2017 task 4 dataset. This dataset consists of 17 classes: screaming, bus, motorcycle, car, car alarm, car passing by, ambulance (siren), "fire engine, fire truck (siren)", police car (siren), civil defense siren, truck, "air horn, truck horn", reversing beeps, train, train horn, bicycle, and skateboard. This number of classes means many different trees can be created, an example of a tree created by me can be seen in Figure 5.1. Many different possible trees means that is hard to make sure that the specific tree is the optimal tree, and therefore if the branching neural network method is useful or not, as we do not know which part of the tree is causing which properties. As such subsets where created to test the branching neural network method. These subsets make creating perfect subsets of a tree more simple, as there are less possible different iterations of the tree. The subgroups are:

CarHorn This subset consists of the classes: Car, Truck, Air horn & truck horn and Train horn. The tree of this subset is given in Figure 8.1a. For the training set the classes were balanced by using 313 files for each class.

TruckSiren This subset consists of the classes: Bus, Truck, Ambulance (siren), and Police car (siren). The tree of this subset is given in Figure 8.1c. For training the classes were balanced by using 524 files for each class.

HornSiren This subset consists of the classes: Ambulance (siren), Police car (siren), Air horn & truck horn and Train horn. The tree of this subset is given in Figure 8.1b. For the training set the classes were balanced by using 313 files for each class.

Shuffle This subset is the same as HornSiren, but has its tree shuffled. This allows us to test the importance of the tree itself. This tree is given in Figure 8.1d.

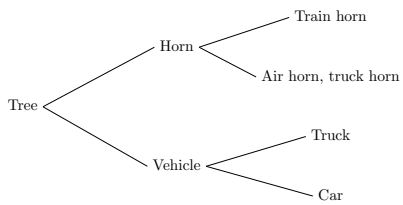
Deep Subset This subset consists of the classes: Bicycle, Skateboard, Bus, Truck, Air horn & truck horn, Train horn, Ambulance (siren), Police car (siren). This subset consists of 3 layers, instead of 2. This is to see if the results scale with deeper levels. The tree of this subset is given in Figure 8.1e. For the training set the classes were balanced by using 313 files for each class.

These subgroups where created by using the Google Audioset ontology explained by Gemmeke et al. (2017). This ontology has a tree shape consisting of 632 audio events grouped into a tree of depth 5, the first two layers of which can be seen in Figure 1.3. The subgroups where chosen in such a way that a tree of depth 2 that made intuitive sense could be made for them, except in the case of the deep subset, where a tree of depth 3 was purposefully created to test if the results would extend to deeper trees. The subsets where then taken from the DCASE2017 task 4 dataset. The training sets from the DCASE2017 task 4 training set, and the evaluation sets from the DCASE2017 task 4 evaluation set. For all training sets the subset is is the first N files, in an alphabetic order, containing one of the classes present, where N and the classes are given in the description above. For all evaluation sets the subset is gained by taking all audiofiles with the classes in the description above present from the DCASE2017 task 4 evaluation set. This means for example that the label for Bicycle is not present in any of the audio files in the training or evaluation set for HornSiren.

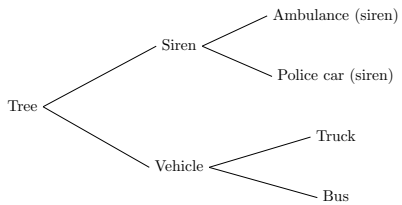
First, a comparison will be made between the tagging results of SED systems, all labels present in an audio-file, since this is the simplest output that also exemplifies most the results of branching neural networks. Afterwards we will look at the SED results, where the labels are accompanied by a start-time and a duration, because these results show if the SED output is correct even when the branching neural network internally uses a tagging system, that only looks at the labels present.

8.1 Tagging Results

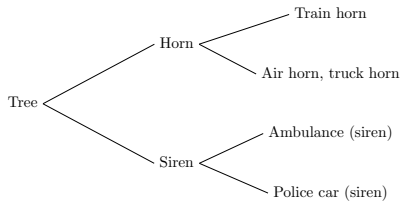
To see the effects of the branching neural networks, all plots will be F1/threshold plots, in which the F1 score is shown as a function of threshold value. For the both method the threshold value shown in the graphs and tables is used on all outputs. However for the branching method internally different thresholds values are used as the branching methods tunes these internal thresholds during training. This is done to ensure a fair comparison. For all results shown, the branching method consists of a multiple of the exact same networks as the compared method. For example, in CarHorn, the branching method consists of 3 of the same neural networks used for the baseline method. This again is to keep the comparison simple, if the neurons inside the internal subsystems were tuned for performance, it would exponentially increase the amount of variables to tune. The effect of this is that the computation necessary for training each of the branching neural networks is equal to the training necessary for training as many comparison methods as there are internal nodes. For example, for



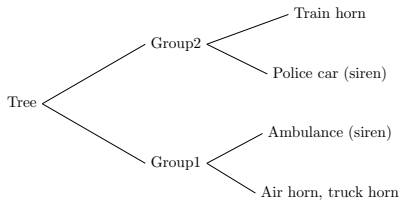
(a) CarHorn



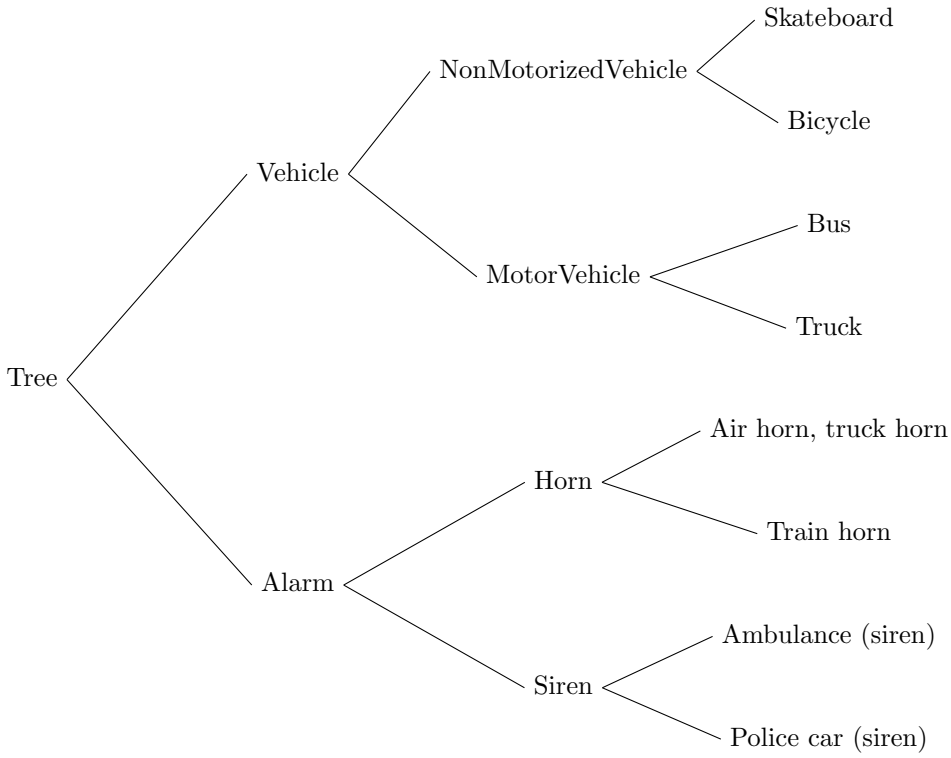
(b) TruckSiren



(c) HornSiren



(d) Shuffled version of HornSiren



(e) Deep subset

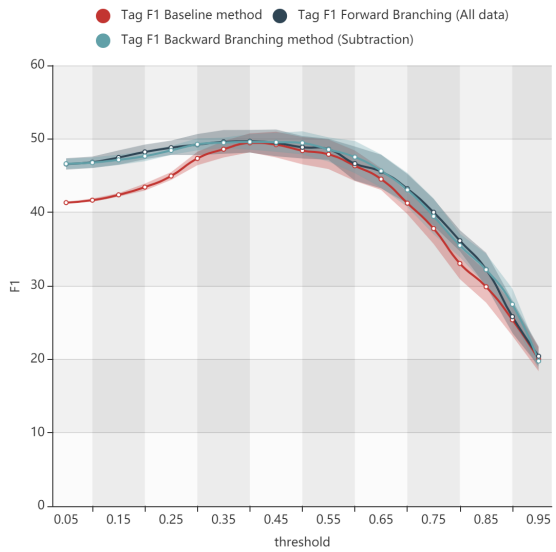
Figure 8.1: Subset trees.

the HornSiren dataset, the training computation necessary is three times the training computation necessary for the baseline method. However, for forward branching, all networks can be trained simultaneously, keeping training time similar. For backwards branching only the networks in the same layer can be trained at the same time, provided enough parallel computing power is available, meaning that the training time is multiplied by the amount of layers in the tree. For example, in the HornSiren dataset, the training time is double that of the original method. In the figures given in this chapter the branching methods will be given in blue, and the compared methods will be given in red, as per the legend of each figure. These figures will be joined by tables, representing the same values given in the figure in a more quantifyable manner.

