NTNU
Norwegian University of
Science and Technology

Maciej Piatkowski

Classification of speech samples using multiple Neural Networks in a tree hierarchy

June 2020

NTNU
Norwegian University of
Science and Technology

Norwegian University of
Science and Technology

Classification of speech samples using multiple Neural Networks in a tree hierarchy

Maciej Piatkowski

Master in Applied Computer Science
Submission date: June 2020
Supervisor: Mariusz Nowostawski

Norwegian University of Science and Technology
Department of Computer Science

Classification of speech samples using multiple Neural Networks in a tree hierarchy

Maciej Piatkowski

CC-BY 2020/06/02

Abstract

This report describes the research project which aimed to investigate ways to reduce the load on the high-end hardware used in training neural networks by utilizing many smaller neural networks, rather than one big network. Neural networks used in this thesis are iteratively re-trained on a progressively larger dataset and then used to form a tree hierarchy. Various learnings from previous research done in the field are applied to accelerate the development of the neural network model to achieve satisfactory results before proceeding with the main experiments.

During the training process, a modified loss function with a filter is applied to guide the neural network to achieve better classifications for the given samples. The filter is applied by adding an extra layer of neurons after the softmax layer, which is then discarded after the training process is finished. Weights of the extra layer have been manually modified to transfer the results of the softmax layer directly to the extra layer. As the goal of this process is to create better classifications, the resulting network is discarded at the end of the process. Better classifications are used as the basis for the clustering of the dataset used in the thesis.

The iterative re-training takes these better classifications, and uses them in the training process as more and more of the dataset is processed. Finally, the networks are assembled to form a tree, chaining clusters together to distribute the dataset into smaller fragments. Both iterative re-training and the neural network tree are attempted in combination. Additionally, a control group not using iterative re-training is being attempted as well.

Results of the thesis show that iterative re-training has some effect on the size and quality of the original data clusters, assuming one tunes the parameters of the networks appropriately. Through the trees generated by using iterative re-training, success is shown by comparing several audio samples that were grouped to same and different tree nodes. Modification of the loss function is shown to have little effect, but the entire process shows clear increase in quality over the alternatives.

Sammendrag

Denne rapporten beskrier et forskningsprosjekt som hadde som mål å utforske metoder til å redusere byrden på det utstyret fra høyeste hylle brukt til å trene nevrale nettverk gjennom å bruke mange mindre nevrale nettverk, i stedet for et stort nettverk. Nevrale nettverk brukt i denne masteroppgaven er re-trent iterativt på et progressivt større sett med data, og deretter brukt i et tre-hierarki. Forskjellige lærdommer fra tidligere forskning ute i feltet er brukt til å akselerere utviklingen av den nevrale nettverk modellen til å oppnå gode resultater før hoved-eksperimentene begynner.

Gjennom treningsprosessen, en modifisert tapsfunksjon med en filter er brukt til å rettlede det nevrale nettverket til å oppnå bedre klassifiseringer for gitte dataprøver. Filteret er brukt gjennom å legge til et ekstra lag med nevroner etter softmax laget, som er deretter fjernet etter treningsprosessen er ferdig. Vektene av dette ekstra-laget er manuelt modifisert til å overføre resultatene fra softmax-laget direkte til ekstra-laget. Ettersom målet med denne prosessen er å skale bedre klassifiseringer, det resulterende nettverket er fjernet på slutten av denne prosessen. Bedre klassifiseringer er brukt som grunnlag for grupperingen av datasettet brukt i denne masteroppgaven.

Den iterative re-treningen tar disse bedre klassifiseringene, og bruker de i treningsprosessen ettersom mer og mer av datasettet er bearbeidet. Til slutt er nettverkene samlet i et tre, og gruppene er kjedet sammen til å distribuere datasettet til mindre fragmenter. Både den iterative re-treningen og det nevrale nettverktreet er forsøkt i kombinasjon. I tillegg er en kontroll-gruppe som ikke bruker den iterative re-treningen brukt til å danne et tre.

Resultatet av masteroppgaven viser at iterativ re-trening har noe effekt på størrelsen og kvaliteten av de opprinnelige data gruppene, så lenge man justerer på parameterne i nettverkene korrekt. Gjennom trærne generert gjennom iterativ re-trening, vellykkede eksperiment er vist gjennom sammenligning av flere lyd klipp som ble gruppert i samme og forskjellige tre noder. Modifisering av tapsfunksjonen er vist til å ha liten effekt, men hele prosessen viser en klar hevelse av kvaliteten over alternativene.

Contents

Abstract . . . iii
Sammendrag . . . v
Contents . . . vii
Figures . . . xi
Tables . . . xiii
Code Listings . . . xv
1 Introduction . . . 1
1.1 Keywords . . . 1
1.2 Research questions . . . 2
1.3 Contributions . . . 2
2 Background . . . 3
2.1 Neural networks . . . 3
2.1.1 History . . . 4
2.1.2 Convolutional neural networks . . . 5
2.1.3 Training the neural network . . . 7
2.1.4 Transfer learning . . . 9
2.2 Audio processing . . . 9
2.2.1 Fourier transform . . . 9
2.2.2 Mel scale . . . 9
2.2.3 Discrete cosine transform . . . 10
2.2.4 Mel-frequency cepstrum coefficients . . . 10
2.3 Hierarchical clustering . . . 11
2.4 Neural networks and data clustering . . . 12
3 Methodology . . . 15
3.1 Dataset . . . 15
3.1.1 Processing steps . . . 16
3.1.2 Statistics . . . 21
3.2 Neural network . . . 23
3.2.1 Early development . . . 23
3.2.2 Extra layer . . . 24
3.2.3 Final model . . . 24
3.3 Loss function . . . 26
3.3.1 Primary filter . . . 26
3.3.2 Second filter . . . 27

3.4 Iterative re-training . . . 28
3.5 Tree generation . . . 28
4 Experiments . . . 31
4.1 Tools used in the thesis . . . 31
4.1.1 Hardware . . . 31
4.1.2 Software . . . 33
4.2 Dataset selection . . . 36
4.2.1 Experiment setup . . . 36
4.2.2 Experiment results . . . 38
4.3 Extra layer experiment . . . 38
4.3.1 Experiment setup . . . 39
4.3.2 Experiment results . . . 39
4.4 Neural network refinement experiments . . . 40
4.5 Loss function experiments . . . 42
4.5.1 Secondary filter . . . 42
4.5.2 Correct cluster count reevaluation . . . 42
4.5.3 Experiment results . . . 43
4.6 Iterative re-training experiment . . . 43
4.6.1 Parameters used . . . 43
4.6.2 Experiment process . . . 43
4.6.3 Experiment results . . . 44
4.7 Tree generation experiment . . . 44
4.7.1 Parameters used . . . 44
4.7.2 Experiment process . . . 44
4.7.3 Experiment results . . . 45
5 Results . . . 47
5.1 Iterative re-training results . . . 47
5.1.1 Size of clusters over the iterations . . . 47
5.1.2 Value of the highest classifications over the iterations . . . 47
5.1.3 Variation between first iteration and the following iterations 49
5.1.4 Variation between neighbor iterations . . . 50
5.1.5 Number of sample files between training iterations . . . 50
5.2 Tree results . . . 54
5.2.1 Cache size of the nodes in the tree . . . 54
5.2.2 Re-training iterations throughout the trees . . . 56
5.2.3 Failed node generation per branch layer . . . 56
5.2.4 Comparison between samples classified by the tree . . . 57
5.2.5 Control insight . . . 57
6 Discussion . . . 61
6.1 Iterative re-training performance . . . 61
6.2 Tree generation process . . . 62
6.3 Effect of parameter change . . . 63
7 Conclusion . . . 65
7.1 Future work . . . 66

Bibliography **67**

Figures

2.1 Illustration of a simple feed-forward neural network 3
2.2 The Inception Module[8], without (left) and with (right) pooling layers in the module. 5
2.3 Example of a convolution operation. Padding operations on the borders not included. 6
2.4 Two examples of the pooling operation on the same initial values . 7
2.5 Two examples of 80 MFCCs generated with DCT-2 and DCT-3 on the same 3.4-second long audio sample 10
2.6 Illustration of a data tree, arrows represent starting point for clustering process 12

3.1 Length of the samples in the dataset 23
3.2 Overlap statistics from the samples that had an overlap recorded . . 23
3.3 Example of the 1010 neuron layer 25
3.4 Loss function filters 0 and 1 27
3.5 Loss function filters 2 and 3 27

4.1 Largest class and validation accuracy as number of samples increase 40
4.2 Custom layer performance with the extra operations 40
4.3 Normal layer performance with the extra operations 41

5.1 Largest cluster over the iterations, with 0.8 iteration threshold . . . 48
5.2 Largest cluster over the iterations, with 0.5 and 0.6 iteration threshold 48
5.3 Largest cluster over the iterations, with 0.7 and 0.9 iteration threshold 48
5.4 Samples below iteration threshold, with parameter set to 0.8. 49
5.5 Samples below iteration threshold, with parameter set to 0.5 and 0.6 49
5.6 Samples below iteration threshold, with parameter set to 0.7 and 0.9. 50
5.7 Variation between first and next iterations, with 0.8 iteration threshold 51
5.8 Variation between first and next iterations, with 0.5 and 0.6 iteration threshold 51
5.9 Variation between first and next iterations, with 0.7 and 0.9 iteration threshold 52
5.10 Variation between neighbor iterations, with 0.8 iteration threshold 52

5.11 Variation between neighbor iterations, with 0.5 and 0.6 iteration threshold . . . 53
5.12 Variation between neighbor iterations, with 0.7 and 0.9 iteration threshold . . . 53
5.13 Change in number of sample files between iterations, with 0.8 iteration threshold . . . 54
5.14 Change in number of sample files between iterations, with 0.5 and 0.6 iteration threshold . . . 55
5.15 Change in number of sample files between iterations, with 0.7 and 0.9 iteration threshold . . . 55
5.16 Largest tree node caches in descending order. All start from full dataset. . . . 56
5.17 Largest tree node iterations in descending order. Excluding full dataset. . . . 57
5.18 Nodes that failed to be created due to insufficient samples in the cache . . . 58
5.19 Samples left per branch level over the course of the tree generation 58
5.20 Audio sample 1, English translation: "Kabu, I'm home!" . . . 58
5.21 Audio sample 2, English translation: "Nu-uh! I'll go by myself..." . . 59
5.22 Audio sample 3, English translation: "All right, I'm finished" . . . 59
5.23 Largest tree node clusters in the control group, in descending order. Including full dataset. . . . 59
5.24 Node failures and samples left per branch level in the control group. 60

Tables

3.1 Time spent processing the dataset 22
3.2 Number of subtitle lines over the course of processing the dataset . 22

4.1 Hardware specs 31
4.2 Software versions 33
4.3 Early neural network model 36
4.4 Librosa functions that were tested 37
4.5 Experiment 1 results 38
4.6 Modified early neural network model 39
4.7 Modified early neural network model 41

Code Listings

4.1 Code to reduce memory usage of Tensorflow in a single console, valid for Tensorflow 2.1 34

Chapter 1

Introduction

In recent years, Artificial Intelligence, through the use of Neural Networks, has become widely used in a broad spectrum of applications. While NNs have shown themselves to be powerful, a point has been reached where the time it takes to build NNs goes up exponentially. Meanwhile, the results are not that much better than what simpler NNs achieve.
Current NNs have shown themselves to be superior to previously developed solutions. However, they currently suffer from the flaw of having to remember all of the information they have seen. If they lose some information while learning something new, this loss may cause them to perform worse in tasks they were previously solving well. In general, the more a network needs to do, the bigger it has to be to do it. The need for a more extensive network leads to a performance problem where one needs costly, powerful hardware to train it in a reasonable time frame. Handling the problem of more extensive networks with brute-forcing the problem through more expensive hardware is not sustainable in the long term. Eventually, the network will require even more performant equipment to process than what will be available on the market.
This master project seeks to test creating multiple NNs that will be able to accurately predict the English translation of a phrase spoken by a person in Japanese. Each NN will be trained iteratively on small batches of the dataset at a time, with the output of the NNs being one of several "super-classes." For each super-class, a new NN will be generated for the data assigned to this super-class, creating unique, smaller super-classes.

1.1 Keywords

Artificial intelligence, Neural networks, Transfer learning, Convolutional Neural Network

1.2 Research questions

The research questions of this master thesis can be boiled down to the following three questions:

- How do neural networks that are iteratively re-trained using transfer learning on an increasing subset of the dataset, to group the entire dataset they receive into "super-classes" perform?
- How does this training technique perform when used to train neural networks in a tree hierarchy that bases itself on these super-classes?
- How does changing the parameters of the training process affect the neural networks and the resulting tree structures?

1.3 Contributions

The research area of this master thesis revolves around the improvement of the scalability of neural networks. The first research question seeks to analyze the iterative transfer learning process of each NN by itself. As the size of the dataset is increased, plugging the entire dataset into the training process will lead to time spent training increasing at minimum linearly with the dataset size. In addition to increased time to train, storage requirements also go up with dataset size, requiring the dataset to be stored on slower memory. Using too much memory can potentially even kill the training process due to insufficient resources on the weakest link in the computing chain. Different networks and parameters may exhibit different characteristics. Analyzing the performance of each NN by itself can bring valuable results for many different applications that would seek to use a similar iterative transfer learning process.
The second research question seeks to expand on the results of the first by introducing the tree hierarchy concept. Most NNs currently in use are single, large models that take a lot of time and processing power to train. While some research has been done in using NNs in hierarchical clusters, this thesis aims to research more autonomous ways for such trees to be created. The tree generation would be done from the view of the neural network, rather than based on human intuition. It is ultimately the neural networks that have to use the super-classes, while humans are generally interested only in the final result. The hypothesis is that the networks should decide on how to distribute the dataset across the super-classes.
The final question aims to compare the different neural networks created during this master thesis against each other to find which NNs worked better or worse in various metrics. Some of the NNs could perform better in generating the root node of the tree, while suddenly collapsing somewhere deeper in the tree. Unfortunately, a direct comparison between each tree will be impossible due to the autonomous nature of the tree generation process. Still, providing some form of analysis between the NNs can give insight into potential pitfalls that some of the trees fell into that others did not.

Chapter 2

Background

2.1 Neural networks

Neural networks are something everyone has, even if they do not understand the concept. Neurons in our brains transmit signals to other neurons, which in turn do the same to the next neurons in line. Some neurons are connected to many neurons, while other neurons may only be connected to a few. Through their connections, the neurons are creating all sorts of connection shapes ranging from massive trees to a short loop. Throughout a lifetime, these connections change as the brain learns and forgets information.

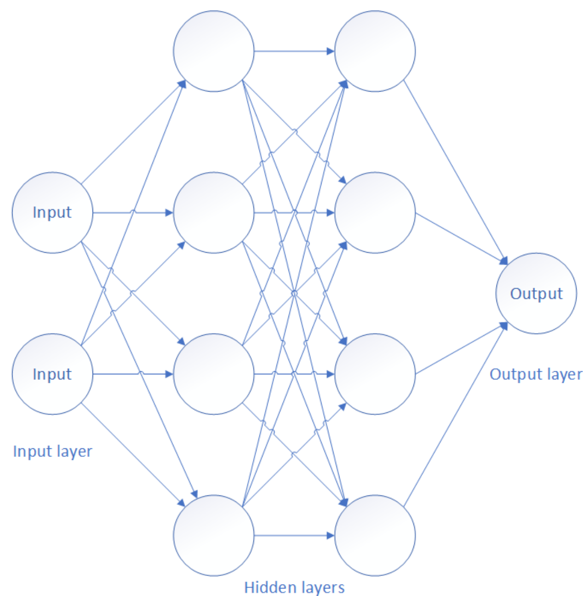


Figure 2.1: Illustration of a simple feed-forward neural network

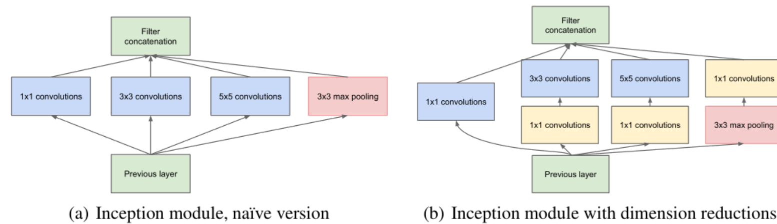
In computing, standard simple neural networks are clearly defined into layers, with a clear point of entry called the "input" layer, with the result coming out of the

"output" layer. A basic illustration of this is provided in Figure 2.1. Information we want to be processed into the output result is inserted into each of its designated input neurons. Between the input and output layers, we can place one or more hidden layers. These hidden layers abstract the input from the output, allowing the network more flexibility in making connections between input and output. In the case of Figure 2.1, the input could be the current weekday and the time of day, with the output being a value between 0 and 1 determining if it is time to eat dinner. On weekdays it could say that between 18:00 and 19:00 is the best time to eat dinner, while during the weekend it could extend the time to be sooner or later.

2.1.1 History

History of artificial neural networks used in computing can be traced back to its common roots with medicine and psychology that the NNs attempt to mimic. Towards the late 1940s, Donald Hebb [1] described a theory of how cells in a brain function together. As the brain cells fire electrical signals to other cells, those connections are strengthened and happen more frequently. Artificial neural networks use this theory loosely to translate the signal received in the input neurons into proper outputs. While the Hebbian theory allowed for the creation of first neural networks, these networks were not very useful due to limitations in computing power and lack of more complex structures. The creation of deep neural networks with multiple layers was practically impossible until the creation of the backpropagation algorithm in 1975 by Paul Werbos[2]. Backpropagation allows for the errors in the learning process to be sent back through multiple network layers, and adjust all of the weights in the network.
Even with the creation of the backpropagation algorithm, the computational power of the time did not allow for very complex neural networks. As execution in software was too difficult at the time, hardware solutions were created. Using the recent at the time metal-oxide semiconductors, in 1989, neural networks were implemented using very-large-scale integration in analog, rather than digital[3]. During the following two decades, various techniques were developed to enable neural networks to handle more complex problems. Among others, max-pooling was introduced in 1992[4], and continuous improvement of existing and new technologies enabled neural networks to grow in relevance as more powerful hardware became available.
The first in the series of convolutional neural networks that drastically improved the field of Artificial Intelligence is the deep neural network by Alex Krizhevsky [5], created in 2012. A neural network becomes "deep" when more than one layer is placed between the input and output layer. AlexNet was constructed with five convolutional layers, followed by three dense layers, of which the last dense layer was the output layer. Also, several max-pooling layers were placed throughout the model. By using multiple convolutional layers in sequence, AlexNet managed to beat all of its competition that year in the ImageNet[6] challenge[7].