The tables and figures in this chapter are of the same type as the tables and figures given in Chapter 6.5: The test consists of running all methods 10 times and plotting their mean and standard deviation in F1 score for their tagging results on a F1/threshold plot. This F1 value is gained by using a subset of the evaluation set given for the DCASE2017 dataset, in which the labels not present in the training subset were ignored. This means that the evaluation subset consists of all files present in the evaluation set that have a label also present in the training set, however, the files without a label present in the training set were removed. This evaluation set was used to ensure the method is not over fitting to the training set. The lines in the figure represent the mean F1 score over the 10 runs, and the area around each line, in its respective color, represents a standard deviation for the 10 runs. In other words, the shaded area represents where the actual mean is most likely to be.

The table shows the F1 mean and F1 standard deviation for each method, followed by a p-value when compared to the baseline method, for thresholds 0.10, 0.20, ..., 0.90. The p-value in the table is used to determine if the difference between 2 methods is significant. This value is calculated by performing a Mann-Whitney U test (McKnight & Najab, 2010), a statistical test between 2 random independent samples that can be used on non-normally distributed data. After all, it is not a given that the data is normally distributed. All test were performed using a single-sided test, when the mean is *lower,given in red*, the test was done to check if the mean was **significantly lower**, given in a bold red. When the mean was *higher*, given in blue the test was done to check if the mean was **significantly higher**, given in bold blue. The difference was considered significant if $p < 0.05$ (or $p < 5\%$), meaning that there is less than a 5% chance that they belong to a distribution with the same mean given the results. In this table, and all following tables, the highest F1 mean value of all methods is **bolded** for each threshold value.

8.1.1 CarHorn



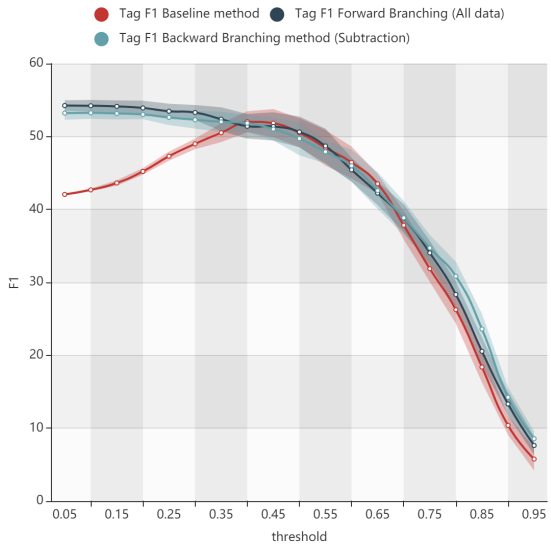
Threshold	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
Baseline F1 mean	41.7	43.4	47.3	49.5	48.4	46.3	41.3	33.0	25.4
Forward Branching F1 mean	**46.8**	**48.2**	49.2	**49.7**	48.9	46.6	**43.2**	**36.1**	25.8
Backward Branching F1 mean	**46.8**	47.6	**49.3**	49.6	**49.5**	**47.5**	43.1	35.5	**27.5**
Baseline Standard F1 Deviation	0.2	0.5	0.9	1.3	1.9	2.1	1.5	2.1	2.2
Forward Branching F1 Standard Deviation	0.8	1.0	1.4	1.5	1.5	2.3	2.0	1.4	2.2
Backward Branching F1 Standard Deviation	0.7	0.7	0.8	1.1	1.6	2.2	2.3	1.6	2.1
Forward Branching p-value	**0.0%**	**0.0%**	**0.4%**	*39.6%*	*26.0%*	*39.6%*	**3.2%**	**0.2%**	*38.1%*
Backward Branching p-value	**0.0%**	**0.0%**	**0.1%**	*23.6%*	*10.6%*	*10.6%*	**2.7%**	**0.9%**	**2.7%**

Figure 8.2: Tagging F1 graph for CarHorn: Baseline vs Branching method

Car, Truck, Air horn & truck horn and Train horn:

From Figure 8.2 and its accompanying table it can be seen that for this subset, although both branching neural network methods work similarly to the baseline method for some thresholds, for most thresholds the branching neural network methods both have significantly better F1 mean, as can be seen from the p-values where $p < 5\%$. Especially for low thresholds, $t = 0.10, 0.20$ and 0.30, both branching neural networks perform significantly better. The same can be seen at threshold values $= 0.70$ and $t = 0.80$, where both the forward and backward method have p-values $p < 5\%$. Overall, for this specific dataset the branching neural network is less sensitive to picking a correct t value than the baseline method: Not only is the highest F1-score mean for both branching neural network higher, for every threshold the F1-score mean is either significantly higher (higher F1 mean with $p < 5\%$) or similar ($p > 5\%$). This makes it so that wrongly tuning the branching neural networks has less negative consequences for the branching neural networks.

8.1.2 TruckSiren



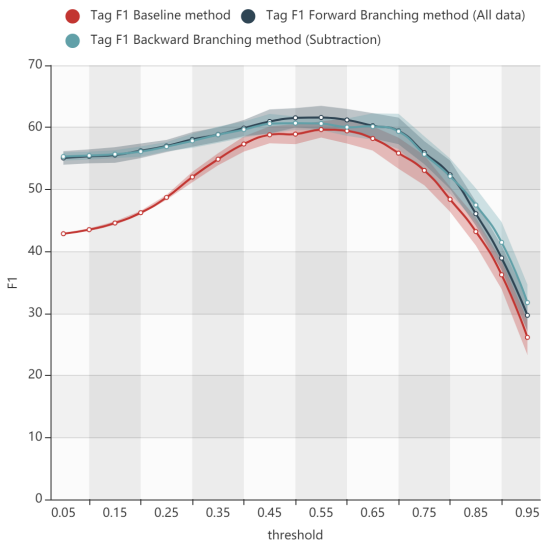
Threshold	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
Baseline F1 mean	42.7	45.2	49.0	**52.0**	50.5	**46.4**	37.8	26.3	10.4
Forward Branching F1 mean	**54.2**	**53.9**	**53.3**	51.4	**50.6**	45.4	**38.8**	28.3	13.3
Backward Branching F1 mean	53.3	53.0	52.3	51.8	49.7	46.0	**38.8**	**30.8**	**14.2**
Baseline F1 Standard Deviation	0.2	0.4	0.7	1.4	2.0	2.2	2.0	1.9	1.3
Forward Branching F1 Standard Deviation	0.8	1.0	1.0	1.7	2.1	1.6	1.9	1.8	1.6
Backward Branching F1 Standard Deviation	0.8	0.6	1.2	1.1	2.3	2.2	2.3	2.0	1.3
Forward Branching P-Value	**0.0%**	**0.0%**	**0.0%**	*23.6%*	*36.7%*	*23.6%*	*13.6%*	**2.7%**	**0.1%**
Backward Branching p-value	**0.0%**	**0.0%**	**0.0%**	*39.6%*	*19.2%*	*42.5%*	*19.2%*	**0.1%**	**0.0%**

Figure 8.3: Tagging F1 graph for TruckSiren: Baseline vs Branching method

Bus, Truck, Ambulance (siren), and Police car (siren):

Similar to CarHorn, the highest F1-score mean belongs to the forward branching method, as can be seen in Figure 8.3 and its accompanying table. Also similar to CarHorn, for lower thresholds the branching method seems to perform significantly better than the baseline method, as the p-value $p < 5\%$. As with the previous subset for high thresholds, $t = 0.80$ and $t = 0.90$ the branching neural networks both performs significantly better, although it has shifted towards slightly higher values, as it no longer the case for $t = 0.70$. One interesting thing for this dataset is that the highest F1-score mean for the branching method is on very low thresholds $t = 0.10$, bringing with it the idea that the first layer is so effective that the second layer of branching method has a negative effect on the system overall.

8.1.3 HornSiren



Threshold	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
Baseline F1 mean	43.5	46.3	52.0	57.3	58.9	59.4	55.8	48.4	36.2
Forward Branching F1 mean	**55.3**	**56.2**	**58.1**	**59.9**	**61.5**	**61.2**	**59.4**	**52.4**	38.9
Backward Branching F1 mean	**55.5**	56.1	57.8	59.7	60.7	60.0	**59.4**	52.1	**41.5**
Baseline F1 Standard Deviation	0.2	0.3	0.8	1.2	1.6	2.0	2.5	2.0	2.3
Forward Branching F1 Standard Deviation	1.1	1.2	1.2	1.3	1.6	1.8	2.1	2.3	2.0
Backward Branching F1 Standard Deviation	0.6	0.8	1.1	1.3	1.1	1.4	2.8	2.9	3.3
Forward Branching p-value	**0.0%**	**0.0%**	**0.0%**	**0.0%**	**0.1%**	**3.8%**	**0.5%**	**0.2%**	**1.3%**
Backward Branching p-value	**0.0%**	**0.0%**	**0.0%**	**0.1%**	**1.9%**	*20.3%*	**1.1%**	**0.4%**	**0.2%**

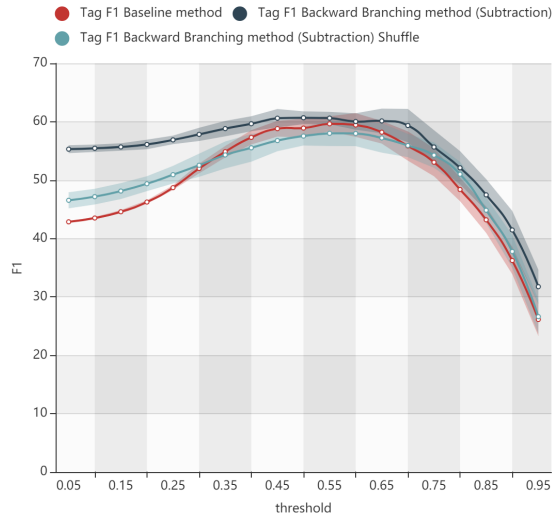
Figure 8.4: Tagging F1 graph for HornSiren: Baseline vs Branching method

Ambulance (siren), Police car (siren), Air horn & truck horn and Train horn:

Again the figure for this subset, Figure 8.4, and its accompanying table show properties seen in the previous subsets: For low threshold values, and high threshold values, the F1-score mean is significantly better (higher with $p < 5\%$) for both the forward and branching methods. The biggest difference is that the significant differences have extended from $t = 0.30$ to $t = 0.50$ as now $p < 5\%$ for both $t = 0.40$ and $t = 0.50$, with a higher F1-score mean for both methods. In this case the forward propagation method is significantly better than the baseline method at every threshold value as the F1-score mean is higher and $p < 5\%$ for every threshold. This suggest that the tree is better suited to some datasets than others, which will be further explored in section 8.1.4.

(a) Forward method

Threshold	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
Baseline F1 mean	43.5	46.3	52.0	**57.3**	**58.9**	**59.4**	55.8	48.4	36.2
Forward Shuffled F1 mean	46.9	49.3	52.3	55.2	57.6	58.4	**57.2**	**51.0**	**38.3**
Forward F1 mean	*55.3*	*56.2*	*58.1*	*59.9*	*61.5*	*61.2*	*59.4*	*52.4*	*38.9*
Backward Shuffled F1 mean	**47.2**	**49.4**	**52.5**	55.5	57.5	58.0	55.9	**51.0**	37.8
Backward F1 mean	*55.5*	*56.1*	*57.8*	*59.7*	*60.7*	*60.0*	*59.4*	*52.1*	*41.5*
Baseline F1 Standard Deviation	0.2	0.3	0.8	1.2	1.6	2.0	2.5	2.0	2.3
Forward Shuffled F1 Standard Deviation	1.1	1.2	0.8	1.3	1.8	2.1	1.7	3.0	2.3
Backward Shuffled F1 Standard Deviation	1.3	1.3	2.0	2.4	1.6	2.2	2.0	2.1	2.8
Forward Shuffled p-value	**0.0%**	**0.0%**	*23.6%*	**0.2%**	*6.1%*	*19.2%*	*9.3%*	**1.1%**	**3.8%**
Backward Shuffled p-value	**0.0%**	**0.0%**	*21.4%*	**4.4%**	**4.8%**	*9.3%*	*38.1%*	**0.9%**	*20.3%*



(a) Backward method

Figure 8.6: F1 graphs for Shuffle: Baseline vs Branching methods vs Shuffled branching

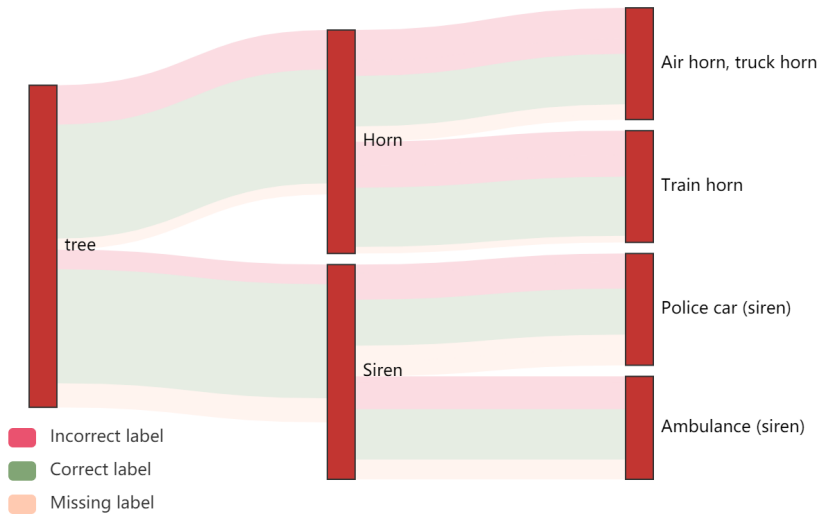
8.1.4 Shuffle

Ambulance (siren), Police car (siren), Air horn & truck horn and Train horn: Although the classes of this dataset are the same as the classes for HornSiren, the performance of the Shuffled branching method, shown in Figure 8.6 and its accompanying table, shows the importance of the tree, and the biggest weakness of the branching methods. Although for the branching methods significantly better F1-score results still show for low thresholds when compared to the baseline, they show significantly worse results than the branching methods with the tree from section 8.1.3. The shuffled method no longer outperforms the base method in highest F1 score and is significantly worse on some thresholds. The reason for this clear when analyzing Figure 8.7 and its accompanying table, Table 8.1. In these we compare the backward branching method with its shuffled variant for one specific run for a threshold value of $t = 0.50$.

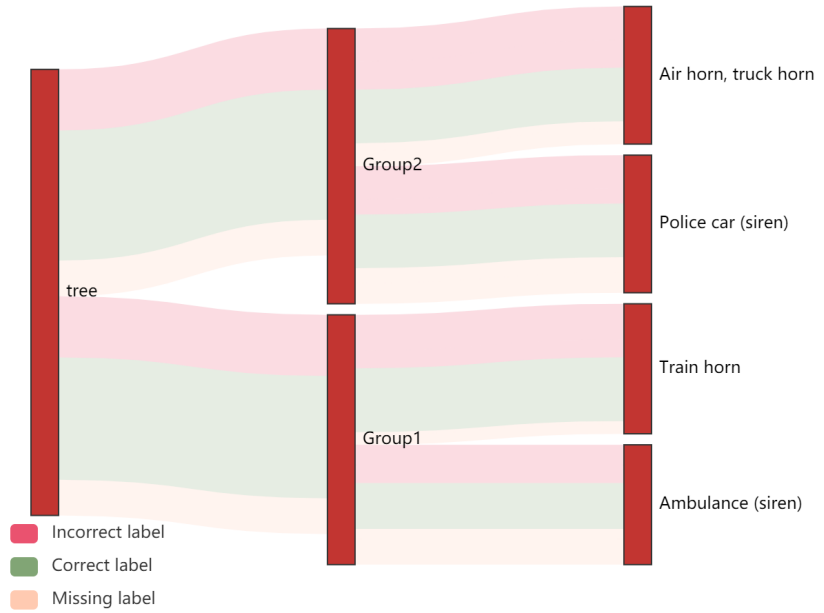
Node	F1	Precision	Recall	Shuffled F1	Shuffled Precision	Shuffled Recall
Tree	82.9	79.4	86.7	72.3	67.4	78.0 s
Horn	68.1	54.4	90.9			
Siren	67.2	58.7	78.6			
Group1				67.7	54.4	89.6
Group2				61.8	49.4	82.4
Overall	64.0	56.3	74.0	58.4	51.8	66.9

Table 8.1: Table accompanying Figure 8.7.