After this breakthrough, commercial entities like Google used the findings in this paper to create the Inception[8] network. This network utilized a combination of different convolutional layers in what it calls the Inception Module, seen in figure Figure 2.2. Unlike the previous models where the next layer strictly followed one layer, the Inception network has one-to-many and many-to-one connections that enable it to do multiple different operations on input from the same previous layer. By combining the layers into building blocks, and then stacking them on top of each other, the first Inception NN beat out its competition in the 2014 challenge[9].



(a) Inception module, naïve version (b) Inception module with dimension reductions

Figure 2.2: The Inception Module[8], without (left) and with (right) pooling layers in the module.

2.1.2 Convolutional neural networks

While a neural network like the one shown in Figure 2.1 can be used to receive some results for simple data with no structure in the input, extracting information from sound and images requires the network to understand the relationship between each input. For the network to learn the meaning of the input structure, several different types of layers are mixed, each applying its specific operation to the input it receives.

Dense layer

The dense layer is the simplest of all layers in all neural networks. Looking back at Figure 2.1, the hidden layer in the figure is a dense layer. A dense layer only applies a simple multiplication operation on the input it receives with the weights of the connections between the neurons. In convolutional neural networks, this layer is usually placed at the end of the network to represent the features that the previous layer has learned to detect.

Convolutional layer

The convolutional layer is the central part of the convolutional neural network. Unlike the dense layer that applies only a simple multiplication operation, this layer applies a filter over multiple inputs next to each other. This filter is can also be called the convolution window or the kernel. The filter applies a multiplication

operation to every item in the convolution window and then sums all of the results into one number. An example of the thesis relevant 1D convolution operation can be seen in Figure 2.3.

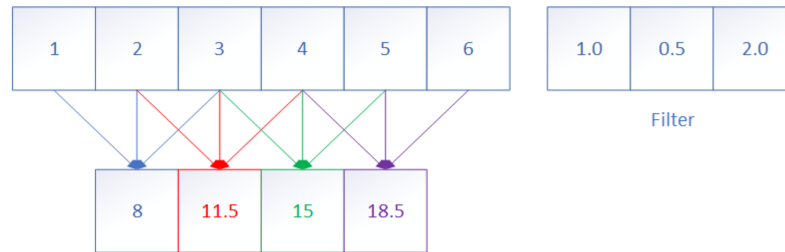


Figure 2.3: Example of a convolution operation. Padding operations on the borders not included.

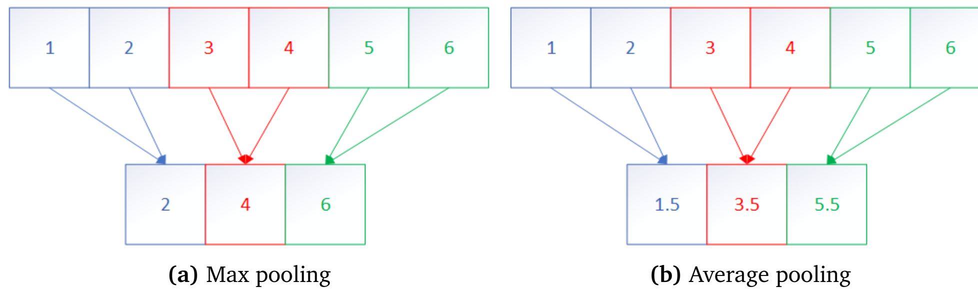
In a convolutional layer, there can be any number of filters, and each of these filters can be different to produce different results from the same data. By combining these different filters, the model can be trained to detect different features in the input data, combinations of which can represent different output classes.

Pooling layer

The pooling layer is a normal part of the complete convolutional neural network. Whether one processes audio or images, the input is often an extensive matrix, while the output is often at most 1000 classes, as is the case with the AlexNet[5] and Inception[8] networks. Since suddenly reducing the matrix down to only 1000 neurons or less would wash out the significance of every single feature detected by the network, pooling layers are applied to reduce the size of the input gradually throughout the network. For this thesis, two different pooling layers have been considered, the max-pooling layer and the average pooling layer. All pooling layers apply a filter similar to the convolution window on the input data. Unlike convolution, in the case of the max pooling operation, the highest value in the window is passed to the next layer. The average pooling layer, on the other hand, calculates the average value in the filter and passes this forward. In addition to doing this, the size of the network is commonly divided by the size of the filter. As shown in Figure 2.4, the input in both pooling types is reduced from 6 values down to 3 in the next layer. While this is common, it is not strictly necessary and can be adjusted freely.

Activation functions

In all neural network layers, the activation function is used to scale the final output of the neuron. As significant variations in the input can give meaningless results to the next layer, following the layer-specific processing done on the input, this input is put through the activation function before being handed off to the next layer.



(a) Max pooling

(b) Average pooling

Figure 2.4: Two examples of the pooling operation on the same initial values

The standard activation function used in most neural network layers is the Rectified linear unit (ReLU) function, which simply sets all input that is negative to zero. All values above zero passed through ReLU are just left as-is. Removing negative values is advantageous when the next layers in the network need to process something while excluding values that may be unimportant to final results from further processing. The ability to exclude negative values is designed to mimic the brain's functionality of only sending a signal once enough energy is gathered in the neuron, mimicking the real world neurons.
In addition to ReLU, the Softmax activation function is often used as the activation function in the final neural network layer. The softmax operation reduces all input to a spectrum between 0 and 1, where all outputs sum together to 1. When used as the activation function of the final layer, the output of the network becomes a probability for what the given output can be.
While Softmax and ReLU are the most commonly used activation functions, any function can be theoretically used as an activation function. The last activation function used in this thesis is the Sigmoid function. The sigmoid function is a bounded, differentiable, real function[10] that also scales its input to a value between 0 and 1, but unlike Softmax, the input value of 0 translates to 0.5 in the sigmoid function. Input over zero is scaled gradually from 0.5 towards 1, while input below zero is gradually scaled down from 0.5 to 0. An example of the formula can be seen in Equation (2.1).

$$f(x) = \frac{1.0}{1.0 + exp(-x)} \tag{2.1}$$

2.1.3 Training the neural network

Creating the neural network structure is only the first step in the process, as the initial weights in the model are utterly meaningless to the desired result. As mentioned in the history of neural networks, backpropagation was already figured out in 1975[2]. For each training pass over the dataset, also called an epoch, the training process executes several vital steps that can be modified to change the weights more optimally to what is desired.

Loss functions

During the training process, the network predicts a value on the given input, and the expected value is also known. A loss function is used to calculate the difference between these two values. When creating a neural network that has to classify the input into different classes, the most commonly used loss function is the categorical cross-entropy loss function[11]. In short, the result of the loss function can be considered the distance between the result predicted by the network and the desired result. Therefore, to achieve the best possible weight combination in the neural network, the goal is to reduce this number as much as possible.

Optimizers

While knowing the loss function number is useful for the human watching the training process, it is the role of the optimizer to take the loss values and turn them into better weights. Optimizer is another word for the stochastic gradient descent algorithm[12], of which there exist many variations. The optimizer takes the loss value from each sample and calculates the most optimal way to reduce the total error by adjusting the weights. As using the entire dataset can be impossible with sufficiently large datasets, and using single samples can produce local minimums, it is common to pass small batches of several dozen samples per batch to the optimizer. To ensure that the weights are not changed randomly, all optimizers use a parameter called the learning rate. The learning rate controls the distance by which each weight can be changed during one training epoch. While the older SGD optimizer only has one learning rate for the entire operation, newer optimizers like AdaGrad, RMSProp and Adam create an adapted learning rate for each parameter. Selective modification of weights allows the optimizers to modify some weights more than others, leaving weights that have little effect on the loss alone while working on the more problematic weights.

Metrics

Finally, various metrics can be reported by the training program to the user, and be used to stop training early. In general, the two metrics used to determine how good a neural network is are loss and accuracy. Loss is the value output by the loss function, and accuracy is the percentage of times the network produced an accurate result. With the loss, the best value is zero, while with accuracy, the goal is to get as close to one as possible. As neural networks can be overfitted for a particular dataset, it is common to use a part of the dataset as a validation set. The goal of the validation set is to also reach as perfect values as possible; however, this dataset is not used during the training process. By excluding part of the dataset in this manner, the second set of metrics is produced with the validation prefix. In addition to the standard loss and accuracy metrics, other metrics can be used, like the top-K categorical accuracy. This variant of the accuracy metrics tracks how often the target value is in the top-K number of targets, rather than tracking how

well the actual target was predicted.

2.1.4 Transfer learning

Once a neural network is trained to do one task, it can often be adapted to do a similar but different task. This process is called transfer learning[13]. In the case of convolutional neural networks, the earlier layers in the model will often pick up generic features in the data, with the more specific features about the output class coming up towards the end of the model.
As the old model is fit to work on its original task, to start the process of transfer learning, it is necessary to replace the classification head of the model that is being adapted. The classification head is the last couple of layers of the network, which is, at minimum, the final output layer. As the weights in the model are already initialized with feature detection, it is necessary to freeze large sections of the network to prevent the model from quickly losing the features in the first few epochs. Once the model reaches the top accuracy for the new problem, the previously frozen layers can be frozen to fine-tune the model to the new task, training with a reduced learning rate to prevent information loss in the new classification head. An example of this process can be found on various documentations of software supporting neural network development, like Tensorflow[14] and Matlab[15].
Assuming that the old and new problem areas allow for transfer learning, the result of transfer learning is a very high-quality neural network model that takes only a fraction of the time to develop.

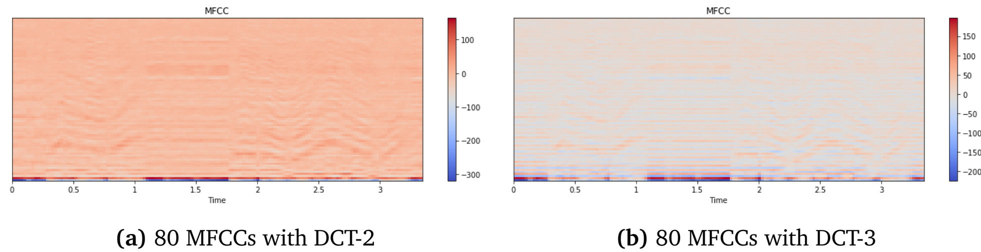
2.2 Audio processing

2.2.1 Fourier transform

As raw audio is not too useful on its own, to make use of the raw audio signal, it is necessary to isolate various parts of the audio into separate parts. As audio is effectively a lot of sines and cosines combined to form a complex signal, these parts of the signal can be separated to find the frequencies that constitute part of the raw audio. The process of extracting these frequencies is computed using the Fourier transform. By extracting each of the signal waves that form the full audio signal, it is possible to analyze how each change over time, identifying the relevant signal waves while also detecting noise in the audio. The detected noisy signal waves can be removed while preserving the vital signal waves for further processing.

2.2.2 Mel scale

Humans hear differences in sound on a different scale than the linear scale of the Hertz frequencies. When the sound is of low frequency, minute differences can be easily detected, while higher frequencies need more significant frequency



(a) 80 MFCCs with DCT-2

(b) 80 MFCCs with DCT-3

Figure 2.5: Two examples of 80 MFCCs generated with DCT-2 and DCT-3 on the same 3.4-second long audio sample

differences to be found by the human ear. To more accurately represent the scale of sound heard by the human ear, the Mel scale was introduced in 1937[16]. By applying the Mel scale to frequency data, the frequencies of the signal can be represented on a more linear scale from the view of a human listener.

2.2.3 Discrete cosine transform

A raw audio signal takes much space and is, therefore, essential to compress into smaller, more relevant blocks of information. Initially developed in the 1970s[17] for use in image compression, it has also seen much use in audio processing. Unlike the Fourier transform that operates on both sines and cosines, the DCT operates only on the cosine. The limitations imposed on the DCT to achieve good compression of the features in raw audio create several assumptions about the input, such as whether the function being transformed is even or odd in the data window being processed. Because of this, a total of 16 variants of the transform exist, of which half are DCT and the other half are the Discrete sine transforms. Of these, the most relevant are DCT-2 and DCT-3, which is the inverse of DCT-2, both described in the original paper[17].

2.2.4 Mel-frequency cepstrum coefficients

Mel-frequency cepstrum is a combination of the above techniques that provide values that are far more analyzable than raw audio. The process of producing the MFC coefficients from raw audio follows the steps defined

1. Process the raw audio signal with the Fourier transform
2. Map the frequencies found in step 1 into the Mel scale
3. Calculate the log values of each Mel frequency
4. Process the Mel log values with the Discrete cosine transform
5. Amplitudes in the spectrum produced by step 4 are the MFCCs

It is possible to use different DCTs to generate MFCC values, as presented in Figure 2.5. One of the benefits of using MFCCs is that the number of coefficients can be scaled as desired. Should one choose to generate 20, 40, or 80 coefficients for a given sample, the first 20 MFCCs in the 40 and 80 options will match the MFCCs

generated in the 20 coefficient option. In Figure 2.5, a total of 80 coefficents are generated. Should only 20 MFCCs be needed, the superfluous coefficients can be removed rather than having to generate a new dataset.

MFCC Applications

One of the recent comparisons between the various audio classification methods has been in grouping audio clips from various entertainment sources into their respective category. A study conducted in 2011[18] aimed to identify if a particular audio sample originated from music, news, sports, advertisement, cartoon or movie. This study compared MFCCs to Linear Prediction coefficients and Linear Prediction Derived Cepstrum coefficients. In the final results of the 2011 study, MFCC has produced superior results when compared to both alternatives, additionally being more superior in helping identify short 1-2 second samples.
Previous students have also used MFCCs at NTNU on similar topics. A former student has used MFCCs to classify audio samples of various marine vessels[19]. The audio samples in this project were sonar data generated using a sonar simulation system developed by Kongsberg Defense & Aerospace. In this project, a small convolutional neural network using the MFCC variant of the data provided the highest accuracy result.

2.3 Hierarchical clustering

Among the current methods of clustering data, hierarchical clustering[20] is a relatively simple but powerful concept. Hierarchical clustering assumes that all data is in some way related to the rest of the dataset. The clustering process starts with assigning a score that represents the distance between each data point. Once the relationship is known, the data is clustered based on the score in one of two different ways. In the first method called «Agglomerative clustering,» the clusters are generated bottom-up, where each data point is its cluster, and the clustering aims to reduce the number of clusters by bringing close data points together. The second method, called «Divisive clustering,» starts with all data being in one cluster, dividing the data into smaller clusters recursively until the desired cluster number is reached. The clustering is illustrated in Figure 2.6. Agglomerative clustering starts on the right and works its way to the left, while Divisive clustering starts from the left and works its way to the right.
Each method of hierarchical clustering has its advantages and disadvantages. The most significant problem for the Agglomerative clustering is the significant performance penalty that requires processing the dataset multiple times. Some variations of the method can improve on this performance drawback to some extent. However, in general, the performance penalty comes from the exhaustive search and checking every possibility of improvement on the resulting clusters. Divisive clustering solves the problem of performance penalties by starting with one big cluster that is split recursively into smaller clusters. The most significant draw-

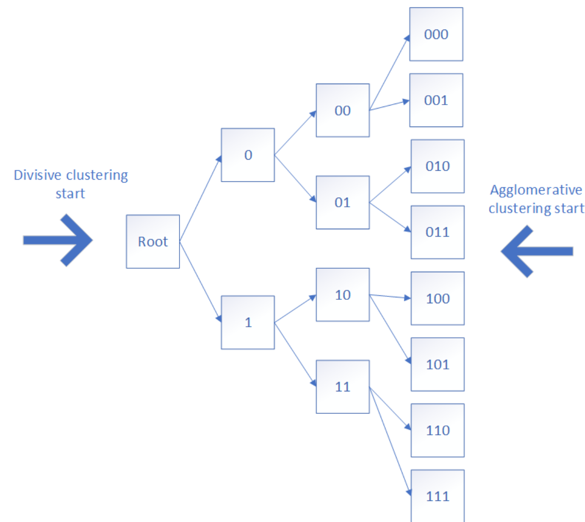


Figure 2.6: Illustration of a data tree, arrows represent starting point for clustering process

back is the potential for more optimal clusters to be available in the sub-clusters, as these will not be checked for potential merges by the algorithm.
One example of agglomerative clustering used in the processing of audio samples is found in an article about a modified Dynamic Tree Warping method used in calculating distance between different audio samples[21]. The authors use agglomerative clustering to compare their modified DTW function with the standard, and then use two different scoring mechanisms to determine how well their function performed. The paper uses two different scoring mechanisms and both top out at around 15 to 20 clusters. Results provided in this paper show potential diminishing results when using more than these numbers of clusters when processing audio.

2.4 Neural networks and data clustering

The concept of using neural networks in tree structures is not new. As NN layers learn information in a structured manner that focuses on more generic feature detection in the first layers, expanding these networks to become more hierarchical classifiers is possible. One example of this is Tree-CNN [22], which in addition to using multiple models in its solution, also implements incremental learning that expands the capability of the model over time, rather than training it on the entire dataset in one go.
The Tree-CNN paper sought to primarily resolve the problem of neural networks becoming final after their initial training. Once the training process is complete, the neural network cannot be generally re-purposed or modified to learn new information, at least not without losing critical information in the model that al-

lowed it to perform well in its previous tasks. By using multiple models in a tree structure, more general classes were created in the root node of the tree that would classify multiple end classes as same, moving the complexity of determining the actual correct class to a more specialized branch node.
In this particular paper, the branches were found to cluster classes in similar-looking groups, even if the classes were not too related to each other. More importantly, in the interest of this master thesis, table 6 of the paper describes the Tree-CNN as being relatively accurate while taking only 60% of the time to construct.
Another case of using data clustering in combination with CNNs is the HD-CNN paper[23] from 2015. The researchers in this paper used hierarchical clustering to cluster the image dataset into more coarse classes. They then used information from those coarse classes in the later layers of the NN that were responsible for the fine classification into specific classes. The top-k error results of HD-CNN were competitive with the first version of Inception[8] and other models, beating them in some metrics.