In Figure 8.7, green are the True Positives, red are False Positives, and yellow are False Negatives. True Negatives are not shown, as they are not important to the precision and recall calculations. Table 8.1 describes these values in a more conventional way. In this table Tree is the name given to the internal neural network that splits into the first subgrouping, while Horn, Siren, Group1 and Group 2 are the neural networks for these subgroupings. Overall shows the performance of the entire branching method for this run and threshold value. In this table we can then get the F1, precision and recall for each neural network. Tree and Overall are the only variables that both the shuffled and the non-shuffled tree contain. In the table we can see that the biggest difference between the branching method and its shuffled variant is the performance of the Tree neural network. In this neural network the 2 methods have a 14.6% relative difference between F1 scores. This difference not only makes intuitive sense, the difference between Siren and Horn is intuitively more clear then between Group1 and Group2, but also shows the reason why the shuffled variant performs worse as a whole. If we could assume the second set of networks was perfect, getting a F1 score of 100, the branching method would get a F1 score of 82.9, while the shuffled method would have an F1 score of 72.3, as such the neural networks for Group1 and Group2 would have to perform much better than the neural networks for Siren and Horn.



(a) Branching errors

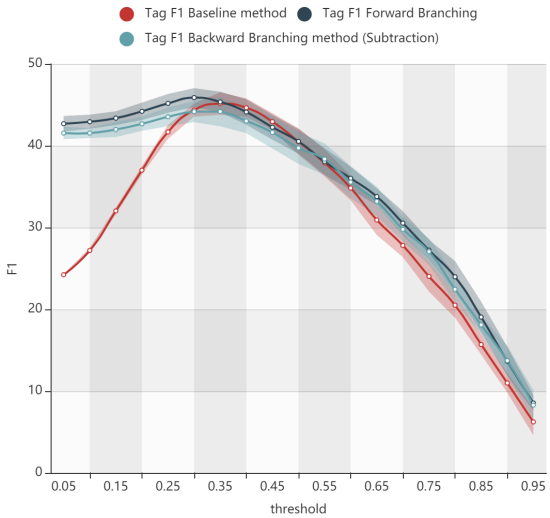


(b) Shuffled errors

Figure 8.7: Sankey plot of performance for threshold $t = 0.5$ for a single run

The table also show a big advantage of having this tree structure. The structure of the branching method allows one to use the internal classifications, i.e the ones given by the Tree neural network (the labels Siren and Horn, or Group1 and Group2). These classifications have a better accuracy then the external classifications, i.e the ones given by the Siren, Horen, Group1 and Group2 neural networks. In the case of the HornSiren tree a 29.5% relative difference is found between accuracy of the Tree neural network and the Overall accuracy. In some cases these internal classifications are very useful. For example, knowing that a sound is a siren might be enough to assess a traffic situation, even without knowing if the siren is a police or ambulance siren.

8.1.5 Deep Subset



Threshold	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
Baseline F1 mean	27.2	37.0	44.4	**44.6**	**40.5**	34.8	27.8	20.5	11.0
Forward Branching F1 mean	**42.9**	**44.2**	**45.9**	44.1	**40.5**	**36.0**	**30.5**	**24.0**	**13.7**
Backward Branching F1 mean	41.6	42.7	44.2	43.0	39.7	35.5	29.8	22.4	**13.7**
Baseline F1 Standard Deviation	0.3	0.5	0.8	1.1	1.6	1.5	1.4	1.5	1.2
Forward Branching F1 Standard Deviation	0.9	1.1	1.2	1.6	1.5	1.4	1.5	1.9	1.8
Backward Branching F1 Standard Deviation	0.6	0.9	1.3	1.5	1.9	2.0	1.5	1.6	1.8
Forward Branching p-value	**0.0%**	**0.0%**	**0.5%**	*13.7%*	*63.3%*	*6.1%*	**0.1%**	**0.1%**	**0.2%**
Backward Branching p-value	**0.0%**	**0.0%**	*39.6%*	**3.2%**	*19.2%*	*23.6%*	**1.0%**	**1.6%**	**0.2%**
Backward vs Forward p-value	**0.1%**	**0.3%**	**0.4%**	*20.3%*	*26.0%*	*28.5%*	*12.1%*	**2.5%**	*48.5%*

Figure 8.8: Tagging F1 graph for DeepSubset: Baseline vs Branching method

Bicycle, Skateboard, Bus, Truck, "Air horn, truck horn", Train horn, Ambulance (siren), Police car (siren): For this deeper subset shown in Figure 8.8 and its accompanying table, it can be seen that both branching

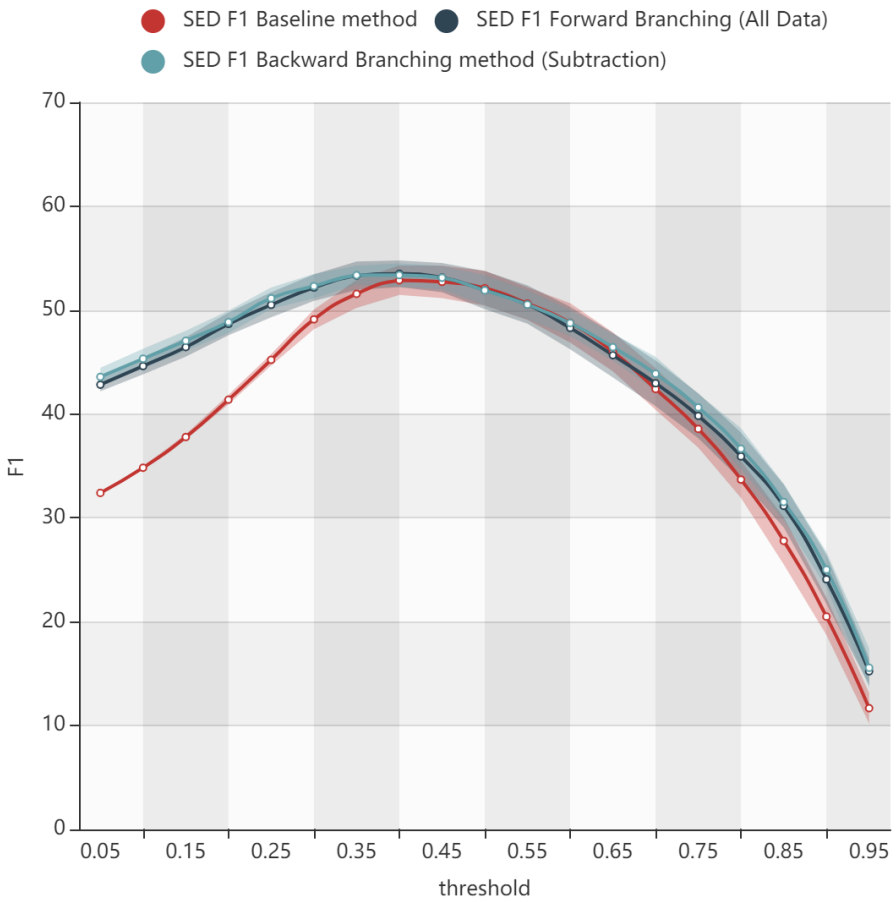
methods again significantly outperform the baseline in F1-score mean on low and high thresholds, with the biggest difference on low thresholds. In this case the backward branching neural network has a threshold value $t = 0.40$ where it performs significantly worse than the baseline method (F1-score mean is lower and $p < 5\%$). Furthermore, it seems that the backward branching method performs significantly worse at thresholds $t \leq 0.30$ and $t = 0.80$ than the forward branching method. As such it is likely that the forward branching method is the best choice for deeper subsets, or that the backwards branching method needs to be rethought.

8.2 SED Results

So far we have only focused on the tagging output of the SED systems, all labels present in an audio-file. However, as the the internal systems used in our branching neural networks are SED systems, it is important that the performance of these systems has not been compromised by our focus on tagging resuls. Therefore we will also look at the SED output of the two different branching neural network methods, where the SED output also contains the start-time and duration of each label. We will look at the results on the HornSiren subset, as these results show the biggest difference for tagging output.

8.2.1 HornSiren

Figure 8.9 and its accompanying table show that the difference between the baseline method and both branching methods become less significant for SED than for tagging, as there are now threshold values where the p-score $p > 5\%$ for both methods. This may be due to the fact that internally the system still uses tagging, not SED, to make decisions. However, similar to how both branching neural networks behave when tagging, we still see significant ($p < 5\%$) differences between the baseline and both branching methods for low and high threshold values in SED in favor of both the branching neural networks.



Threshold	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
Baseline F1 mean	34.8	41.4	49.1	52.9	**52.1**	**48.8**	42.4	33.7	20.5
Forward Branching F1 mean	44.6	48.7	52.2	**53.5**	51.9	48.3	43.0	35.9	24.1
Backward Branching F1 mean	**45.3**	**48.9**	**52.3**	53.4	51.9	48.8	**43.9**	**36.7**	**25.0**
Baseline F1 Standard Deviation	0.1	0.5	0.9	1.4	1.7	1.9	2.0	1.8	1.8
Forward Branching F1 Standard Deviation	0.8	1.1	1.3	1.3	1.8	2.0	2.2	2.3	2.4
Backward Branching F1 Standard Deviation	0.9	1.1	1.2	1.1	1.5	1.3	1.6	2.0	1.7
Forward Branching p-value	**0.0%**	**0.0%**	**0.0%**	*19.2%*	*68.8%*	*66.1%*	*7.0%*	**2.7%**	**0.5%**
Backward Branching p-value	**0.0%**	**0.0%**	**0.0%**	*21.4%*	*63.3%*	*68.8%*	**3.8%**	**0.4%**	**0.1%**

Figure 8.9: SED F1 for HornSiren: Baseline vs Branching method

8.3 Other Internal Systems

8.3.1 HornSiren

To test if the internal system can be replaced by other methods, the forward branching method was also tested on HornSiren with the Convolutional Gated Recurrent Neural Networks from section 4.4.2 as the internal system. In Figure 8.11 the results of this are compared to the winning method itself. It shows that the forward branching CGRNN no longer outperforms its internal method in this case. Although the forward branching method still has a higher mean than the CGRNN method on the threshold values $t = 0.10$ and $t = 0.20$, it is no longer significant ($p > 5\%$). The biggest change however is that for high threshold values the mean has gone down significantly compared to the CGRNN method. Combined with the significantly larger standard deviation, this means that the forward branching cannot be recommended over the CGRNN method. However, as can be seen in Figure 8.12, no significant difference between the max value of the CGRNN and the forward branching method can be found. (p-value of 48.4% with a Mann-Whitney U test)

A possible reason this internal method does worse is because the internal systems are exact copies of the original system. However, each of the internal systems is done on a simpler task. This causes an exploding gradient to occur in some cases. This can be seen when training the networks. The loss and accuracy when using the CGRNN method normally have a consistent pattern. However when this CGRNN system is used on the smaller datasets sudden major changes can sometimes happen, after which the neural network does not recover, and convergence is not found. This normal behaviour, and this sudden change can be seen in Figure 8.10. In this figure the normal gradient shows a mostly decreasing loss value, and a stable accuracy, while for the exploding gradient the loss goes up greatly when compared with the normal gradient, while the accuracy is very unstable.

```
loss: 0.1320 - acc: 0.9458
loss: 0.1306 - acc: 0.9466
loss: 0.1311 - acc: 0.9459
loss: 0.1301 - acc: 0.9467
loss: 0.1290 - acc: 0.9474
loss: 0.1300 - acc: 0.9468
loss: 0.1287 - acc: 0.9474
loss: 0.1276 - acc: 0.9481
loss: 0.1301 - acc: 0.9475
loss: 0.1292 - acc: 0.9481
loss: 0.1334 - acc: 0.9470
loss: 0.1319 - acc: 0.9476
loss: 0.1309 - acc: 0.9482
loss: 0.1348 - acc: 0.9471
loss: 0.1376 - acc: 0.9459
loss: 0.1393 - acc: 0.9454
loss: 0.1398 - acc: 0.9449
loss: 0.1392 - acc: 0.9449
loss: 0.1406 - acc: 0.9444
```

(a) Normal

```
loss: 0.3812 - acc: 0.8000
loss: 0.4151 - acc: 0.8500
loss: 0.3903 - acc: 0.8833
loss: 0.4915 - acc: 0.8250
loss: 0.5151 - acc: 0.7900
loss: 0.4894 - acc: 0.8000
loss: 0.5319 - acc: 0.7714
loss: 0.5346 - acc: 0.7687
loss: 0.5622 - acc: 0.7500
loss: 0.5886 - acc: 0.7400
loss: 0.5720 - acc: 0.7455
loss: 0.5841 - acc: 0.7417
loss: 0.5980 - acc: 0.7154
loss: 0.6053 - acc: 0.7000
loss: 0.6053 - acc: 0.6933
loss: 0.6025 - acc: 0.6937
loss: 0.6022 - acc: 0.6941
loss: 0.6016 - acc: 0.7000
loss: 0.5916 - acc: 0.7079
```

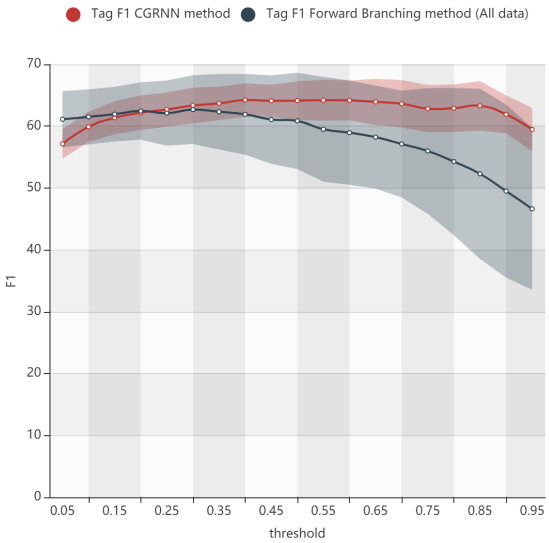
(b) Exploding

Figure 8.10: Normal versus Exploding gradient

The results of exploding gradients on the neural networks can be seen when comparing a single run of the CGRNN forward branching method. The results of a 2 runs, for the same threshold value can be found in Figure 8.13 and its accompanying table 8.2. Here we can see that in run 2 the Horn neural network has not converged, having a significantly lower F1 value. This one non-converged neural network then decreases the performance of the whole branching neural network significantly. The advantage for the branching neural network is that this network could be retrained, however it does mean that if the branching neural network needs to show consistently better results, the internal system needs to be tested on the smaller datasets for exploding or vanishing gradients.An explanation on why these non-converged neural networks are less of a problem on lower t values is that for lower threshold values the only important neural network is the internal neural network. As such, for lower threshold values, only the failing of one neural network will show, not three.

Node	run 1 F1	run 1 Precision	run 1 Recall	run 2 F1	run 2 Precision	run 2 Recall
Tree	86.4	87.9	85.0	87.4	85.6	89.2
Horn	76.3	66.7	89.3	12.0	35.4	7.1
Siren	61.5	51.5	75.5	60.5	54.9	67.2
Total	64.5	59.2	70.9	41.1	52.4	33.9

Table 8.2: Table accompanying Figure 8.13.