Chapter 3

Methodology

Development of the hypothesis for this master thesis and the research questions has begun during the summer of 2019, after contemplating the results of the project work done in the Image Processing and Analysis course. Over the following year, various parts of the thesis scope and hypothesis have been adjusted to fit in the allocated time, and based on work done during the thesis. In the autumn semester of 2019, preliminary work was done to assess the feasibility of the master thesis. Based on the results of these feasibility studies, the spring semester work was focused on implementing the iterative re-training process and tree hierarchy generation.
The research questions of this thesis are aimed at the final result at the end of the development process. However, the reasoning behind the decisions taken during development needs to be documented. The development of the code used in this thesis was done using agile methods. After a module was developed, it was repeatedly tested, and based on the results of these tests; the next tests were created.
The following chapter presents the methodology of the development done in this thesis. For each section that has experiments associated with decisions taken during development, the intent behind the experiment is explained. As the thesis does not seek to answer the question of how to develop iterative re-training process or tree hierarchy generation, the experiments done during the development of this thesis are documented in chapter 4.

3.1 Dataset

In all neural network projects, having a big dataset is critical to getting meaningful results. Of course, the dataset also has to be correct for the given problem that is being solved. For this thesis, there was no problem, in particular, that was being solved. Unlike other networks that may need to classify specific samples into distinct classes like AlexNet and Inception, the goal for the networks created in this thesis was to group samples into classes. The result of the thesis was dependent on having a big enough dataset for these groupings to generalize enough. Therefore,

the size of the dataset was the primary concern.
The original, raw dataset in this thesis was the personal media library of the student, in particular the segment consisting of Japanese animated TV shows and movies. As almost all of the media in this dataset contained Japanese audio with accurately timed English subtitles, small samples of audio could be extracted with a matching English translation tied to this sample. Due to the nature of the dataset, the English translation may carry minor artifacts in the label that would make it unsuitable for use in a neural network translator. In addition, longer sentences may have their appearance order reversed due to language differences. However, the labels provide enough information for a human operator to analyze the results of the groupings and present them in the report. In terms of the quality of the audio, some noise is bound to exist in the background. However, the same conditions apply in the real world, adding to the authenticity of the dataset. Measures to limit the potential bad samples in the final dataset are presented in the next subsection.
As the primary priority in selecting the dataset for this thesis was its size, the following list details the size of the raw dataset.

- Video files: 62.508
- Storage size: 32.5 TB
- Video length: Over 1000 days of uninterrupted video

The dataset was acquired over several years, mostly through the private torrent tracker AnimeBytes[1]. Content on this tracker is curated by its users, leading to a vast library that can be relied on. Multiple versions of a particular series can exist, of which some may be ripped from a blu-ray disc while others may be downloaded from web streaming services. As the dataset was manually downloaded, each series can be expected to be of the highest quality that was available at the time of the download. Thus, it can be moderately depended on for use as the dataset in this thesis.
There are some legal considerations needed given the nature in which the dataset has been acquired. The entire dataset has been acquired through the use of torrents on public and private trackers throughout the last couple of years. For this project, the video/audio content has been stripped to audio-only and cut into small several second long pieces. Afterward, the audio was converted into different formats through a lossy process meant for use in neural networks. The neural networks produced from this dataset during this master thesis are also not planned to be published. Given this, it is considered that the legal considerations are not significant enough to prohibit the use of this dataset in the master thesis.

3.1.1 Processing steps

The dataset was processed in four stages to prepare the raw dataset for use in neural networks. Each of these stages represents a step that the dataset has been

[1] `https://animebytes.tv/`

processed in, removing potential bad samples and selecting the correct format for the final dataset.

Stage 1

The first stage of the dataset processing was to extract the audio and subtitles from the media library. Standardization of the formats was a central part of this process, as the media library had a wide variety of video, audio, and subtitle formats. While some of the dataset was relatively new and used modern formats, some of the files had used more ancient formats that have not been used much for well over a decade. For this step, FFmpeg was used for its compatibility with a vast number of formats that would be able to process the dataset more or less completely.
Most of the storage used on the dataset was expected to be in audio. Therefore compression of the audio segment was done in this stage. The push for compression was fueled in part since parts of the dataset, movie blu-ray rips in particular, used multi-channel lossless FLAC formats for the audio track. In some cases, the audio track alone was over 1 GB in size for little less than 2 hours of audio. To compress the audio, the OPUS codec[2] was used. OPUS was selected for its superior quality over other codecs[24]. To ensure that data loss due to compression would be kept to a minimum and predictable, a constant bitrate of 128 kbit/s was used. While standardization of audio content was easily selected, differences were more significant in the standardization of the subtitle content. Throughout the years, many different formats have been used to attach text content to videos. In the past, subtitles were often attached as separate files with the same name as the video file. More recently, the Matroska container has allowed for subtitles to be combined with video files for a smoother distribution of content. These subtitles could be elementary lists of lines with just a timestamp and the subtitle. In the anime community, a separate dedicated subtitle codec has been used, called Advanced SubStation Alpha[25] (ASS). Unlike the more primitive standards that attempt to only present the text in a simple, clear manner, ASS files can contain formatting and styling information to be rendered along with the video, providing among other features font and karaoke styling. As information about styling could be used in the later stages to filter undesired text, ASS was selected as the standard subtitle format.
Unfortunately, on this stage, a significant part of the dataset was written off as unusable. While conversion of all text-based subtitles had been successful, around 20% of the dataset used image-based subtitles like VOBSUB or HDMV-PGS, common standards used in DVDs, and Blu-ray discs. Initially, the extraction of these subtitles failed silently due to a configuration error in FFmpeg, not specifying these particular standards. After extraction of subtitles using these formats, a minor attempt using OCR software called "Subtitle Edit"[3] was conducted to convert the subtitles to text. The results of these attempts were unsatisfactory, with too many

[2] `https://opus-codec.org/`

[3] `https://www.nikse.dk/subtitleedit`

artifacts in the few samples processed to be considered reliable for further use. Besides, this process would take too long to process over 10 thousand files, with one file taking more than a couple of minutes.
Ultimately, this stage produced roughly 1.6 TB of compressed audio and subtitle files.

Stage 2

Following the standardization of the data formats, the second stage of the dataset processing sought to cut the audio content into samples based on the timestamps in the subtitle tracks.
To extract the small audio samples, selecting the appropriate audio and subtitle track is necessary. While most of the dataset contained strictly one audio and one subtitle track, some files contained multiple audio tracks, and others contained multiple subtitle tracks. In most of these files, the Japanese language flag was used to identify the audio stream, and the English language flag was used to identify the subtitle stream. Some of the streams had multiple Japanese audio streams, and a lot more had multiple English subtitle streams. In the case of the audio streams, most of these extra streams carried extra ID flags like "Commentary" that allowed those streams to be filtered. If the stream could not be filtered, the first stream was selected.
Subtitle streams were a more complicated process. Some of the extra subtitle streams could also be related to the aforementioned "Commentary" streams and were subsequently filtered out. Other streams were dedicated sign and song streams, often used in conjunction with files that carried both Japanese and English audio. Many shows that have English dubbing retain their original Japanese intro and outro songs, in addition to specific sign translation, which these extra subtitle streams provide translations for. As filtering out this content was essential to improve the quality of the final dataset, these tracks have been used to filter matching subtitles in the bigger subtitle file from the dataset.
While extra streams were useful in identifying some of the bad samples, most of the subtitles required more analysis of the subtitle stream itself. Following the ASS specification, all subtitles that could be a relevant audio sample are likely to carry the "Dialogue" style option. Unfortunately, analysis of some files showed that, in some cases, other tags were used for the relevant lines. Because of this, a black flag approach was used to remove irrelevant samples. Subtitle lines using the "ED," "OP," "Sign," "Song," "Comment," and "Logo" style names were removed from the dataset.
In addition to these bad samples, the ASS specification permits creators to put in all sorts of visual effects in their subtitles. These visual effects are also present in the same subtitle file and had to be removed before the dataset could be considered usable. As styling information in the ASS file uses many modifiers using the
character, lines that included more than two of these modifiers were removed

from the dataset before sample extraction. Newlines were excluded from this process as exceptions, being replaced with spaces before the check took place. Upon analysis, some relevant subtitle lines used more than two modifiers to position the text, but it was concluded that this sample loss would be insignificant.
All subtitle lines that passed all checks were extracted from the audio stream to a ramdisk, and once the entire subtitle file has been processed, the resulting subtitles and labels were zipped and saved to disk. A total of 45.732 media files had at least one relevant audio sample for the final dataset, with the storage size being 532.4 GB.

Stage 3

Once data has been split into appropriate fragments, it was necessary to convert it into a format that could be read by a neural network. Raw audio feeds contain a lot of noise and redundant data that is not critical for analysis but could also cause the neural network to reach wrong assumptions about the dataset. The dataset has been processed with the Python library Librosa to remove irrelevant noise and strengthen the values of significance.
Librosa supports a wide variety of different feature detection functions. To determine which of these functions would be best, and with which parameters, experiment 1 has been run. The purpose of this experiment was to verify that a convolutional neural network can reach high levels of accuracy in classifying the processed audio samples. Should it be impossible for a neural network to reach any proper levels of accuracy for any of the data types, this stage would already reveal problems.
Also, it served as a selection process for the third stage of the data preprocessing stage of the thesis. Each function type in Librosa takes time to process, not to mention the time it takes to read in a sample from disk. During performance measurements, it would take roughly 0.5 seconds to read in an audio sample, and 1.5 seconds in total to read in and process it using all considered function combinations. Conversely, it would take 0.7-0.8 seconds to read in an audio sample and process it using only a select few functions. As each neural network also has to be adapted to the dataset format it has to process; it was unlikely that more than one function combination would be used. However, the time penalty of reading in an audio sample meant that if the selected function combination were problematic, it would take an excessive amount of time to prepare another dataset from this stage.
Two actions have been taken to reduce the number of samples. Firstly, samples that were registered as having an overlap with another sample were removed at this step. Having an overlap with another sample is likely to indicate two characters talking over one another, or a song playing in the background. As these potential errors are trivial to detect and did not constitute a significant portion of the dataset, they were reduced with minimal impact on the dataset. Secondly, the length of the samples was constrained to between 0.7 and 6 seconds. Convolu-

tional neural networks have a defined input size, meaning that a maximum input length had to be picked. A sample longer than 6 seconds is likely to be a very long sentence, or contain some non-dialogue content. In some cases, some samples could span the entire length of the original file. Similarly, samples shorter than 0.7 seconds are likely to be either quickly spoken single word phrases or formatting options that succeeded in passing the previous checks. If these samples were actual words, there was a risk for these samples to be mistimed, losing a significant portion of the phrase, if not all of it. While the timing error could also happen in longer samples, as sample length goes up, the effect of a minor mistiming goes down, and therefore this was not an issue in longer samples.
For the decision on which function combination to pick, a compromise decision was reached based on the results of experiment 1. Both MFCC function combinations have been generated, with the MFCC generated with DCT 3 being considered for use as the primary dataset. As the MFCCs scale linearly, a total of 80 coefficients have been generated for each sample. If the number of coefficients turned out to be too high, the extra values could simply be discarded, while if more were needed, the entire dataset would require reprocessing.
In addition to the two MFCC combinations, all three of the Constant-Q chromagram combinations have been generated as well. This decision was made based on both satisfactory performance of these combinations, and the need to have a secondary dataset generated should the primary prove to be useless beyond the first experiment. The size on disk for all three of these combinations was equivalent to an MFCC combination with 84 coefficients, meaning the storage penalty for this decision was insignificant. Spectral contrast has also been chosen as a secondary dataset, in particular, the version using the FFT window size parameter of 4096. This variable would also take an insignificant amount of storage and provide extra stability in the event of failure.
Following the processing of stage 3, the dataset took over 1.5 TB in storage space, containing around 12.6 million samples. Both MFCC combinations took 393 GB of storage, the three chromagrams took 60, 118, and 236 GBs, and the two spectral contrast variants took 35.2 GB. Most critically, this stage was the longest to process, taking over two weeks of continuous processing during the Christmas break on the primary hardware, in addition to recruiting some help from extra hardware from relatives.

Stage 4

All of the previous stages have generated samples based on a path mimicking the original dataset. While useful for organizing data in the preprocessing stage, this had to be corrected in the final stage.
In addition to moving all samples into one directory, it was also necessary to normalize the dataset values. Neural networks create their conclusions based on the variation between the input variables. If these values differ too much, the network could overfit and conclude that each sample fits its class based on some arbitrary

input number that just so happens to match in all samples processed at the time. Given that at the dataset size was too big to create a normalization process over the entire dataset, a per-sample normalization was applied.
The process of normalization was done in two steps. First, the mean and variance of the sample were calculated. Then, Equation (3.1) was processed. To prevent a division by zero, an epsilon variable was added to the equation with the value of 1e-12.

$$norm_sample = (sample - mean)/\sqrt{variance + epsilon} \tag{3.1}$$

Similar to the previous stage, generating multiple versions of this stage was relevant from the safety standpoint. While the primary goal of the thesis was to process the entire dataset, the difference in length between all samples could prove too big to overcome. Three subsets of the dataset were generated along with the full, ready for use dataset to enable some degree of freedom in selecting the right dataset. These three versions carried only short, medium, and long sample lengths. As all of the generated dataset samples carried the length of the sample in the form of their row length, this length was used to differentiate the samples. The short dataset carried only samples with row lengths between 30 and 100, representing samples between 0.7 and 2.3 seconds in length. The large dataset carried samples that did not fit in the short dataset, with row length between 100 and 260, representing samples between 2.3 and 6 seconds in length. The medium dataset carried samples with row length between 60 and 130, double of the minimum length and half of the maximum length, representing samples between 1.4 and 3 seconds in length.
Of all dataset preprocessing stages, this stage was the shortest and allowed for some leniency in potential errors appearing in the normalized versions of the results. Stage 4 has only taken 24 hours to prepare all four versions of the final dataset. As the variables in the normalized dataset are of the same size and length, the full dataset was of the same size and sample number. The large-only dataset version contained 5.9 million samples, while the small-only dataset contained 6.7 million samples. The medium-length dataset contained roughly 6.8 million samples.

3.1.2 Statistics

Overall, a significant amount of time has been spent in processing the samples down to a usable format. Except for some Tensorflow functions in Stage 4, all of the processing was strictly CPU-bound. As detailed in subsection 4.1.1, the CPU used in most of the processing is a quad-core from 2015, which was very limiting for this task. A newer CPU with more cores could likely handle this task better, given that the task of processing the dataset scales pretty much linearly with the number of cores that can be thrown at the problem. In Table 3.1, the time spent on processing each stage is listed.
During the processing in Stage 2, the total number of samples present in the dataset became known. From the 54 million subtitle lines, around 40 million were de-

Stage	Time taken
Stage 1	5 days
Stage 2	5 days
Stage 3	16 days
Stage 4	24 hours

Table 3.1: Time spent processing the dataset

termined to be bad and removed at that stage. Most of those 40 million samples could be found to be between 0 and 1 second in length, clearly representing some form of formatting option that was used on the line. Of the remaining 14.5 million samples, roughly 700 thousand overlapped another sample to some extent, while the rest that did not make it into the good group failed some of the lesser checks in the code. Following the time limitation on the dataset, the final number was brought down to the final 12.6 million. A table detailing these stats can be found at Table 3.2.

Samples	Count
All lines	54 599 719
Removed	40 087 565
Total	14 512 154
Good	13 217 123
Overlap	723 152
Final	12 625 000

Table 3.2: Number of subtitle lines over the course of processing the dataset

From the total dataset mentioned in Table 3.2, a graph showing the distribution of the samples is listed in Figure 3.1. A small rise in the sample length can be seen at the beginning of the figure, which can be attributed to formatting options that evaded deletion during processing. The bulk of the dataset can be found in the 1-2.5 second range, with the rest of the dataset slowly descending in size as sample length increases. This graph is an expected result, as most subtitle lines with dialogue are expected to translate the speech in the video without cluttering the whole screen with text.
As correcting for potential errors has been crucial in the dataset preprocessing, the overlapping samples were analyzed for their characteristics. The Figure 3.2 shows the length of these overlaps, both in time and as a percentage of the sample. Most of the overlapping samples had overlapped another sample completely while simultaneously being very short. Another peak can also be seen at the beginning of the percentage graph, indicating that some samples were overlapped minimally. Both peaks were expected to have been mostly caused by formatting and other non-dialogue content in the subtitle files. Due to the nature of the data in the percentage graph, one overlap causes two results to appear in the graph as sample

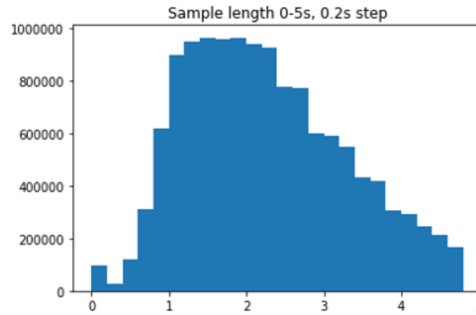
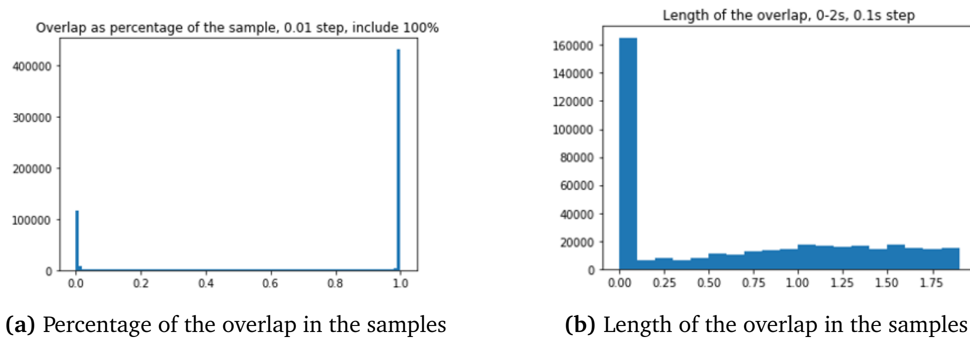


Figure 3.1: Length of the samples in the dataset



(a) Percentage of the overlap in the samples

(b) Length of the overlap in the samples

Figure 3.2: Overlap statistics from the samples that had an overlap recorded

A overlaps sample B and vice versa.