Threshold	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
Baseline F1 mean	59.9	62.2	**63.3**	**64.2**	**64.1**	**64.2**	**63.6**	**62.9**	**61.9**
Forward Branching F1 mean	**61.5**	**62.5**	62.7	61.9	60.9	59.0	57.1	54.3	49.5
Baseline F1 Standard Deviation	2.4	2.8	2.9	2.7	3.1	3.2	3.8	3.8	3.0
Forward Branching F1 Standard Deviation	4.5	4.6	5.6	6.5	7.8	8.4	8.6	11.9	14.0
Forward Branching p-value	*10.6%*	*31.2%*	*71.5%*	*45.5%*	*28.5%*	*15.4%*	**4.4%**	**1.6%**	**0.1%**

Figure 8.11: Tag F1 for HornSiren: CGRNN vs CGRNN Branching method

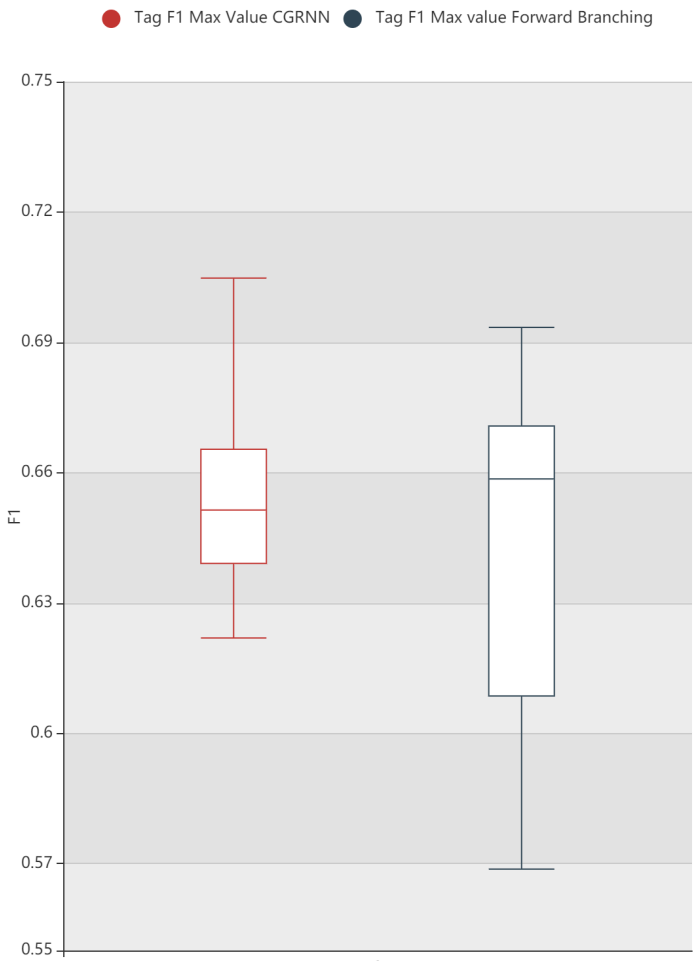
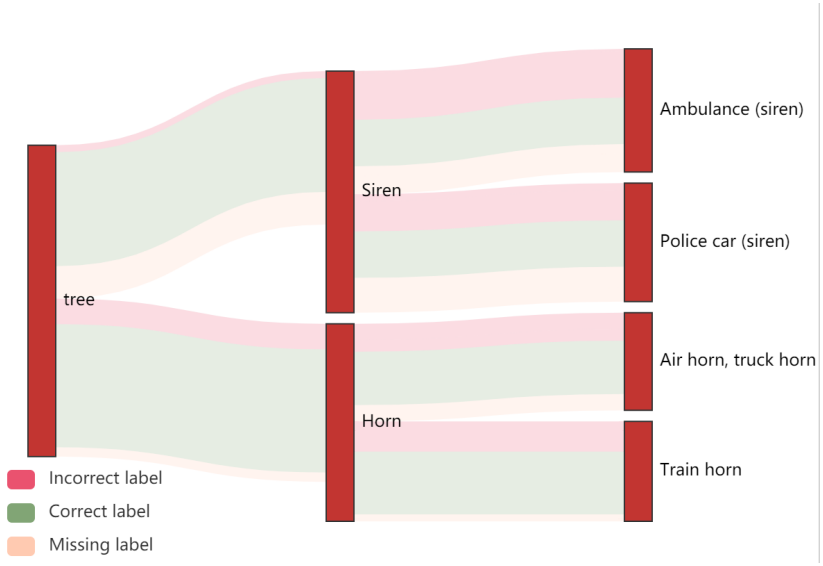
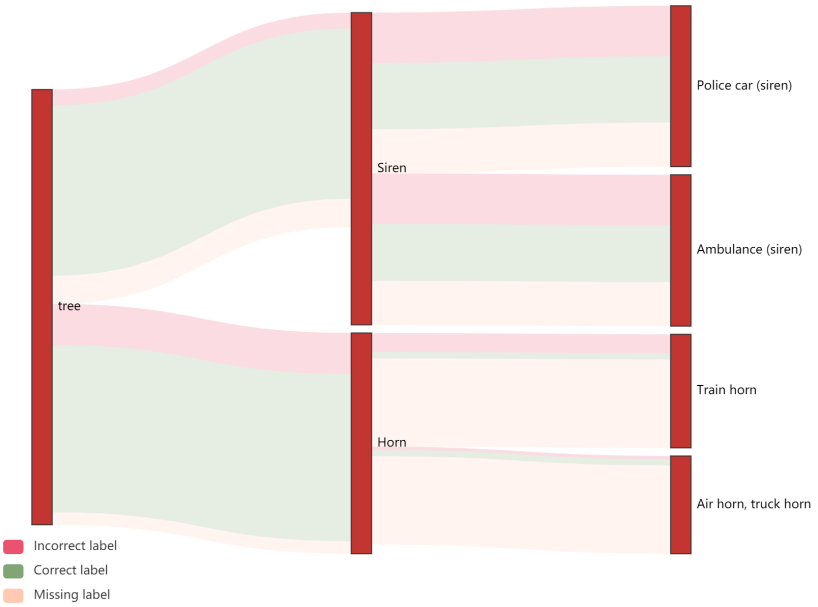


Figure 8.12: Max Value CGRNN vs Forward Branching



(a) run 1



(b) run 2

Figure 8.13: Sankeys of different runs for Forward CGRNN method for $t = 0.5$

Chapter 9

Conclusion

The goal of this thesis was to test the viability and the properties of our devised new branching neural network method for improving SED systems. To do this we asked multiple research questions.

Can we successfully build a classifier with this new Branching Neural Network method?

It has been shown that an SED can be build with this branching neural network by building multiple SED systems on multiple datasets for multiple variants of the branching neural network method.

Does this new method improve in F1 score and error rate on the current winner of the DCASE 2017 task 4 challenge?

Both forward and backward propagation have shown to be an improvement over the baseline method on lower thresholds, while maintaining performance on the other thresholds. This is not only on all covered datasets, but it also applies to both the tagging task and the SED task. These properties render branching neural networks a robust choice, as this means that using the branching neural network methods is the safer choice, especially if the threshold value is too low in the final implementation of the SED system. On some datasets the branching neural network methods were even shown to beat out the baseline method on all thresholds.

The intermediate information provided by branching neural networks, in the form of the outputs of the inner neural networks, allows one to gain useful information that the internal method did not provide, even on the thresholds where no significant difference is found. For example for a SED system built for smart car systems, knowing that a particular sound is a siren, could be more useful than knowing if it is a police or ambulance siren in some situations. This siren output was shown for the HornSiren dataset to be 29.5% more accurate than the police or ambulance siren output.

Can current state-of-the-art SED systems be used as the internal tree nodes for the new method? Specifically the Convolutional Gated Recurrent Neural Network (CGRNN) method?

The possibility of replacing the internal method was shown by replacing the baseline method with the CGRNN method. This shows that it is possible to replace the internal method with one that is more accurate, when it is formulated, which could increases the accuracy of the system created with the branching neural network method.

The forward branching method did not beat the CGRNN method while using it as an internal method on the HornSiren subset, as the neural networks did not converge consistently. However, it was also not shown that the CGRNN method beat out the branching neural network method, as results were inconclusive.

What is the training and running time of this method versus the current winning method? How much does randomly changing the ontology change the training and running time of the method? Is this change dependant on the shape of the network?