3.2 Neural network

During the course of this thesis, multiple neural network structures were tested with multiple parameters to determine the best network to use on the dataset. As at the beginning of the thesis, the experience of the student with developing neural networks was negligible, which made this development process useful in both an educational and exploratory purpose. Each of the subsections mentions the methodology used in an associated experiment.

3.2.1 Early development

Early development of the neural network model has been a very chaotic process of learning new features of neural networks, applying them to the first experiment, and seeing what would stick. During this time, the Classification of Marine Vessels[19] thesis written by a fellow NTNU student served as a helpful guide in using neural networks for audio processing.
The first neural network model in the thesis consisted of three dense layers, followed by the final dense output layer. Unfortunately, this model never achieved

any stable results, and rarely achieved results better than the random classification used as the control group in section 4.2. Following repeated failures, the three dense layers were replaced with one-dimensional convolution layers, all containing 64 filters and a filter size of 3.
Basing the network structure on the AlexNet[5] and Inception[8] papers, the early model has also included a single classification layer towards the end of the model. Since unlike Inception, the network data was not flattened during data processing throughout the network, a flattening layer was included before this classification layer.
Multiples of 1000 samples were used in the early development of the neural network model, based on the number of classes used in the papers mentioned above. As the goal of the thesis has been to group audio samples autonomously, a dense layer representing the output layer was added to the network, with a neuron count of 10, followed by another with a neuron count of 1000. The decision to spread the 1000 samples across ten classes was made based on the related work[21], excluding more than 15-20 classes from being used. The ten classes were picked to make the division into classes more simple.

3.2.2 Extra layer

As the early model was only designed to handle 1000 samples, and it did not carry any capacity to grow to learn more, a modification was done to the final output layer of the network. Instead of 1000 neurons, the network was expanded to carry 1010 neurons, with the final ten being manually modified to transfer the results from the softmax layer directly to the final output layer.
An illustration of this can be seen in Figure 3.3.
By manually modifying the weights to translate the results of the network to the last layer, parts of the dataset that have been classified already can be used by assigning them to the last ten classes. The manual weights in the layer prevent any significant modification from being done by the training optimizer to the classifications of these last neurons while leaving the first 1000 free to be changed. The only values that the optimizer can use to classify the new dataset are the values provided by the softmax layer, meaning that the samples have to be assigned by the optimizer to be one of these ten classes. As the new dataset is trained on with the old dataset, the weights cannot be adjusted to the point of overfitting by the new dataset. However, some adjustment is desired, as these adjustments can be picked up on by the iterative re-training process to improve the neural network as a whole.
An experiment detailed in Section 4.3 was conducted to verify this layer in practice

3.2.3 Final model

Following the development of the extra layer, the final neural network model for this thesis was developed. Since the earlier model was built very quickly to make

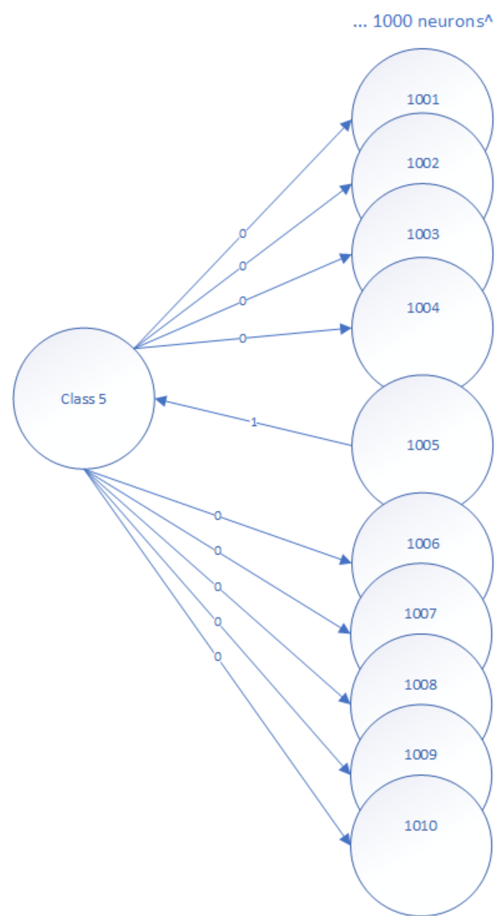


Figure 3.3: Example of the 1010 neuron layer

decisions about parts that were needed to develop the final model, with those parts in place, the final model could be put together.
Operational variables were identified in the various components used in a convolutional neural network to develop the neural network model that would be suited for the task it would work on, As running all of the possible variable combinations would take forever, determining the correct number of convolutional layers was deemed the top priority. The full details on the process of selecting the optimal network configuration are listed in Section 4.4.

Stabilizing the result

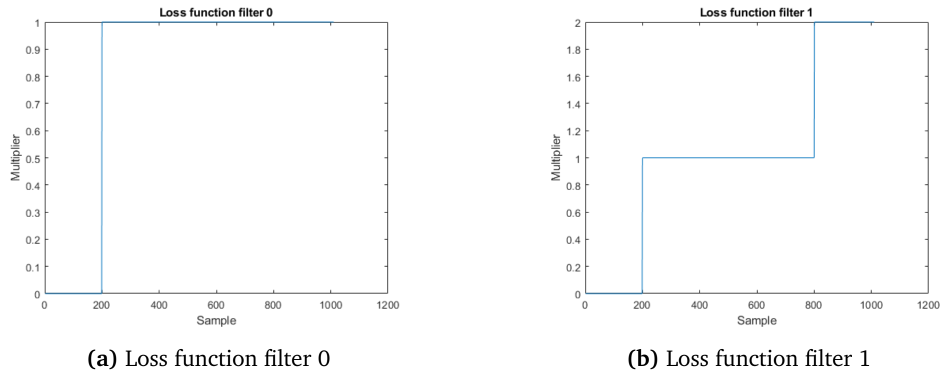
In the beginning, it was intended that the neural network would do its classification from the very first iteration, generating the initial clusters by itself. While it was determined that the network is capable of doing this in Section 4.2 and Section 4.3, the results were not stable and predictable. Each run on the same data would return a different cluster, meaning that comparing two different configurations of neural networks would be mostly meaningless. The first cluster has been from this point onward generated using the Scikit-learn Agglomerative clustering function to bring stability to the network, and to make the results more predictable.

3.3 Loss function

The loss function modifications have been a central part of the development part of the thesis. As the custom layer has proven in Section 4.3 that a custom part can significantly improve the performance of the training process, the same has been assumed for the loss function modifications.

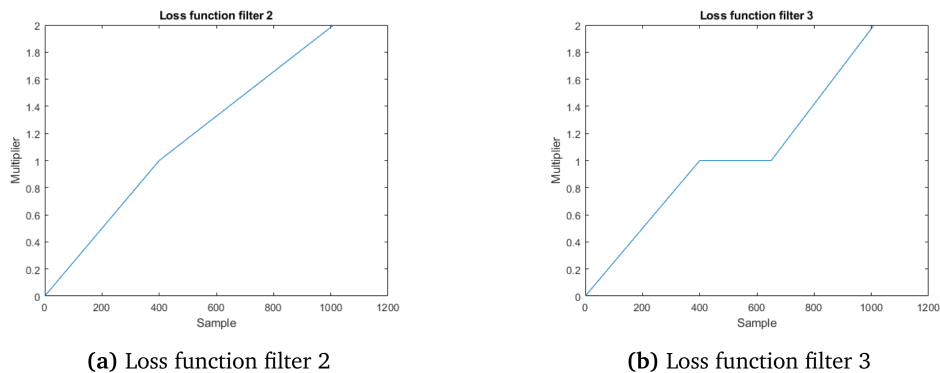
3.3.1 Primary filter

The filter in the loss function targets the results of the network after a top-K function has sorted them. To prevent the filter from damaging the result of the current class, that class is excluded from the filter. As the results are sorted highest to lowest, the filter adds an extra penalty to the result by multiplying the result with itself. In the case of the highest values, the penalty is inverted, as the cost of errors is significantly reduced. The goal of the filter was to incentivize the network to cluster samples more equally by penalizing massive clusters and rewarding the smaller ones.
Figure 3.4 and Figure 3.5 show the filters used in the thesis, following the results of Section 4.5. Filter 0 (Figure 3.4a) is a simple forgiveness filter that removes the penalty of the first 200 samples. Filter 1 (Figure 3.4b) is a similar filter, except the final 200 samples are penalized doubly. Filter 2 (Figure 3.5a) is the first gradual increase filter, where the filter gradually progresses towards a defined point at a constant rate, and then changes the rate once somewhere in the middle of the



(a) Loss function filter 0 **(b)** Loss function filter 1

Figure 3.4: Loss function filters 0 and 1



(a) Loss function filter 2 **(b)** Loss function filter 3

Figure 3.5: Loss function filters 2 and 3

filter. Filter 3 (Figure 3.5b) is based on a similar concept as filter 2; however, it also has a shelf similar to filter 1 in the middle of it.
The first two filters are meant to investigate the steep increase in penalties and rewards, while the last two filters are meant to investigate the gradual increase in penalties and rewards.

3.3.2 Second filter

In addition to the primary filter, another filter has been added to the loss function. As the last ten neurons are not the normal sample neurons, these need to be treated differently. The most substantial value among these ten neurons has its loss function cost removed, to guide the network to group samples according to these groups. A sub-experiment in Section 4.5.1 has aimed to add an extra cost to the remaining nine neurons that would follow after the first. However, the experiment has yielded inconclusive towards negative results as to the usefulness of this second filter increase. Thus, the second filter only removes the first of the ten final neurons from the loss function cost.

3.4 Iterative re-training

Iterative re-training has been considered critical to the thesis due to the sheer size of the dataset. While the dataset could be reduced to fit more efficiently during the training process, it would ultimately be a tiny percentage of what was available. A good part of the value in the research done in this thesis is the large dataset involved, and therefore it was critical to at least attempt this.
Given the objective of the thesis, iterative re-training was considered to be the right choice regardless of the dataset size. While one can always achieve a better cluster distribution if one throws more data at the algorithm, odds are the cluster generated with only a couple thousand samples will be relatively close, at a fraction of the computational cost. Additionally, the data that does not fit well into any of the existing clusters is likely to have the most significant effect on the network, meaning that these are the samples that carry the most substantial significance.
The goal of the iterative re-training process was to start with a minimal dataset and work through the dataset, accumulating new knowledge as more data is parsed. As more data is processed, more samples appear out of the norm and are absorbed into the training dataset. Over time, the new samples would give insight into better clustering adjustments that would prevent some of the samples that were yet to be processed, to be inserted into the dataset.
A parameter has to be picked that will determine if a sample is important enough to be considered for use in the dataset, or if it should be ignored to start the process of iterative re-training, In the thesis report, this parameter is referred to as the "Iteration threshold." If the iteration threshold is set to 0.8, for example, any sample that is classified higher than the threshold will be ignored for use in the dataset, while samples classing lower will be added to the dataset.
Following the decision on the iteration threshold parameter, the process of iterative re-training follows these steps:

1. Train the first network using a pre-determined method
2. Classify samples until the minimum amount of valid samples are detected
3. Re-classify the samples using the selected process
    a. Samples can also not be re-classified. Instead one can use the results of the first classification
4. Train the network using the bigger dataset
5. Return to step 2 until the entire dataset is processed

The experiment detailing the practical parts of the iterative re-training can be found in Section 4.6.

3.5 Tree generation

Similar to iterative re-training, the tree generation experiment has been central to the thesis. By using the iterative re-training in a tree hierarchy, the goal is to

eventually track down a particular audio sample in the dataset. For the purposes of this thesis however, the precision of the classification is reduced somewhat.
Based on the results of Section 4.6, the iterative re-training process is intended to be used to train each node in a tree containing neural networks. The objective is to group samples that are similar to each other together in a computationally cheap manner. Once the tree would grow big enough, each node could receive a much bigger neural network to solve its part of the entire problem. The current alternative is to use the same bigger network to solve the entire problem, which quickly fails to achieve satisfactory results due to the problem becoming too complex. A small library of 2000 words is far easier to remember than a library with hundreds of thousands of words, in addition to all possible forms and permutations of each word.
The practical aspects of creating the trees in this thesis are specified in Section 4.7.

Chapter 4

Experiments

During the work on this thesis, various experiments have been conducted to find the correct parameters to use in subsequent development. The following section provides details on the experiments done during this thesis to explain how the final results in the thesis have been achieved.

4.1 Tools used in the thesis

The following section lists the hardware and software used during the experiments in this master thesis. It also lists tool-specific findings that are not relevant to mention in the other sections.

4.1.1 Hardware

A table with the primary hardware used in this thesis can be found in Table 4.1.

Part type	Name	Main speed	Total memory size
GPU	RTX 2080 Ti	1.65 Ghz	11 GB GDDR5
CPU	Intel i7 4790k	4.00 GHz	32 GB DDR3
Motherboard	Z97X-UD5H-BK		
SSD	2x Samsung 860 EVO	540 MB/s	2TB
HDD	5x 12TB + 1x 10 TB	160 MB/s	70 TB
Off. HDD	5x 4 TB	100 MB/s	20 TB

Table 4.1: Hardware specs

In addition to the primary hardware, in several steps, processing was accelerated using other available PCs. While helpful, the presented hardware has proven itself to be much faster than this alternative hardware, as noted in experiment 1 (data processing) and experiment x (iterative train root node).
The primary workhorse of the thesis is the GPU, the RTX 2080 Ti[26]. When the GPU is under minor load, the default clock is lower than advertised, at 1350 MHz.

The lower clock is due to the adaptive overclock features of the GPU. Since the neural network processing does not utilize all GPU functions, and smaller models do not use the GPU to their fullest, the operating system lowers the clock speed to reduce heat. To circumvent this and use the hardware to its fullest extent, one can start multiple Python consoles and train multiple neural networks in parallel. In the case of the RTX 2080 Ti, up to 5 simultaneous threads have been used stably, with some limitations. During full utilization of the GPU, clock speeds above 1920 MHz have been recorded, while maintaining temperatures below 60 degrees celsius. As at that point, resource utilization is at its fullest; network training time is increased for each individual thread. However, the total processing time is still reduced as hardware is utilized more overall.
One problem that has arisen during development was the lacking capabilities of other hardware when compared to the GPU. The RTX 2080 Ti was purchased in late 2019 for the thesis, while the CPU and RAM were purchased in early 2015. Running multiple Python consoles includes having the cache for each console in RAM. Python is not a memory-optimized scripting language, and uses much space for each variable in memory[27]. While the amount of RAM available on the hardware is above the standard of desktops built even today, maxing the capacity of the motherboard, some experiments have still been limited in running more simultaneous processes due to RAM shortage. Also, the limitation of the quad-core nature of the CPU has been a limiting factor in the more CPU intensive tasks in the thesis, most notably experiment 1.Tasks that only run on the CPU like data preparation and even python consoles themselves during neural network processing need a minimum amount of CPU resources to function properly. Even if the RAM issue was resolved, the limited number of CPUs would block more than 5 python consoles running simultaneously.
In terms of findings on storage and caching, when using Pythons pickle library, storing the dataset on an SSD will improve read times by an estimated 35% over using HDDs. According to a speed test done on the hardware, the read-time of an SSD is around 540 MB/s, while the read-time of the tested HDD is around 160 MB/s. Therefore, the benefits of moving the dataset to a faster medium is entirely dependent on the size of the dataset, and frequency of changes to the dataset. In the case of this thesis, experiment x (root node) used the SSD as a cache for the dataset.In contrast, experiment y (trees) used almost all available HDDs for the dataset cache, excluding the external drive.Given the nature of the last experiment, each experiment variation would need a full copy of the dataset. Also, these copies of the dataset would be repeatedly created and deleted, adding to the wear of the drive. Each experiment variation was given a dedicated drive to alleviate the potential performance loss due to drive seeking present on HDDs, with the results also being printed to a dedicated drive. As mentioned, the external drive was not used as a dedicated drive, due to lower performance compared to the internal drives operating over SATA-3.
It is also important to mention the massive data storage pool of the hardware. Thanks to the available storage, each step of the dataset generation could be stored

for safe-keeping. Large amounts of free storage also enabled various storage-expensive methods of caching, as mentioned above. Also, multiple older HDDs have been used as backups and portable storage for the dataset.

4.1.2 Software

The development platform used during this thesis has primarily been Python on a Windows 7 computer. Table 4.2 details the different versions of the software used for development in the thesis.

Name	Version
Anaconda	2019.10
Python	3.7
TensorFlow	2.0, 2.1
TensorBoard	2.0, 2.1
Librosa	0.7.2
Scikit-learn	0.22.2
Spyder	3.3.6, 4.1.2
PowerShell	5.1
FFmpeg	4.2

Table 4.2: Software versions

Python environment

A Python environment requires a package manager to use it to the fullest extent. Visual Studio that the student has already had installed on the computer also included a full Python environment with Anaconda[1]. While the included version has proven itself to be unreliable and required multiple re-installations, certain features have proven themselves to be necessary throughout this thesis. The most crucial feature that prioritized Anaconda over picking pip was the ability to create separate virtual environments with different versions of packages installed simultaneously. Individual packages often require specific versions of dependencies that may not be available anymore, in which case an outdated version of the package may be installed without notice to the user.
One case of the separate environments being critical for this thesis was the Librosa[2] library. Librosa is a python package for music and audio analysis, which this thesis used to process the dataset with the various feature extraction methods that Librosa supports. Deep in the dependency chain for Librosa, a dependency conflict arose that forced Anaconda to install version 0.6.3 of Librosa, while the latest current version is 0.7.2. As the feature extraction methods are based on scientific algorithms and thus should not change between versions, each new version of

[1] https://www.anaconda.com/
[2] https://librosa.github.io/librosa/index.html

a package can include more methods that the user expects to have available. A separate environment dedicated exclusively for Librosa had to be developed to handle this dependency conflict to resolve the matter in the thesis.
The Tensorflow[3] package was selected for the development of the thesis to create and train neural networks. As Google develops TensorFlow, one of the leading firms in Artificial Intelligence research that also employs researchers that published the Inception paper[8], it was picked as the superior choice. Shortly before the thesis project started the development of neural networks, version 2.0 of Tensorflow was released. This thesis used version 2.0 at the beginning of the thesis, updating to version 2.1 in the middle of the thesis. Version 2.2 was released in the final month of the thesis. While certain features that were released in this version would be exciting to use during development, it was not used to maintain the stability of the thesis results.
As noted in the hardware section, multiple Python consoles have been used to train multiple neural networks simultaneously. To achieve this, Tensorflow needs to be configured only to use a limited amount of memory on the GPU. Under regular operation, Tensorflow will attempt to use all available memory for itself, which will lead to potential system instability and hanging even if only one network is trained at a time. Because of the recent update from 1.x versions of Tensorflow, a compatibility layer needs to be used in current versions of Tensorflow as the equivalent functionality has not been found in versions 2.0 and higher. The memory fraction parameter has to be changed to the desired level to adjust the maximum memory used by Tensorflow, code for which is provided in the code listing below. It is important to note that the memory fraction does not represent the actual memory fraction used by Python, and is a significant percentage higher than the parameter set in the code. A tool provided by Nvidia called Nvidia SMI had been used to monitor resource utilization during the development of the minimal parameters that could provide a stable and fast environment.

Code listing 4.1: Code to reduce memory usage of Tensorflow in a single console, valid for Tensorflow 2.1

```
#Tensorflow config to prevent lag
config = tf.compat.v1.ConfigProto()
config.gpu_options.allow_growth = True
config.gpu_options.per_process_gpu_memory_fraction=0.10
session = tf.compat.v1.Session(config=config)
```

The Tensorboard[4] package was used to process the results of the various experiments detailed in the later sections of this chapter. Tensorboard is a visualization toolkit developed for Tensorflow. The primary feature used in the toolkit was the ability to record, view, and sort individual neural network tests. As Tensorboard integrates easily in Tensorflow, this allowed for agile development of the test code, where tests were rapidly coded, run, and analyzed. However, given the sheer amount of tests run during this thesis, the thesis has stumbled across a critical

[3] https://www.tensorflow.org/
[4] https://www.tensorflow.org/tensorboard

performance bug. Tensorboard attempts to record the state of the neural network as it is trained, and recreate a visual graph for the training process. While this may be a useful feature for other projects, it was not relevant for this thesis, while simultaneously causing some experiments to max out the RAM on the computer. The features that cause this bug to manifest can be disabled by using Anacondas' capacity to create custom environments, creating an environment exclusively for Tensorboard. Other packages can also be installed in this environment; however, Tensorflow has to be explicitly excluded from this environment. Doing so causes Tensorboard to enter a restricted feature mode, which allows for much larger tests to be viewed with minimal lag.
In addition to the above packages, several other packages have been used extensively in the project. To create the hierarchical clusters for the first clusters, scikit-learn[5] has been used. Scikit-learn provides a lot of various functions that are useful in neural network development. For the IDE, Anaconda comes preinstalled with Spyder[6], which was used for the development of the thesis. Lastly, the pickle package in Python was used to save results to disk, along with handling cache logic.

Other

In addition to using Python during most of the development, Powershell was used in the early stages of dataset preparation. Powershell was chosen for these first tasks due to earlier experience in using Powershell to maintain the media library used as the dataset in the thesis. As Powershell would be satisfactory in the first tasks, it was chosen to be used over Python due to students lacking experience in using Python at the time. Since the data extraction script was intended to be run only once, writing the scripts quickly and starting to develop Python code was preferable, especially given the expected time to process the dataset. The Powershell version used in the thesis was 5.1.
The media converter FFmpeg[7] was used to facilitate the processing of the dataset into a standardized form. FFmpeg supports a wide variety of video, audio, and subtitle formats. The ability to process any input was critical when various video files in the dataset had an unknown number of codecs used to encode them. Some of the more recent media may use more modern formats, while older media may use some more obscure format that other tools could fail to process. As each sample in the final dataset needed to carry only the particular subtitle fragment, the ability to specify file length options with a command line was also necessary for the selection of the conversion tool. Since losing samples due to shortcomings in the selected tool would be undesirable, FFmpeg fit the requirements the most. The FFmpeg version used in the thesis was 4.2.

[5] https://scikit-learn.org/stable/index.html
[6] https://www.spyder-ide.org/
[7] https://ffmpeg.org/

4.2 Dataset selection

The goal of this experiment was to investigate if neural networks could produce any relevant results for the thesis and determine which of the function combinations would be best suited to proceed with.

4.2.1 Experiment setup

The first experiment revolved around a minimalist neural network to select the appropriate function combination to use in Librosa to generate the dataset for the rest of the thesis. A table detailing each layer in the network can be seen in Table 4.3.
Three one-dimensional convolution layers start the network, followed with a standard max pool layer. An operational flattening layer is added to make the layers fit into the dense layers. One dense layer is added before the final two layers, to serve as an abstraction of the input data. Then, a dense layer with the softmax activation function is added to the network. The softmax layer serves as the classification layer, forcing the network to pick one of the neurons to classify the sample into. An extra dense layer is added with the sigmoid activation function to represent the dataset This layer represents the dataset used in this experiment, and each sample maps to their neuron in this layer.

Layer type	Parameters	Activation
Conv1d	64 filters, size 3	ReLU
Conv1d	64 filters, size 3	ReLU
Conv1d	64 filters, size 3	ReLU
Max pool	Size 2	ReLU
Flatten		
Dense	250	ReLU
Dense	10	Softmax
Dense	1000	Sigmoid

Table 4.3: Early neural network model

The process of getting the results can be summarized as follows:

- Train a fresh network on the entire dataset.
- Remove the last layer
- Run a prediction on the dataset
- Generate a new instance of the network, minus the last layer
- Train the network on the dataset using a 70% train and 30% validation split

By creating a new network and training the network based on the results of the first, the goal was to determine if the network was able to find any features to group samples autonomously. To control for potential overfit errors, in addition to the standard validation loss and accuracy values, the number of classes and the

size of the largest class was tracked. Therefore, the best result would be the function combination that achieved the best performance in all three combinations.

Dataset

Using the entire dataset for this experiment was not yet possible, both because it did not exist in the correct form yet, and that the neural network was built to handle only one thousand networks, to begin with. For this experiment, a selection of around 38 thousand samples were processed in all function combinations.
The function combinations could consist of the following options:

- FFT window size: 1024, 2048, 4096
- n_chroma: 12, 24, 48
- DCT: 2, 3
- Power: 1, 2, 3
- Order: 0, 1, 2
- Center: True, False

The functions that were considered and the parameters that were tried with them are presented in Table 4.4. In total, 67 possible combinations were tried.

Function type	Parameter type
chroma_stft	FFT, n_chroma
chroma_cqt	n_chroma
chroma_cens	n_chroma
melspectrogram	FFT, Power
mfcc	DCT
rms	FFT
spectral_centroid	FFT, Center
spectral_bandwidth	FFT
spectral_contrast	FFT, Center
spectral_flatness	FFT, Power
spectral_rolloff	FFT
poly_features	FFT, Order
tonnetz	
zero_crossing_rate	

Table 4.4: Librosa functions that were tested

Control group

A control test was run to verify that the network achieved a useful result. The control test consisted of training the neural network model on a randomly generated group of classes. Should the experiment be successful, then the results of the tested networks will easily surpass the control group. In addition, if the control

group scored a high level of accuracy, different problems with the neural network itself could be highlighted.

4.2.2 Experiment results

The results that were used for further work in the thesis are listed in Table 4.5. While some of the results completely removed some functions entirely, others presented very good results. As expected, the control group achieved accuracy rates within the proximity of 10%, which given the ten output classes, means that the network failed to find anything in the control group.
In the experiment, however, the results were much better. Mel-frequency cepstrum coefficients with the Discrete Cosine Transform set to three provided the best results of all. While the Constant-Q chromagram has marginally better results in the loss and accuracy, its highest class count is significantly higher. The higher largest size of the 12 chroma test is caught up by the other two combinations, indicating the potential for improvement. Spectral contrast has also shown some promising results. However, the loss accuracy values are considered to be too perfect on the false parameter test, indicating potential pitfalls in using it as the primary dataset.

Function type	Param type	Loss/Acc	Class count and largest size
Control group	-	- / 9-12%	-
MFCC	DCT 2	0.6 / 0.86	10 - 203
MFCC	DCT 3	0.17 / 0.94	8 - 181
chroma_cqt	12	0.16 / 0.95	8 - 220
chroma_cqt	24	0.59 / 0.88	10 - 203
chroma_cqt	48	0.52 / 0.91	10 - 187
spectral_contrast	4096, False	0.01, 0.99	8 - 225
spectral_contrast	4096, True	0.64, 0.88	9 - 204

Table 4.5: Experiment 1 results

Ultimately, these three different function types have been determined to be the best options to use for the rest of the thesis. Of these, MFCC with DCT 3 was the only one used, as no issues with the dataset have arisen that would require a dataset change.

4.3 Extra layer experiment

As detailed in the technical walk-through of the extra layer modifications in Section 3.2.2, this experiment sought to put this layer to the test. The experiment was done using the neural network from the previous experiment. Thus, the dataset was also re-used, albeit with the learnings from that experiment.

4.3.1 Experiment setup

First, a custom dense layer was developed using the Tensorflow class interface. An initial group of 1000 weights was generated using standard functions. Then, a one-hot matrix was applied as the source for the final ten weights. The modified neural network model can be seen in Table 4.6. The first five layers have been frozen in this experiment to ensure that the model does not overfit heavily in this experiment.

Layer type	Parameters	Activation
Conv1d	64 filters, size 3	ReLU
Conv1d	64 filters, size 3	ReLU
Conv1d	64 filters, size 3	ReLU
Max pool	Size 2	ReLU
Flatten		
Dense	250	ReLU
Dense	10	Softmax
Dense (custom)	1010	Sigmoid

Table 4.6: Modified early neural network model

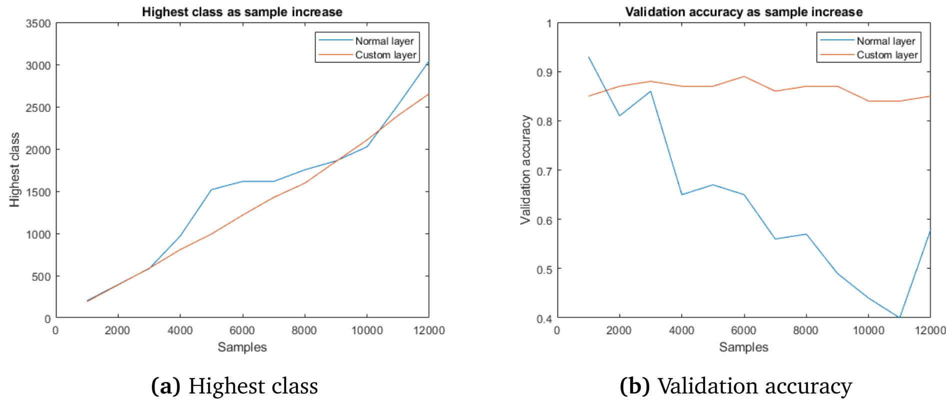
As an additional test to look for improvements in the experiment result, two new operations have been developed for the training process in this experiment.
The first operation sought to attempt to reduce the size of the largest class by running an iteration of the training process on the most abundant class. One thousand samples were picked at random to lose their initial classification and be reclassified. The "max" step has been programmed to happen every five iterations.
The second operation sought to increase the performance of the lowest-performing classifications in the dataset. The lowest-scoring 1000 samples would be reclassified again, with the aim to improve the certainty of the network. The "uncertain" step has been attempted every third iteration.

Control group

A standard dense layer with 1010 neurons has been used as the control group in this experiment. Should the custom layer completely fail at its task, or be a completely meaningless endeavor, using a standard dense layer would serve as a comparison.

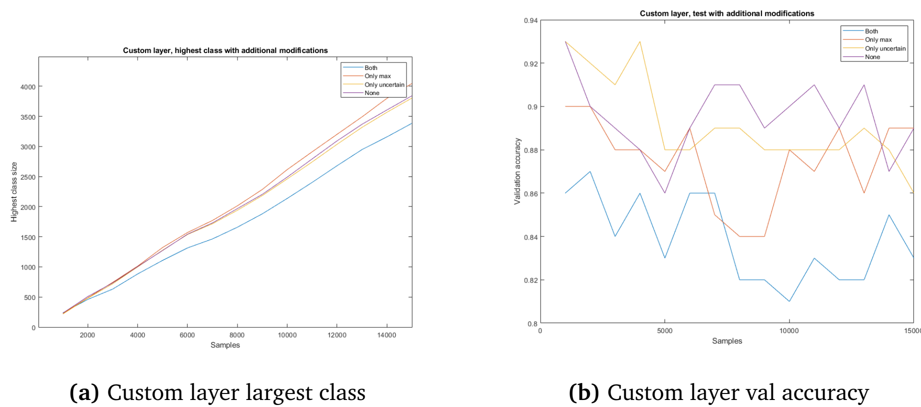
4.3.2 Experiment results

The experiment results have clearly shown that the custom layer is a superior option in the training of neural networks. Figure 4.1, Figure 4.2 and Figure 4.3 show that the custom layer maintains a steady growth with the dataset, while also maintaining accuracy. Meanwhile, the standard layer quickly loses in the largest



(a) Highest class **(b)** Validation accuracy

Figure 4.1: Largest class and validation accuracy as number of samples increase



(a) Custom layer largest class **(b)** Custom layer val accuracy

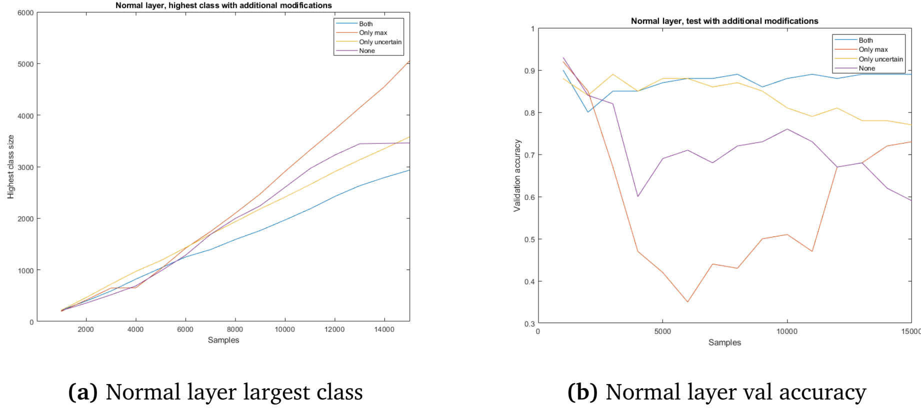
Figure 4.2: Custom layer performance with the extra operations

class metric and loses badly in the accuracy test. While some promise can be seen on Figure 4.3b, where using both additional operators has reached the same levels as the custom layer, the fact that the "max" option scored as low as it did indicates that this would likely not last for much longer.
Therefore, this experiment verified that the custom layer is an excellent addition to the neural network.

4.4 Neural network refinement experiments

During the refinement of the neural network model, the following workflow was adopted:

- Determine the correct number of convolutional layers
- Test correct parameters to use in the conv layers
- Test the correct number of pooling layers, and their type
- Test the correct number of neurons in the first dense layer



(a) Normal layer largest class

(b) Normal layer val accuracy

Figure 4.3: Normal layer performance with the extra operations

The last three steps were repeated in combination or by themselves depending on the number of parameters tested. Each of the tests generated thousands of results, of which most did not fall too far away from the best performing model. Therefore, the feeling of urgency and strictness with the neural network model was not present during development. Instead, this experiment aimed at being thorough in its trials, to make sure the experiment would not need to be repeated. After many different combinations of neural network structures have been attempted, the structure of the model was finalized. The final network structure can be seen in Table 4.7.

Layer type	Parameters	Activation
Conv1d	128 filters, size 2	ReLU
Max pool	Size 2	ReLU
Conv1d	64 filters, size 3	ReLU
Average pool	Size 2	ReLU
Conv1d	64 filters, size 4	ReLU
Max pool	Size 2	ReLU
Conv1d	16 filters, size 4	ReLU
Average pool	Size 2	ReLU
Flatten		
Dense	200	ReLU
Dense	10	Softmax
Dense (custom)	1010	Sigmoid

Table 4.7: Modified early neural network model

4.5 Loss function experiments

A similar process has been adapted to what has been used in Section 4.4 to investigate the correct parameters to use to determine the loss function filter boundaries, The parameters that were picked centered around the final output layer neurons divided by the number of classes that the samples could be allocated to. Thus, the clusters that would be below average would be rewarded, while the massive clusters would be penalized.

In addition to these parameters, two other components were tested in this experiment. The first component was the use of differently sized initial clusters generated by Scikit, to have a more generalized cluster in the beginning. The second component aimed to freeze most of the layers of the network used in the loss function processing, to prevent the result from overfitting. In the beginning, the model would only have its fresh final layer to train for a couple of epochs, until which point the softmax layer would also be opened for training, albeit at a reduced learning rate. The goal of this layer freezing was to reduce the instability of the loss function processing, by making sure that the significant shifts in the clusters would need to be justified, rather than caused by a configuration error.

4.5.1 Secondary filter

During the early experimentation, the secondary filter described in Section 3.3.2 has been tested with the rest of the parameters. While the difference between the combinations not using the filter was negligible, it was on the negative side of the results. As the secondary filter never added much to the results, only the removal of the first of the last neurons was kept.

4.5.2 Correct cluster count reevaluation

Following the first proper loss function experiment, it was revealed that all combinations of parameters have descended into three clusters. After an investigation, it was discovered that the other clusters generated by Scikit were unlikely to ever generalize with the rest of the dataset, and thus would only pollute the results. The loss function experiment has been rerun with only three clusters to correct for this error,

The code for the layers and the network was not adjusted to fit this discovery, as most of this code would remain dormant with no samples ever touching the extra nodes. As the network was optimized with ten clusters in mind, one more optimization pass was run on the network to be sure that the model would be good. Additionally, the parameters of the loss function experiment have been adjusted to fit the new three cluster output.