One disadvantage of the branching neural network method is the increase in training time, as each decision requires another neural network to train. Luckily multiple neural networks can be trained an run at the same time, allowing parallel computation. With enough computing power there would be no difference for the forward branching neural network, however for the backward branching neural network only each layer can be trained at the same time, meaning that even with enough computing power the minimum training time is $l * t$, where l is the number of layers, and t the training time of the internal method. This means that the running time of the branching neural network methods is dependant on the shape of the tree, and that each decision added will add to the training and running time of the method, especially if this decision adds more layers to the tree.

How much does randomly changing the ontology change the error rate and F1 score?

Another disadvantage of the branching neural network is the importance of the tree structure. Even though it was shown that the branching method with an non-optimal tree still performed better on lower thresholds, and comparable on the other thresholds, than the baseline method, the extended training time needed for training multiple of the same neural networks might not warrant the small increase in performance. This however implies that with tuning of the tree a more optimal tree could be found for a particular dataset.

Does tuning the ontology based on the number of instances for each class change the results?

All subsets given were balanced subsets, removing the necessity to tune the ontology based on the number of instances for each class. This however does mean that it has not been tested how the SED systems performed for a less balanced dataset.

9.1 Future Work

As it was shown that the internal method could be replaced with the GCRNN method, future work includes tuning the branching neural network method enough that this replacement process has consistent results, for state-of-the-art methods. This might include a way to decrease the size of the internal methods to smaller parts designed for the smaller tasks the internal systems need to perform. It might even be possible for this to be done automatically, without increasing the training time, maybe even decreasing training time by simplifying the network.

A second direction the research could be taken to is automatic creation of the tree by using unsupervised learning methods to find the best tree splits and branches. If this could be done successfully, the big weakness of bad trees could be circumvented. This would not only mean not needing the extra tree input compared to the standard SED methods, but also that tuning of the tree based on ontologies might no longer be necessary.

A third direction for further research is different applications. Theoretically the branching method could work on any type of neural network. This means research could be extended into other systems built with neural networks, like object detection systems, which are systems that use neural networks to detect the presence of objects in pictures. This could even be done in such a way that some internal choices use one system whilst others use another. For example that the branching neural network consist of both object detection nodes, and SED nodes.

A fourth direction is towards using more information from the internal groups. It has been shown that internal groups have higher performance than the external groups. We could therefore use a system that needs multiple groups to agree before the data flows towards the next neural network to see if this would increase performance. This situation is pictured in Figure 9.1, where both *Red* and *Fruit* classifications are necessary before reaching *Red Fruits*. This could even be combined with multiple different systems, where both a sound and a object need to exist before continuing onto the next layer.

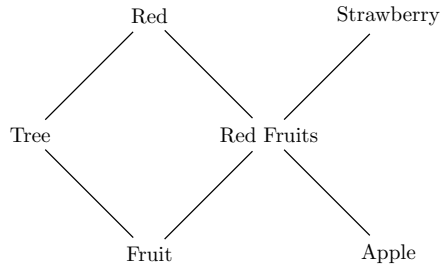


Figure 9.1: Joining branches

A fifth direction for further research is a different way of threshold tuning.

In the work so far we have only tuned in a forwards direction, from the root of the tree each network was tuned in order. However, another option becomes available in our tree structure. We could tune deeper networks first, and afterwards tune backwards. This cascading t value tuning would allow the tuning of the internal networks to have more information about the performance of the later networks, and in turn increase the performance of their tuning. This allows us for example, to attribute a different value to data labelled *Police car (siren)* that goes to the *Siren* network but ends up in the *Ambulance (siren)* category in the final decision, than if that data would have gone to the *Horn* category. Creating a tuning method that can take these special cases into consideration.

A sixth direction is analyzing the effect the balancing the datasets. In the current case the datasets were all balanced, however this means the results only follow for balanced training sets. Of course all datasets can be balanced by not using all the data in the training set. In fact that is how the balanced training sets for this research were created. However another method of balancing could be by multiplying data points, or using partial audio files to duplicate labels that are not abundant. However, it might not be necessary to balance the datasets at all. By doing further research in this direction the viability for the branching methods can be found for unbalanced datasets.

The repository for the code, so that this future work can be performed, can be found at:
`https://git.science.uu.nl/K.A.Schalkwijk/branchingNN`

Acknowledgement

I would like to thank my project supervisor Dr. A. Volk from Utrecht University for the guidance I needed to get this idea on paper and everyone from the Intrasonics crew with the help necessary to get it in code. I would also like to thank Dr. P. Kranenburg from Utrecht University whose fresh eyes helped me see the questions that up until that point were only answered in my head. A special thanks is in order for both Dr. J. Schalkwijk and Dr. P. Kelly from Intrasonics for helping me ask the right questions during my explorations, a feat that sounds easier when the questions have been asked than when you're looking at a dead end.

References

Adavanne, S., & Virtanen, T. (2017). Sound event detection using weakly labeled dataset with stacked convolutional and recurrent neural network. *arXiv preprint arXiv:1710.02998*.

Agarwal, M. (2017). *Back propagation in convolutional neural networks - intuition and code.* Retrieved from `https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199`

Banerjee, S. (2018). *An introduction to recurrent neural networks.* Retrieved from `https://medium.com/explore-artificial-intelligence/an-introduction-to-recurrent-neural-networks-72c97bf0912`

Benetos, E., Lafay, G., Lagrange, M., & Plumbley, M. D. (2016). Detection of overlapping acoustic events using a temporally-constrained probabilistic model. In *Acoustics, speech and signal processing (ICASSP), 2016 IEEE international conference on* (pp. 6450–6454).

Benetos, E., Lafay, G., Lagrange, M., Plumbley, M. D., Benetos, E., Lafay, G., ... Plumbley, M. D. (2017). Polyphonic sound event tracking using linear dynamical systems. *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, *25*(6), 1266–1277.

Cai, R., Lu, L., Hanjalic, A., Zhang, H.-J., & Cai, L.-H. (2006). A flexible framework for key audio effects detection and auditory context inference. *IEEE Transactions on audio, speech, and language processing*, *14*(3), 1026–1039.

Cakir, E., Heittola, T., Huttunen, H., & Virtanen, T. (2015). Polyphonic sound event detection using multi label deep neural networks. In *Neural networks (IJCNN), 2015 international joint conference on* (pp. 1–7).

Cakir, E., Ozan, E. C., & Virtanen, T. (2016). Filterbank learning for deep neural network based polyphonic sound event detection. In *Neural networks (IJCNN), 2016 international joint conference on* (pp. 3399–3406).

Cakir, E., Parascandolo, G., Heittola, T., Huttunen, H., Virtanen, T., Cakir, E., ... Virtanen, T. (2017). Convolutional recurrent neural networks for polyphonic sound event detection. *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, *25*(6), 1291–1303.

Cavaioni, M. (2017). *Machine learning: Decision tree classifier.* Retrieved from `https://medium.com/machine-learning-bites/machine-learning-decision-tree-classifier-9eb67cad263e`

Changhau, I. (2017). *Loss function in neural networks.* Retrieved from `https://isaacchanghau.github.io/post/loss_functions/`

Chou, S.-Y., Jang, S., & Yang, Y.-H. (2017). Framecnn: a weakly-supervised learning framework for frame-wise acoustic event detection and classification. *Recall*, *14*, 55–4.

Chung, Y., Lee, J., Oh, S., Park, D., Chang, H., & Kim, S. (2013). Automatic detection of cow's oestrus in audio surveillance system. *Asian-Australasian journal of animal sciences*, *26*(7), 1030–1030.

Cotton, C. V., & Ellis, D. P. (2011). Spectral vs. spectro-temporal features for acoustic event detection. In *Applications of signal processing to audio and acoustics (WASPAA), 2011 IEEE workshop on* (pp. 69–72).

Dauphin, Y. N., Fan, A., Auli, M., & Grangier, D. (2016). Language modeling with gated convolutional networks. *arXiv preprint arXiv:1612.08083*.

Dikmen, O., & Mesaros, A. (2013). Sound event detection using non-negative dictionaries learned from annotated overlapping events. In *Applications of signal processing to audio and acoustics (WASPAA), 2013 IEEE workshop on* (pp. 1–4).

Elizalde, B., Kumar, A., Shah, A., Badlani, R., Vincent, E., Raj, B., & Lane, I. (2016). Experiments on the DCASE challenge 2016: Acoustic scene classification and sound event detection in real life recording. *arXiv preprint arXiv:1607.06706*.