4.5.3 Experiment results

After the second loss function experiment has concluded, the following results were generated:

- Filter 0: First parameter 200
- Filter 1: First parameter 200, second parameter 800
- Filter 2: First parameter 400
- Filter 3: First parameter 400, second parameter 850
- Best iteration cluster size: 2000 (2x1000)
- First epoch limit: 5
- Second epoch limit: 10 (total of 15)

4.6 Iterative re-training experiment

4.6.1 Parameters used

The parameters used in the experiment are as follows:

- Function option, -2, -1, 0, 1, 2, 3
- Iteration threshold: 0.5, 0.6, 0.7, 0.8, 0.9

The iteration threshold defines the limit of how low certainty a sample needs to have to become a part of the dataset.
The function options 0 to 3 refer to the results of the experiment in Section 4.5, and the Figure 3.4 and Figure 3.5 filters used in the custom loss function. The option -1 is created to verify that the loss function filters are of any use. Option -1 is defined not to have a custom loss function; instead, it uses the standard sparse categorical cross-entropy. As the process of the re-classification also needs to be investigated, option -2 is created. Option -2 uses the classifications found during the search for the bad samples as the actual classes to use in the classification. Should the entire loss function process be compromised, while there is still some value to the iterative re-training itself, this option is expected to have the best results.

4.6.2 Experiment process

Most of the logical process behind the experiment can be found in Section 3.4.
Among the practical decisions made during the experiment, was the resetting of the final classification layer and layer freezing during the model re-train, after the new classifications were acquired. The process is similar to the one used in the loss function processing defined in Section 4.5; however, this process has three steps. As the final classification layer carries the most dataset-specific conclusions about the previous iteration, this layer was removed and replaced with a fresh layer. The final layer was then processed for a maximum of 20 epochs, with the remainder of the model frozen. Following this training process, the second last layer is unfrozen, and the model is set to resume training with a reduced learning

rate for another 20 epochs. Once the training freezes again, all but the first four of the network layers are unfrozen, and the training continues for another five epochs. An early stopping mechanism is used to prevent the model from needlessly churning the iterations, which stops the training process if the accuracy has not improved beyond 5% in the last ten epochs. Should this happen, the remaining epochs are transferred to the next training step to utilize.

4.6.3 Experiment results

The bulk of the results for this experiment can be found in Section 5.1. One variable parameter, however, had to be eliminated for the following tree experiment. Based on the initial results of this experiment, it was discovered that in almost all cases, iteration threshold 0.6 and 0.7 fail to achieve functional three clusters. Because of the detected failure in those clusters, the generation of those neural networks was frozen at iteration 25. As more and more observations were made on the results, it was determined that an iteration threshold of 0.8 would be used in the future experiment.

4.7 Tree generation experiment

4.7.1 Parameters used

The parameters used in this experiment are:

- Function option, -3, -2, -1, 0, 1, 2, 3
- Cache threshold: 0.5, (0.2, 0.3, and 0.4 with -3 only)

The function options are mostly the same parameters as used in Section 4.6.1. There is one new parameter called -3, which does not use any iterative re-training for the tree generation process.

The cache threshold option mimics the iterative threshold option, in that it determines if a sample is used for a purpose or not. With this parameter, a classification value higher than the threshold is necessary for the sample to be inserted into the next branch layer in the tree. The standard cache threshold value across most of the function options is 0.5. As the tree generation is expected to last a while, option -3 has been used to generate other tree combinations that there was not enough time for the other function options. Option -3 therefore uses cache thresholds 0.2, 0.3, and 0.4 in addition to 0.5.

Following the data from the previous experiment, an iteration threshold of 0.8 is used across the entire experiment.

4.7.2 Experiment process

At the start of the experiment, the networks generated in the previous experiment have been reused for this experiment. As the results would have been the same, it served to cut down the time to start significantly. A cache from the root node

has been generated for each of the three clusters that would receive their own networks.
The process can be summarized as follows:

1. Check if the node has at least 20'000 samples, delete node cache if not
2. Create the initial cluster using Scikit - AgglomerativeClustering
3. Train the first network iteration
4. Use iterative training (except -3) to create a new neural network
5. Create cache for the next branch layer
6. Repeat until no more cache folders to process

In addition to creating the cache for the next branch layer, should a node cluster so severely that it put all of its samples into the same cluster, that cluster is removed. These steps have utilized all available hard disk drives as caches for the tree generation, as each of the options had to have its copy of the dataset, in the case of the control options, even more than one copy.

4.7.3 Experiment results

All of the results for this experiment can be found in Section 5.2.

Chapter 5

Results

Results produced in this thesis stem from the final two experiments. First, from the iterative re-training of the neural network, then from using the iterative re-training to create a classification tree. As the experiments did varying amounts of repeating steps that can magnify potential errors in the final results, the results are presented in a broad view.

5.1 Iterative re-training results

This section goes through the entire result dataset for Section 4.6, in various forms. The parameters used in the various examples are:

- Function option, -2, -1, 0, 1, 2, 3
- Iteration threshold: 0.5, 0.6, 0.7, 0.8, 0.9

An explanation of these values is listed in Section 4.6.1. The results are based on parsing the entire dataset with the neural network created on each iteration of the re-training process. As the results can be broken up into three general types, the resulting graphs are broken up into different groups where appropriate.

5.1.1 Size of clusters over the iterations

Figure 5.1, Figure 5.2, and Figure 5.3 detail the size of the result clusters as the number of iterations increases.
As the size of the clusters is the most straightforward value to track across the iterations, it is listed first. Partial cluster size results were used to make the decision on which iteration threshold to pick for Section 4.7.

5.1.2 Value of the highest classifications over the iterations

The value of the highest classification is also a vital characteristic to track across the iterations. Should the cluster size vary wildly with the highest classification value remaining the same, it could indicate a heavy overfit in the network and

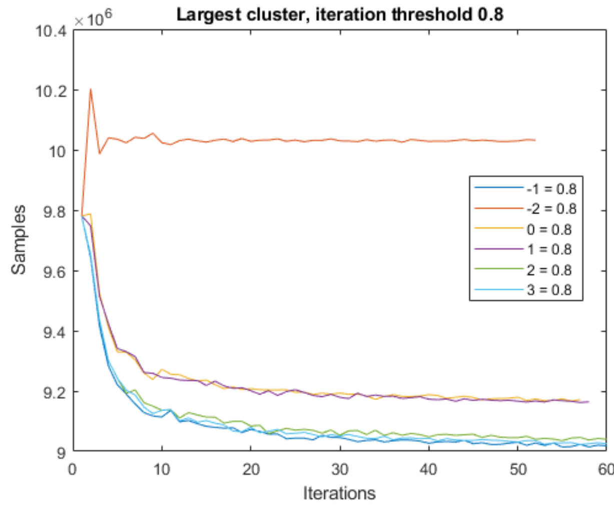
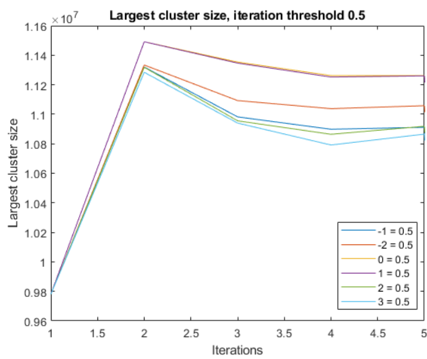
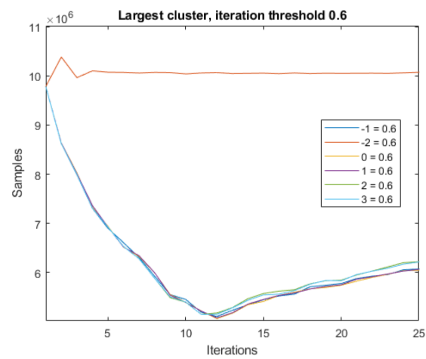


Figure 5.1: Largest cluster over the iterations, with 0.8 iteration threshold

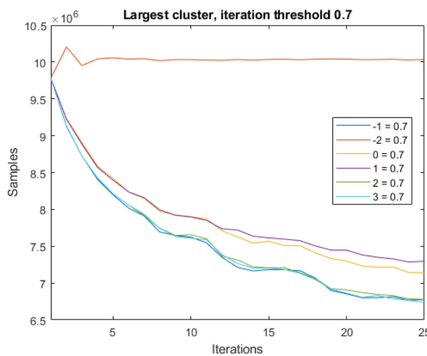


(a) 0.5 iteration threshold

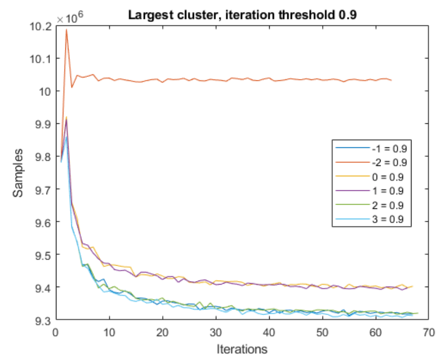


(b) 0.6 iteration threshold

Figure 5.2: Largest cluster over the iterations, with 0.5 and 0.6 iteration threshold



(a) 0.7 iteration threshold



(b) 0.9 iteration threshold

Figure 5.3: Largest cluster over the iterations, with 0.7 and 0.9 iteration threshold

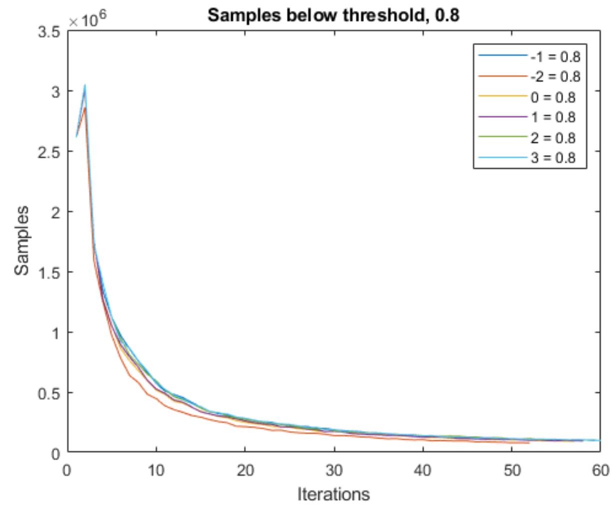
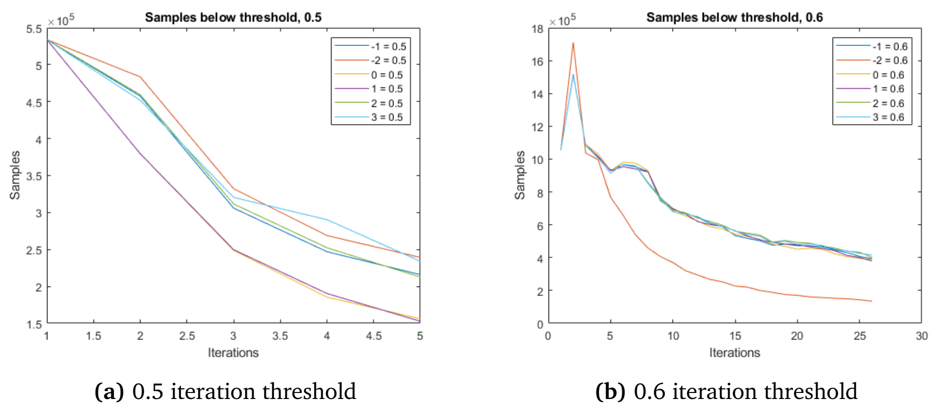


Figure 5.4: Samples below iteration threshold, with parameter set to 0.8.



(a) 0.5 iteration threshold

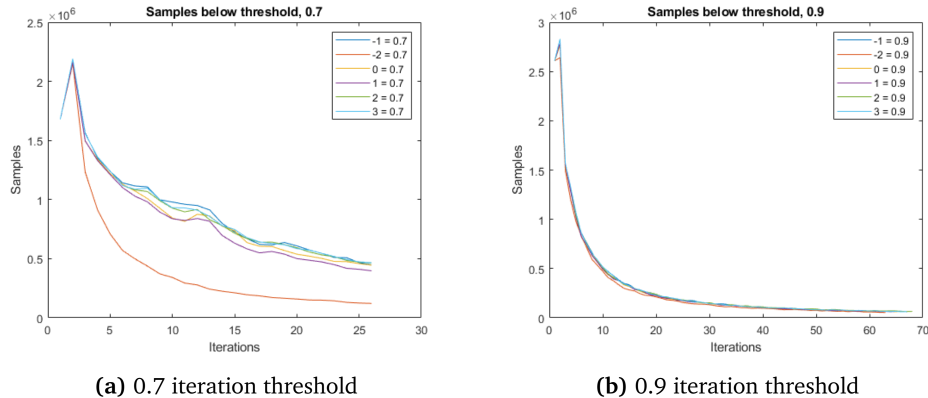
(b) 0.6 iteration threshold

Figure 5.5: Samples below iteration threshold, with parameter set to 0.5 and 0.6

flimsy reasoning for the cluster separation. Besides, the goal of the iterative re-training is to reduce the uncertainty of the classification, which can be tracked by following the highest classification values for the given samples.
Following the iteration threshold parameter, the most critical change in the highest classification happens when the iteration threshold is reached. Figure 5.4, Figure 5.5, and Figure 5.6 present the number of samples that are below the iteration threshold over the course of the iterative re-training.

5.1.3 Variation between first iteration and the following iterations

The goal of the re-training process is to train the network more, faster than just using the entire dataset and a big data center worth of hardware. However, significant variations throughout the entire training process can indicate that the



(a) 0.7 iteration threshold

(b) 0.9 iteration threshold

Figure 5.6: Samples below iteration threshold, with parameter set to 0.7 and 0.9.

parameters do not converge into a particular cluster group. While the first iteration is expected not to be the perfect network immediately, it is still expected to be a general approximation of the problem, and significant changes to the initial cluster group can indicate problems in the combinations.
Only changes of 0.1 or more are tracked to represent the change as something more significant than a minor rounding error. Figure 5.7, Figure 5.8, and Figure 5.3 present the number of samples that had a change in the highest class.

5.1.4 Variation between neighbor iterations

In a similar manner to the previous section, this section aims to compare the neighboring iterations with each other. While significant variations could remain present when comparing the first iteration with the others after new information was gained, comparing neighbors serves to localize the comparison. If the variation between the iterations remains significant, it could indicate that the networks never converge into a particular classification, and just keep juggling samples between the classes.
The result in this section mimics the previous section, only with a different set of results. Only changes of 0.1 or more are tracked to represent the change as something more significant than a minor rounding error. Figure 5.10, Figure 5.11, and Figure 5.12 present the number of samples that had a change in the highest class.

5.1.5 Number of sample files between training iterations

Finally, an important variable to analyze is the effect of the iterative re-training over time, as the dataset is processed. While in the beginning little is known about the dataset, as more and more information is gleaned on the dataset, it is expected that the network becomes more and more capable of handling information it has not seen yet. Should the number of files between iterations not change, or even go

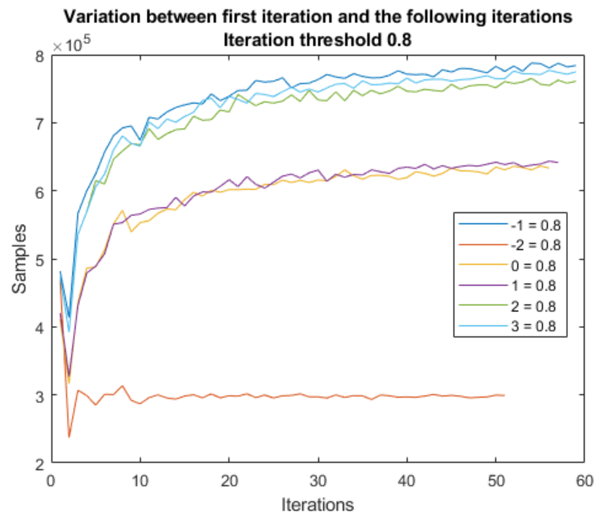
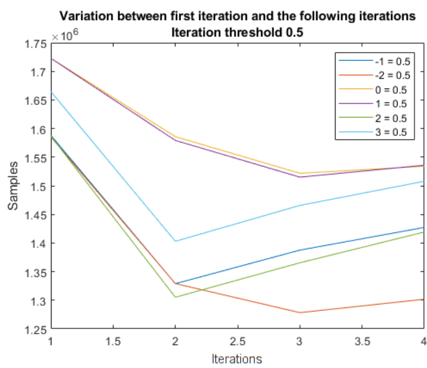
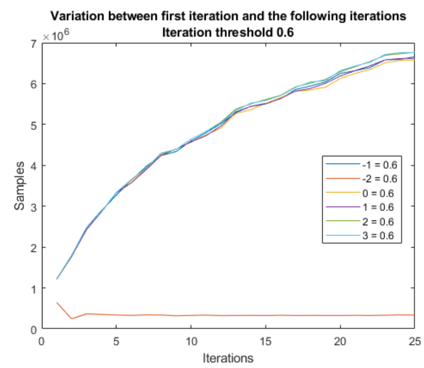


Figure 5.7: Variation between first and next iterations, with 0.8 iteration threshold

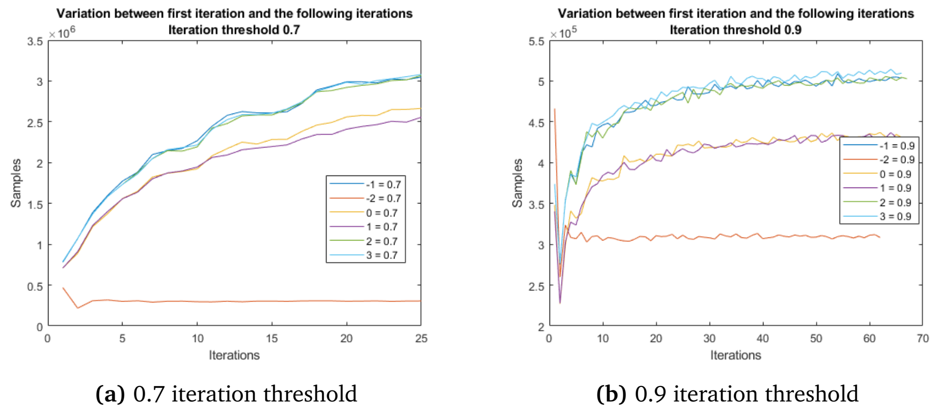


(a) 0.5 iteration threshold



(b) 0.6 iteration threshold

Figure 5.8: Variation between first and next iterations, with 0.5 and 0.6 iteration threshold



(a) 0.7 iteration threshold

(b) 0.9 iteration threshold

Figure 5.9: Variation between first and next iterations, with 0.7 and 0.9 iteration threshold

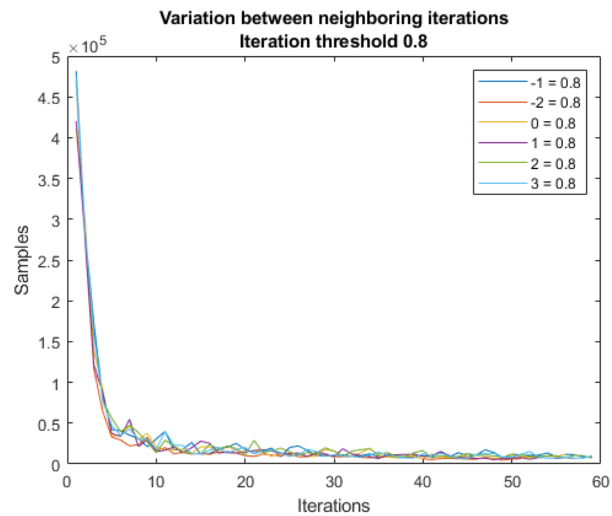
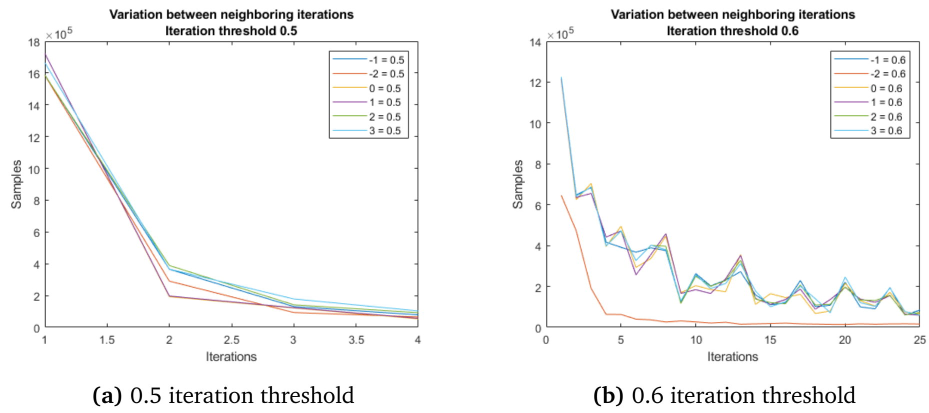


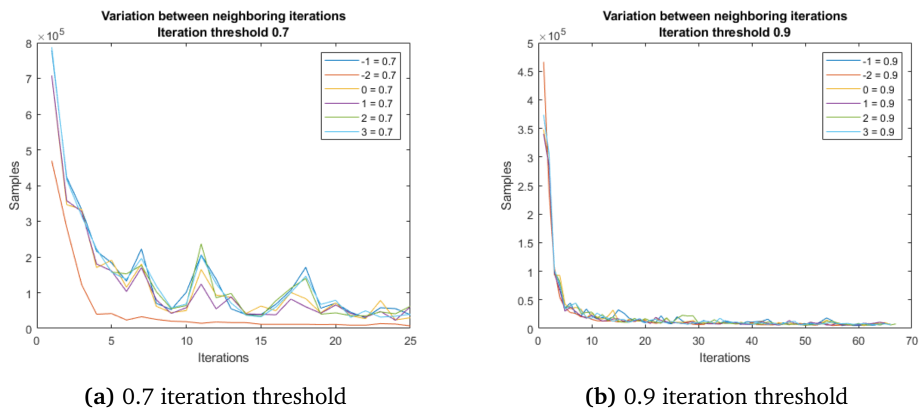
Figure 5.10: Variation between neighbor iterations, with 0.8 iteration threshold



(a) 0.5 iteration threshold

(b) 0.6 iteration threshold

Figure 5.11: Variation between neighbor iterations, with 0.5 and 0.6 iteration threshold



(a) 0.7 iteration threshold

(b) 0.9 iteration threshold

Figure 5.12: Variation between neighbor iterations, with 0.7 and 0.9 iteration threshold

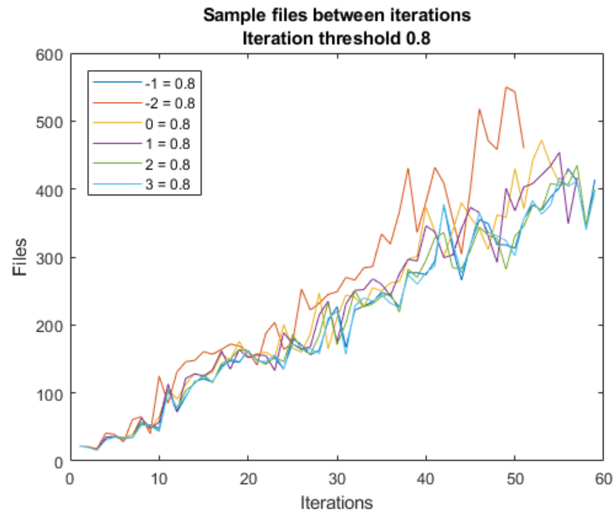


Figure 5.13: Change in number of sample files between iterations, with 0.8 iteration threshold

down, it could indicate that the network loses information in the training process, or that the samples have a meager amount of common features between each other. Figure 5.13, Figure 5.14, and Figure 5.15 show the change in number of files used by the various function options over the course of the iterative re-training process.

5.2 Tree results

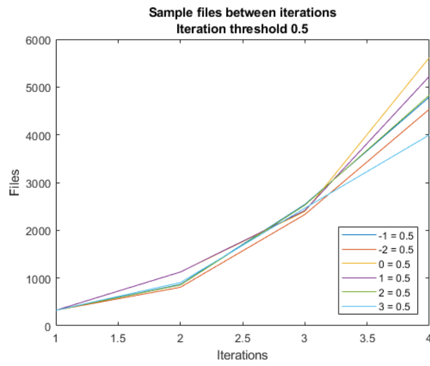
This section goes through the entire result dataset for Section 4.7. The parameters used in the various examples are:

- Function option, -3, -2, -1, 0, 1, 2, 3
- Cache threshold: 0.5, (0.2, 0.3, and 0.4 with -3 only)

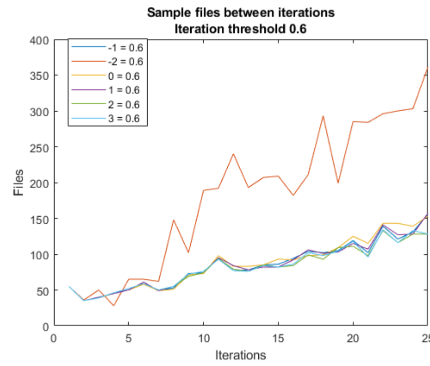
An explanation of these values is listed in Section 4.7.1. As the results are far more numerous in this section, processing of these results using all of the neural networks would take an excessive amount of time. Therefore, a compromise was used in the form of utilizing the classification results generated for the caching process.
In addition to the standard comparison of function options, Section 4.7 included a control group that did not use iterative re-training. Results of that particular group of tests are included in the last subsection.

5.2.1 Cache size of the nodes in the tree

Among the most relevant results for the tree generation process are the sizes of the caches used to train the various nodes in the tree. Some function options could

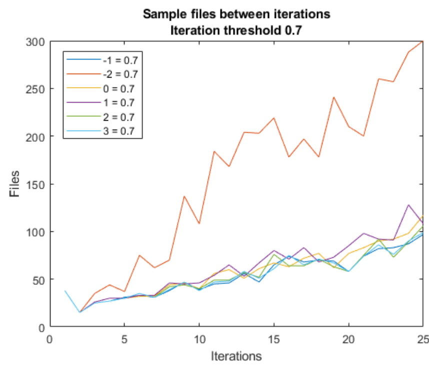


(a) 0.5 iteration threshold

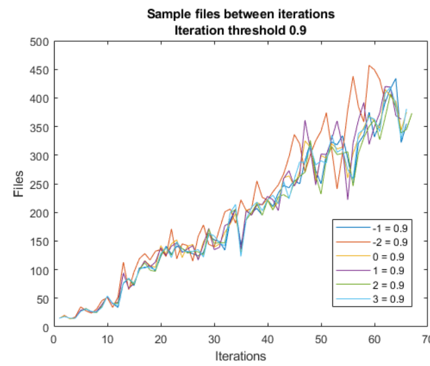


(b) 0.6 iteration threshold

Figure 5.14: Change in number of sample files between iterations, with 0.5 and 0.6 iteration threshold

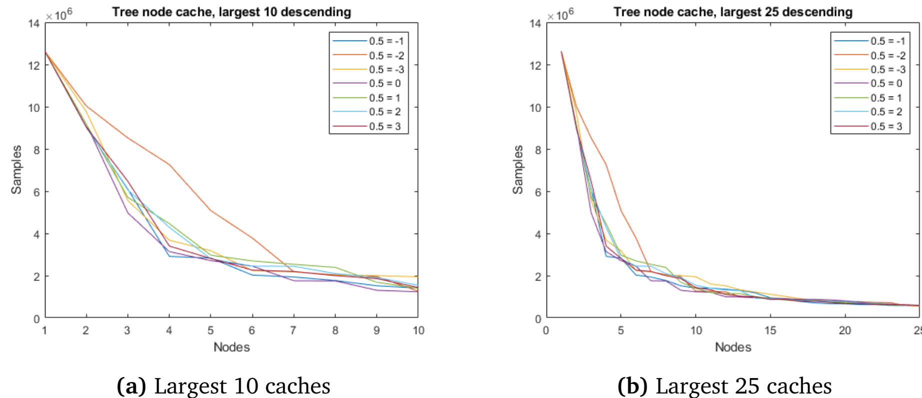


(a) 0.7 iteration threshold



(b) 0.9 iteration threshold

Figure 5.15: Change in number of sample files between iterations, with 0.7 and 0.9 iteration threshold



(a) Largest 10 caches

(b) Largest 25 caches

Figure 5.16: Largest tree node caches in descending order. All start from full dataset.

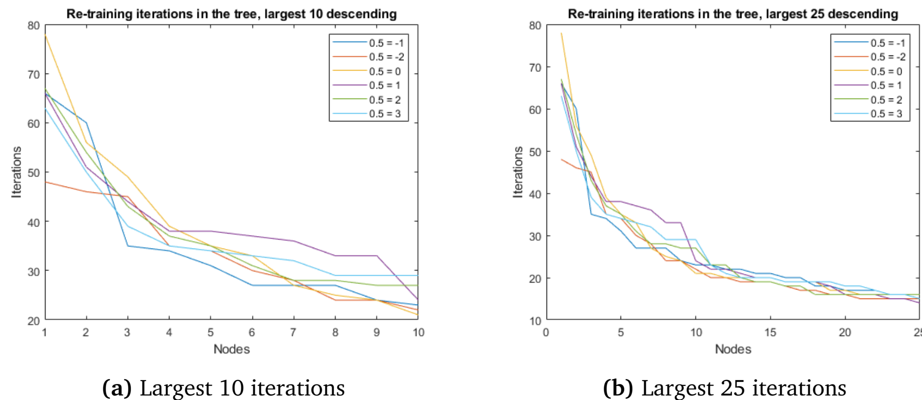
distribute their dataset more evenly, while others could heavily lean towards putting all samples in just one class. Furthermore, the control groups could provide the same or better results than the developed experiment, casting doubt into the need for the extra processing steps.
Figure 5.16 shows the size of the cache, descending from the largest to smallest. As each tree can have different trees putting samples in differently named clusters, the names of the clusters are ignored.

5.2.2 Re-training iterations throughout the trees

Using the iterative re-training process in the tree generation is a very dynamic way to test the method on a vast number of sample combinations. However, it can also show signs of failure if it takes more and more iterations to train each next node in the tree. Iterative re-training adds a constant number of samples per iteration to the training process, which can, in the worst case, end up using the entire dataset.
Figure 5.17 show the number of iterations used in the tree nodes, descending from largest to smallest.

5.2.3 Failed node generation per branch layer

Throughout the tree generation process, various samples are removed from further use in the tree. First, a small number of samples are removed due to a hard requirement of one thousand samples per sample file. Second, every time a tree node receives less than 20 thousand samples to work with, it is considered finished and subsequently removed. How this process has developed for each tree generation is a valuable insight into how the tree has formed, and its structure as a whole. Each tree node can only lose so many samples, meaning that if the tree does not distribute samples enough, it will be very narrow.



(a) Largest 10 iterations

(b) Largest 25 iterations

Figure 5.17: Largest tree node iterations in descending order. Excluding full data-set.

Figure 5.18 shows how many nodes failed to be created on a per branch level. Figure 5.19 shows how many samples were lost during the tree generation process on a per branch level.

5.2.4 Comparison between samples classified by the tree

To compare the performance of the tree generation on the real data, three samples are presented in figures Figure 5.20, Figure 5.21, and Figure 5.22.
The first sample is the most distinct of the three, and splits off from the other two in the tree at the root node, being classified as class 0 while the other two samples are classed as class 2. The two similar samples follow each other until the end of the tree, with the resulting class combination "210012".

5.2.5 Control insight

Section 4.7 used a control function option called "-3" that avoided the re-training process entirely to control for potential faults in the iterative re-training process The iterative re-training of the other options meant that while training the other options could take hours due to slowly processing the entire dataset; this control option could proceed immediately to the caching stage after the first iteration was trained. The faster processing time enabled the use of lower cache threshold options, which carry different values than the rest of the trees. Due to the nature of the softmax layer, with a caching threshold of 0.5, only one node could receive a sample to process. Using a lower cache threshold enabled the possibility of multiple nodes receiving the same sample, inflating the original dataset throughout the tree generation.
The results of the control group can be compared with the other trees through the combination using the 0.5 cache threshold. Figure 5.23 displays the cache size of the nodes in the control trees.

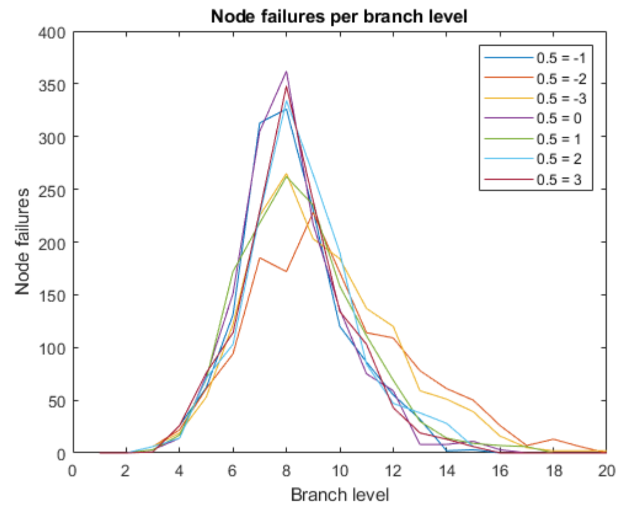


Figure 5.18: Nodes that failed to be created due to insufficient samples in the cache

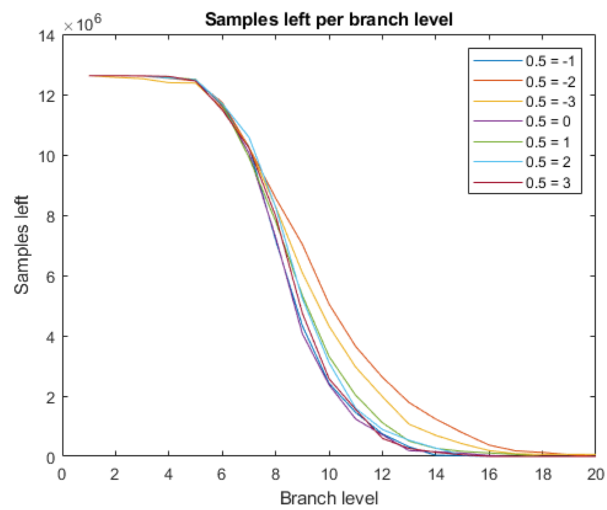


Figure 5.19: Samples left per branch level over the course of the tree generation

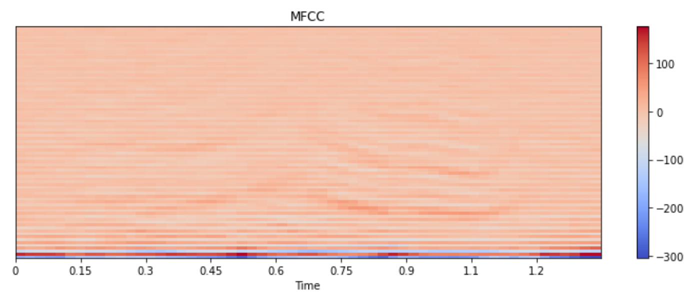


Figure 5.20: Audio sample 1, English translation: "Kabu, I'm home!"

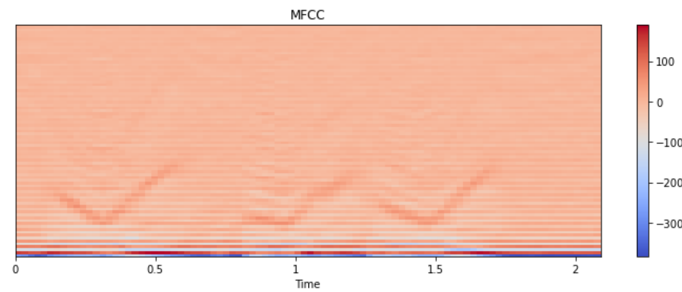


Figure 5.21: Audio sample 2, English translation: "Nu-uh! I'll go by myself..."