Fayek, H. (2016). *Speech processing for machine learning: Filter banks, mel-frequency cepstral coefficients (MFCCs) and what's in-between.* Retrieved from `https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html`

Foggia, P., Petkov, N., Saggese, A., Strisciuglio, N., & Vento, M. (2015). Reliable detection of audio events in highly noisy environments. *Pattern Recognition Letters*, *65*, 22–28.

Gemmeke, J. F., Ellis, D. P., Freedman, D., Jansen, A., Lawrence, W., Moore, R. C., ... Ritter, M. (2017). Audio set: An ontology and human-labeled dataset for audio events. In *Acoustics, speech and signal processing (ICASSP), 2017 IEEE international conference on* (pp. 776–780).

Gemmeke, J. F., Vuegen, L., Karsmakers, P., Vanrumste, B., & Van hamme, H. (2013). An exemplar-based nmf approach to audio event detection. In *Applications of signal processing to audio and acoustics (WASPAA), 2013 IEEE workshop on* (pp. 1–4).

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770–778).

Heittola, T., Mesaros, A., Eronen, A., & Virtanen, T. (2013). Context-dependent sound event detection. *EURASIP Journal on Audio, Speech, and Music Processing*, *2013*(1), 1.

Heittola, T., Mesaros, A., Virtanen, T., & Eronen, A. (2011). Sound event detection in multisource environments using source separation. In *Machine listening in multisource environments.*

Heittola, T., Mesaros, A., Virtanen, T., & Gabbouj, M. (2013). Supervised model training for overlapping sound events based on unsupervised source separation. In *2013 IEEE international conference on acoustics, speech and signal processing* (pp. 8677–8681).

Jeong, I.-Y., Lee, S., Han, Y., & Lee, K. (2017). Audio event detection using multiple-input convolutional neural network. *Detection and Classification of Acoustic Scenes and Events (DCASE).*

Karn, U. (2016a). *An intuitive explanation of convolutional neural networks.* Retrieved from `https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/`

Karn, U. (2016b). *A quick introduction to neural networks.* Retrieved from `https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/`

Karpathy, A. (2015). *The unreasonable effectiveness of recurrent neural networks.* Retrieved from `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`

Kim, B., & Pardo, B. (2018). A human-in-the-loop system for sound event detection and annotation. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, *8*(2), 13–13.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kohl, N. (2010). *Role of bias in neural networks.* Retrieved from `https://stackoverflow.com/questions/2480650/role-of-bias-in-neural-networks`

Komatsu, T., Senda, Y., & Kondo, R. (2016). Acoustic event detection based on non-negative matrix factorization with mixtures of local dictionaries and activation aggregation. In *Acoustics, speech and signal processing (ICASSP), 2016 IEEE international conference on* (pp. 2259–2263).

Kong, Q., Xu, Y., Wang, W., & Plumbley, M. D. (2017). A joint detection-classification model for audio tagging of weakly labelled data. In *2017 IEEE international conference on acoustics, speech and signal processing (ICASSP)* (pp. 641–645).

Kumar, A., Khadkevich, M., & Fügen, C. (2018). Knowledge transfer from weakly labeled audio using convolutional neural network for sound events and scenes. In *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)* (pp. 326–330).

Kumar, A., & Raj, B. (2016). Audio event detection using weakly labeled data. In *Proceedings of the 2016 ACM on multimedia conference* (pp. 1038–1047).

Kumar, A., & Raj, B. (2017). Audio event and scene recognition: A unified approach using strongly and weakly labeled data. In *Neural networks (IJCNN), 2017 international joint conference on* (pp. 3475–3482).

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324.

Lee, D., Lee, S., Han, Y., & Lee, K. (2017). Ensemble of convolutional neural networks for weakly-supervised sound event detection using multiple scale input. *Detection and Classification of Acoustic Scenes and Events (DCASE)*.

Lu, R., & Duan, Z. (2017). Bidirectional gru for sound event detection. *Detection and Classification of Acoustic Scenes and Events*.

Lucas Schuermann, C. M. (2016). *The math behind backpropagation.* Retrieved from `https://bigtheta.io/2016/02/27/the-math-behind-backpropagation.html`

Lyons, J. (2013). *Mel frequency cepstral coefficient (MFCC) tutorial.* Retrieved from `http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs//#computing-the-mel-filterbank`

McKnight, P. E., & Najab, J. (2010). Mann-Whitney U test. *The Corsini encyclopedia of psychology*, 1–1.

Mesaros, A., Heittola, T., Dikmen, O., & Virtanen, T. (2015). Sound event detection in real life recordings using coupled matrix factorization of spectral representations and class activity annotations. In *Acoustics, speech and signal processing (ICASSP), 2015 IEEE international conference on* (pp. 151–155).

Mesaros, A., Heittola, T., Diment, A., Elizalde, B., Shah, A., Vincent, E., ... Virtanen, T. (2017). DCASE 2017 challenge setup: Tasks, datasets and baseline system. In *DCASE 2017-workshop on detection and classification of acoustic scenes and events.*

Mesaros, A., Heittola, T., Eronen, A., & Virtanen, T. (2010). Acoustic event detection in real life recordings. In *Signal processing conference, 2010 18th european* (pp. 1267–1271).

Mesaros, A., Heittola, T., & Virtanen, T. (2016). Metrics for polyphonic sound event detection. *Applied Sciences*, *6*(6), 162.

Müller, M. (2015). *Fundamentals of music processing: Audio, analysis, algorithms, applications.* Springer.

Nielsen, M. (2018). *Why are deep neural networks hard to train?* Retrieved from `http://neuralnetworksanddeeplearning.com/chap5.html`

Parascandolo, G., Huttunen, H., & Virtanen, T. (2016). Recurrent neural networks for polyphonic sound event detection in real life recordings. *arXiv preprint arXiv:1604.00861*.

Quinlan, J. R. (1986). Induction of decision trees. *Machine learning*, *1*(1), 81–106.

Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems* (pp. 91–99).

Rohan Badlani, A. S. (2017). *Large-scale weakly supervised sound event detection for smart cars.* Retrieved from `http://www.cs.tut.fi/sgn/arg/dcase2017/challenge/task-large-scale-sound-event-detection`

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., ... Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, *115*(3), 211-252. doi: 10.1007/s11263-015-0816-y

Salamon, J., McFee, B., Li, P., & Bello, J. P. (2017). *DCASE 2017 submission: Multiple instance learning for sound event detection* (Tech. Rep.). Technical report, DCASE2017 Challenge (September 2017).

Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, *45*(11), 2673–2681.

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

Su, T.-W., Liu, J.-Y., & Yang, Y.-H. (2017). Weakly-supervised audio event detection using event-specific gaussian filters and fully convolutional networks. In *2017 IEEE international conference on acoustics, speech and signal processing (ICASSP)* (pp. 791–795).

Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. A. (2017). Inception-v4, inception-ResNet and the impact of residual connections on learning. In *AAAI* (Vol. 4, p. 12).

Temko, A., Nadeu, C., Macho, D., Malkin, R., Zieger, C., & Omologo, M. (2009). Acoustic event detection and classification. In *Computers in the human interaction loop* (pp. 61–73). Springer.

Vuegen, L., Broeck, B., Karsmakers, P., Gemmeke, J., Vanrumste, B., & Hamme, H. (2013). An MFCC-GMM approach for event detection and classification. In *IEEE workshop on applications of signal processing to audio and acoustics (WASPAA)* (pp. 1–3).

Walber. (n.d.). *Precisionrecall.svg.* Retrieved from `https://en.wikipedia.org/wiki/File:Precisionrecall.svg`

Wang, Y., & Metze, F. (2017). A first attempt at polyphonic sound event detection using connectionist temporal classification. In *Acoustics, speech and signal processing (ICASSP), 2017 IEEE international conference on* (pp. 2986–2990).

Xu, Y., Kong, Q., Wang, W., & Plumbley, M. D. (2018). Large-scale weakly supervised audio classification using gated convolutional neural network. In *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)* (pp. 121–125).

Yu, C.-Y., Liu, H., & Qi, Z.-M. (2017). *Sound event detection using deep random forest* (Tech. Rep.). Technical report, DCASE2017 Challenge.