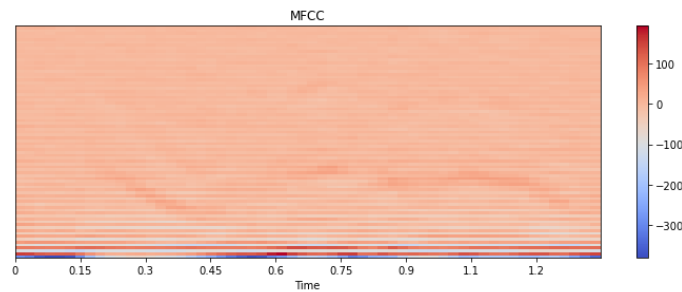
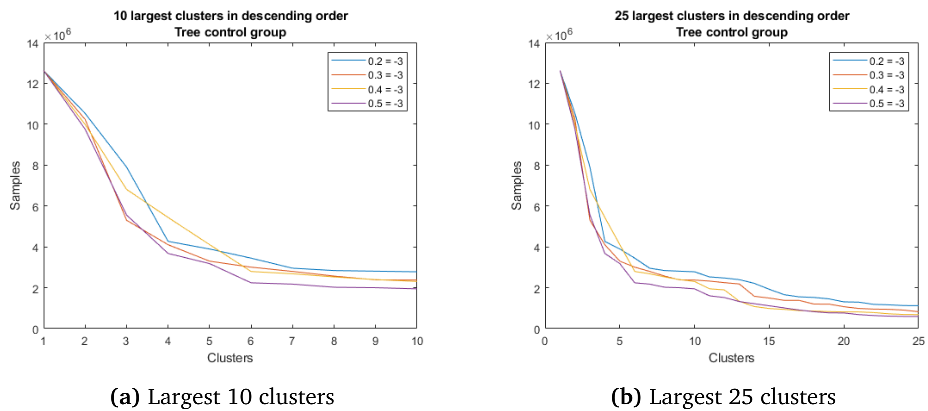
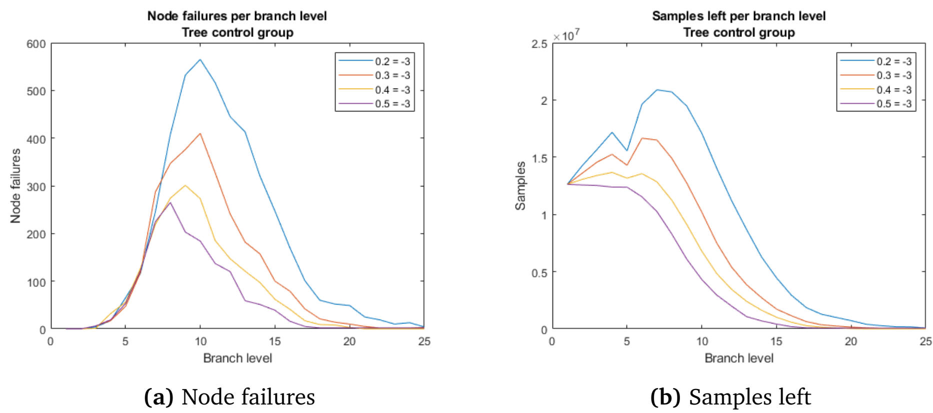


Figure 5.22: Audio sample 3, English translation: "All right, I'm finished"



(a) Largest 10 clusters **(b)** Largest 25 clusters

Figure 5.23: Largest tree node clusters in the control group, in descending order. Including full dataset.



(a) Node failures **(b)** Samples left

Figure 5.24: Node failures and samples left per branch level in the control group.

Figure 5.24 shows how many nodes failed to be created, and the loss of samples on a per branch level.

Chapter 6

Discussion

The main research goal was to investigate these three research questions that were presented in the introduction:

- How do neural networks that are iteratively re-trained using transfer learning on an increasing subset of the dataset, to group the entire dataset they receive into "super-classes" perform?
- How does this training technique perform when used to train neural networks in a tree hierarchy that bases itself on these super-classes?
- How does changing the parameters of the training process affect the neural networks and the resulting tree structures?

Each of the following sections aims to answer how the research conducted in this master thesis provided insight into the potential answer to these questions.

6.1 Iterative re-training performance

From the results of the iterative training performance, it is very apparent that the process of using the loss function training step provides better results than merely skipping that step. In Figure 5.1, it is clear that while all other combinations follow the slope down to lower the size of the largest cluster, option -2 that does not use this process fails and even proceeds to create an even bigger cluster. Looking more directly on the rest of the options, the control option -1 is showing similar levels of cluster distribution as the gradual loss filters 2 and 3. Meanwhile, the drastic filter options 0 and 1 are above in the largest cluster size, both in the 0.8 iteration threshold, and others like Figure 5.3. From these graphs alone, it is easy to conclude that while the loss function filters provide little benefit, the process itself is superior to just going with the most significant classification from the previous model.
However, a somewhat different picture can be seen on Figure 5.7. At first glance, the -2 option turns out to be the one that varies from the first iteration the least, while the rest vary more and more. The variation, however, is a desired trait of the loss function process, and the gradual stabilization of the superior three (-1,

2, 3) options indicates a steady stabilization of the process.
The stabilization becomes the most apparent on Figure 5.10. In all options, the variation between the neighboring iterations rapidly descends to a minimum that can be attributed to training process artifacts. By the time the 10th iteration is reached, little more can be learned by the network. With each iteration adding another 2000 samples to the dataset, a significant portion of the variation can be caused by new samples being used in the training process.
To ultimately conclude that going too far beyond the 10th iteration is bound to bring diminishing returns, Figure 5.4 can be referenced. At around 10th iteration, over 80% of samples that were initially below the threshold have been put above it. At that point, only 20'000 samples have entered the dataset. Since all samples appear to be descending at the same speed, there is no difference between them in this comparison. However, it is interesting to note that the -2 option that has been relatively bad in the primary 0.8 results, appears to be maintaining its results stably across all cluster size figures. In contrast, other options struggled at 0.6 and 0.7.
Lastly, in terms of the rate of sample files between iterations, the results are relatively similar in all cases. As previously, option -2 sticks out and does its own thing, but in general, the number of sample files between iterations goes up on a somewhat linear basis across the dataset.

6.2 Tree generation process

During the tree generation process, the distinctness of the -2 option has stayed from the iterative re-training experiment. In Figure 5.16, it is the most distinct of the options by having excessively large first six or so rows. On the flip side, on Figure 5.17 it has the least amount of iterations in the first two nodes.
The other options, however, very quickly descend to two and then one million samples per node. However, the first node in all remaining options has managed to pass its root node in iterations to finish. The tree generation had to be conducted strictly on a single thread basis for a while, as even one thread had consumed all available RAM storage for its dataset.
Over time, the number of iterations drops down as there are fewer samples to go around in the nodes. By the time Figure 5.17b finishes at 25 nodes, the iterations drop down to below 20, which is close to where the sweet spot for iterative retraining appears to be from the previous section's findings.
Continuing in investigating the performance of the trees, we can see from Figure 5.18 that most of the nodes are removed at around level 8, with some stragglers remaining until the end of the graph. As the first cluster generation is tied strictly to how good the first 2000 samples are in clustering, this result may not represent each of the function options accurately. However, given that this trend repeats itself for all options, it is relatively safe to say that for a dataset of 12.6 million samples, most of the branching will be finished by branch level 10, assuming a minimum branch size of 20'000 samples.

Lastly, for the comparison of the main options, all options had lost roughly the same amount of samples on each branch level, descending rapidly around level 8, where most nodes were removed. Option -2 is again losing the least amount of samples per level, followed shortly by the control option -3.
Moving on to the control options, the size of the most massive clusters in Figure 5.23 is not indicative of much. The lower cache threshold options are slightly larger than the option using the shared variable, but in general, the largest cluster size maintains itself steadily.
The difference becomes apparent on Figure 5.24. In the case of the control option using the cache threshold of 0.2, the cache size almost doubled before going down eventually. As the size of the dataset increases, so did the number of node failures per branch level, with the 0.2 control option losing almost more branches in one level than some of the trees had in total.
Lastly, the tree is proven to achieve some degree of similarity detection in Section 5.2.4. While it can be argued that the two samples that were classified as similar should have been separated at some point earlier, it is clear that both are distinctly different from the first example.

6.3 Effect of parameter change

As noted in the previous sections, including the loss function process carries a clear benefit to the results, even if only the normal sparse categorical cross-entropy loss function is used. The samples receive more specific classification during this step, which may correctly reclassify these samples as something else.
Between the iterative re-training options that did use the loss function process, a clear split emerged between the options that went with a more heavy-handed approach to the filter process, and the ones that used more gradual filters. The decrease in quality from the drastic filters may be a sign that the filters did not do much for the training process if only to slow it down with custom code. As the more drastic effects of the gradual filters would only become pronounced at the ends of the filters, their filters are likely to produce similar results to those of the standard loss function.
Perhaps most interestingly, the effect of the iteration threshold parameter has had a tremendous effect on the training process. Threshold of 0.6 Figure 5.2b and 0.7 Figure 5.3a displayed significant reduction in quality compared to the other options. Meanwhile 0.5 Figure 5.2a increased its largest cluster size to encompass almost the entire dataset. While more research on these parameters should be done for other datasets and purposes, in this thesis, using an iteration threshold of less than 0.8 has not been productive.
Unlike the root node generation in iterative re-training, comparing the parameter changes on the tree generation is not as easy. Due to the nature of the autonomous cluster generation, each tree is bound to become different from the rest very quickly. However, some interesting facts can be gleaned from the results. Even though the control option without iterative training has the least amount of know-

ledge about the dataset, it manages to achieve the same largest cache sizes as the other options. On the other hand, the option -2 that uses the previous network's results are becoming not only inferior but likely damaged by its training process. In the few parameters that were investigated for tree generation, other than the erroneous option -2, none of the results stick out from the rest.
Changing the cache threshold as has been done in the control group of tree generation, however, does have a substantial effect on the time spent processing the tree. The effect of reducing the cache threshold does not appear to be of much significance beyond spending more time on processing, however.

Chapter 7

Conclusion

In conclusion, the thesis has investigated a way to use currently existing hardware more optimally by clustering data through the use of smaller neural networks in tree hierarchies. The tree clustering has been facilitated by a vast home-made dataset containing over 32 TB of raw video containing Japanese audio and English text. This could make the thesis relevant for future students and researchers that wish to start developing neural networks but lack the dataset to start. Even after preprocessing all of the data, more than 370 GB and over 12 million samples were ready to be used without any large-scale labeling actions taken by commercial or private actors.

Iterative re-training has proven to provide meaningful results, while the loss function filters have not been all that effective. Their development has still led to the introduction of the loss function process in the thesis, which turned out to be the key to make the iterative re-training work for the thesis dataset. Usage of less computationally expensive alternatives has simultaneously been proven to be not very productive.

Due to the nature of the tree generation, it has been impossible to study in-depth the differences between the trees. Nonetheless, the metadata provided during their creation has shown that the method is relatively stable, if not without some minor bad clusters. Comparing the samples classified by a tree, it has been shown that more similar samples are clustered together, while more distinct samples are kept separate.

The effects of changing the various parameters during the experiments have also been investigated. Specific parameters like the iteration threshold have been proven to be very sensitive to change, with lower values producing vastly inferior results. Other parameters, like the cache threshold, have been shown to produce very similar results across the different tested values while wasting more time for the developer.

7.1 Future work

While this thesis aimed to succeed in accomplishing many goals set up at the beginning of the thesis, plenty of work remain. Initially, the thesis aimed to produce a full-fledged Japanese speech to English text translator. Eventually, only the core hypothesis of iterative re-training and tree generation has been accomplished in time. It is likely to be possible to accomplish on the hardware available to consumers using the work done in this thesis as a basis.
Of the future work that could be done in the immediate future, the replacement of the activation function in the custom dense layer is an interesting aspect to research. Currently, only Sigmoid has been used as the last activation function. As Sigmoid due to its nature, compresses the input it receives from a Softmax activation function, and other functions could perform better. A potential candidate for improvement is the ReLU activation function.
Another area that could be tested more is more variants of the loss function filters. The filters used in the thesis are simple filters that are easy to program, but may not be the most optimal mathematical functions that could distribute the loss values better. It may be possible that in combination with a different activation function, loss function filters could prove to be slightly superior to the standard sparse categorical cross-entropy loss function.
Lastly, more architecture and analysis of the trees themselves is necessary to reach any decisive conclusion about them. Due to time limitations caused in part from the trees taking an excessive amount of time to generate, only the cache used in the creation of the trees could be used for analysis. Other useful information could come forward if more analysis is conducted.

Bibliography

[1] D. Hebb, *The Organization of Behavior: A Neuropsychological Theory*. Taylor & Francis, 2005, ISBN: 9781135631901. [Online]. Available: https://books.google.no/books?id=ddB4AgAAQBAJ.

[2] P. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Harvard University, 1975. [Online]. Available: https://books.google.no/books?id=z81XmgEACAAJ.

[3] C. Mead and M. Ismail, *Analog VLSI Implementation of Neural Systems*, ser. The Springer International Series in Engineering and Computer Science. Springer US, 2012, ISBN: 9781461316398. [Online]. Available: https://books.google.no/books?id=oNjTBwAAQBAJ.

[4] J. Weng, N. Ahuja and T. S. Huang, 'Cresceptron: A self-organizing neural network which grows adaptively', vol. 1, 576–581 vol.1, 1992.

[5] A. Krizhevsky, I. Sutskever and G. E. Hinton, 'Imagenet classification with deep convolutional neural networks', *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017, ISSN: 0001-0782. DOI: 10.1145/3065386. [Online]. Available: http://doi.acm.org/10.1145/3065386.

[6] ImageNet, *Imagenet large scale visual recognition challenge (ilsvrc)*, 2019. [Online]. Available: http://www.image-net.org/challenges/LSVRC/.

[7] ImageNet, *Imagenet large scale visual recognition challenge 2012 (ilsvrc2012)*, 2012. [Online]. Available: http://image-net.org/challenges/LSVRC/2012/results.html.

[8] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, *Going deeper with convolutions*, 2014. arXiv: 1409.4842 [cs.CV].

[9] ImageNet, *Imagenet large scale visual recognition challenge 2014 (ilsvrc2014)*, 2014. [Online]. Available: http://image-net.org/challenges/LSVRC/2014/results.

[10] Wikipedia contributors, *Sigmoid function — Wikipedia, the free encyclopedia*, [Online; accessed 27-May-2020], 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Sigmoid_function&oldid=955047622.

[11] Wikipedia contributors, *Cross entropy — Wikipedia, the free encyclopedia*, [Online; accessed 20-May-2020], 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Cross_entropy&oldid=945192009.

[12] Wikipedia contributors, *Stochastic gradient descent — Wikipedia, the free encyclopedia*, [Online; accessed 20-May-2020], 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Stochastic_gradient_descent&oldid=956880506.

[13] Wikipedia contributors, *Transfer learning — Wikipedia, the free encyclopedia*, [Online; accessed 22-May-2020], 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Transfer_learning&oldid=958093515.

[14] Google - Tensorflow tutorials, *Transfer learning with a pretrained convnet*, [Online; accessed 22-May-2020], 2020. [Online]. Available: https://www.tensorflow.org/tutorials/images/transfer_learning.

[15] Matlab, *Pretrained deep neural networks*, 2019. [Online]. Available: https://se.mathworks.com/help/deeplearning/ug/pretrained-convolutional-neural-networks.html.

[16] S. S. Stevens, 'A Scale for the Measurement of the Psychological Magnitude Pitch', *Acoustical Society of America Journal*, vol. 8, no. 3, p. 185, Jan. 1937. DOI: 10.1121/1.1915893.

[17] N. Ahmed, T. Natarajan and K. R. Rao, 'Discrete cosine transform', *IEEE Transactions on Computers*, vol. C-23, no. 1, pp. 90–93, 1974.

[18] P. Dhanalakshmi, S. Palanivel and V. Ramalingam, 'Pattern classification models for classifying and indexing audio signals', *Engineering Applications of Artificial Intelligence*, vol. 24, no. 2, pp. 350–357, 2011, ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2010.10.011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0952197610001971.

[19] H. Gimse, *Classification of marine vessels using sonar data and a neural network*, 2017. [Online]. Available: http://hdl.handle.net/11250/2453247.

[20] Wikipedia contributors, *Hierarchical clustering — Wikipedia, the free encyclopedia*, [Online; accessed 24-May-2020], 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Hierarchical_clustering&oldid=955878698.

[21] L. Lerato and T. Niesler, 'Feature trajectory dynamic time warping for clustering of speech segments', English, *EURASIP Journal on Audio, Speech, and Music Processing*, vol. 2019, no. 1, pp. 1–9, Apr. 2019, Copyright - EURASIP Journal on Audio, Speech, and Music Processing is a copyright of Springer, (2019). All Rights Reserved.; © 2019. This work is published

under http://creativecommons.org/licenses/by/4.0/ (the "License"). Notwithstanding the ProQuest Terms and Conditions, you may use this content in accordance with the terms of the License; Last updated - 2019-04-05. [Online]. Available: https://search.proquest.com/docview/2203088894?accountid=12870.

[22] D. Roy, P. Panda and K. Roy, *Tree-cnn: A hierarchical deep convolutional neural network for incremental learning*, 2018. arXiv: 1802.05800 [cs.CV].

[23] Z. Yan, H. Zhang, R. Piramuthu, V. Jagadeesh, D. DeCoste, W. Di and Y. Yu, 'Hd-cnn: Hierarchical deep convolutional neural networks for large scale visual recognition', pp. 2740–2748, 2015.

[24] Kamedo2, *Results of the public multiformat listening test (july 2014)*, [Online; accessed 29-May-2020], 2014. [Online]. Available: https://listening-test.coresv.net/results.htm.

[25] Wikipedia contributors, *Substation alpha — Wikipedia, the free encyclopedia*, [Online; accessed 29-May-2020], 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=SubStation_Alpha&oldid=934658921.

[26] TechPowerUp, *Gigabyte rtx 2080 ti gaming oc*, 2019. [Online]. Available: https://www.techpowerup.com/gpu-specs/gigabyte-rtx-2080-ti-gaming-oc.b6097.

[27] Theano - Theano tutorial, *Python memory management*, [Online; accessed 26-May-2020], 2020. [Online]. Available: http://deeplearning.net/software/theano/tutorial/python-memory-management.html.