RADBOUD UNIVERSITY NIJMEGEN

FACULTY OF SOCIAL SCIENCES

Detecting mispronunciation of digit nine in ATC communciation using keyword spotting

BACHELOR THESIS ARTIFICIAL INTELLIGENCE

Supervisor:
dr. Franc GROOTJEN

Author:
Honghong ZHU

Second reader:
dr. Umut GÜÇLÜ

July 2020

Abstract

Air Traffic Control (ATC) communication is an important process to ensure aviation safety. A minor mistake can lead to a disastrous event. Therefore, communication mistakes by pilots and controllers should be prevented. Previous studies have investigated the communication mistakes that are being made and its factors. However, information about automatically detecting communication mistakes are still inadequate, while it can provide better insights into the mistakes being made and help prevent them. One of the common mistakes made is the mispronunciation of the digit nine. Therefore, this thesis aims to detect the mispronunciation of the digit nine by pilots and controllers. A keyword spotting system based on convolutional recurrent neural networks by Kim and Nam (2019) is used to detect mispronunciations of the digit nine in ATC audio fragments. Furthermore, three different class imbalance techniques are explored to improve the model performance: random oversampling, weighted random sampling and weighted cross-entropy loss. The results of the techniques are analyzed both individually and comparatively to determine which technique is best suited for the model and dataset. The results of this thesis indicate that the model with weighted cross entropy-loss can detect the pronunciations significantly above chance level. However, further improvement on the model is still necessary to achieve at least the same results as Kim and Nam (2019) and provide aid in reducing the ATC communication mistakes.

Contents

1 Introduction

The growing demand for global air traffic in the past decades has led to more challenges for Air Traffic Control (ATC) operations. The size of global air traffic has doubled every 15 years since 1977 and the growth is expected to continue (ICAO, 2014). Therefore, it is important that ATC operations are investigated and improved to ensure they can meet the growing demand. One of the important components in ATC operations is ATC communication. To ensure aviation safety, good communication between pilots and air traffic controllers is crucial. ATC communication follows a standard phraseology to prevent confusion between pilots and controllers. Nevertheless, miscommunication still occurs frequently and can lead to disastrous accidents. For example, in 1977 the KLM Boeing 747 crashed with 583 deaths due to a small miscommunication. One of the main reasons for miscommunication is that pilots and controllers fail to use the standard phraseology (Said, 2011). Non-standard phraseology is everything that does not correspond to the prescribed standard words and phrases. Therefore, the possibilities of non-standard phrases are large. Non-standard phraseology includes using random words, words in a wrong order, wrong pronunciation for letters and numbers, and a lot more. Some more detailed information about ATC communication and the standard phraseology is described in Section 1.1.

A lot of research has been focused on investigating the different types of mistakes that are being made. Additionally, factors influencing the use of non-standard phraseology and what changes can be made to prevent future mistakes have been explored as well (Chang et al., 2007; Krifka et al., 2003; Molesworth and Estival, 2015). However, existing literature lacks sufficient insights on systems that automatically detect non-standard phrases pilots and controllers use while communicating. Detecting non-standard phrases automatically can help with analyzing the errors being made and improve future communication between pilots and controllers. As an example, a non-standard phrase detection system could be used to help pilots and controllers practicing the standard phraseology of ATC communication. One of the non-standard phrases used by pilots and controllers is the mispronunciation of the digit nine. According to the standard phraseology rules of ATC communication, the digit nine should be pronounced as NIN-er instead of NINE. Therefore, this thesis will focus on creating a system that can detect whether the digit nine is pronounced correctly. The mispronunciation of the digit nine is a small example of the overall non-standard phraseology and it is more ideal to detect a broader group of non-standard phraseology. However, detecting the mispronunciation of the digit nine is a good starting point for detecting non-standard phraseology. The purpose of this thesis is to build a system that can detect the mispronunciation of the digit nine by pilots and controllers.

To detect the mispronunciation of the digit nine, a keyword spotting system is built using the architecture proposed by Kim and Nam (2019). A keyword spotting system detects certain words or phrases from spoken utterances (see Section 1.2). Kim and Nam (2019) use a convolutional recurrent neural network with temporal feedback connections (TF-CRNN) to detect keywords (see Section 1.3). Their TF-CRNN architecture is inspired by the human auditory system and has shown good performances on keyword spotting. This thesis will investigate whether similar results can be achieved when the TF-CRNN architecture is used for detecting pronunciation mistakes on the digit nine.

1.1 ATC communication

Pilots and air traffic controllers play an important role in aviation to avoid accidents. The International Civil Aviation Organization (ICAO) has set several safeguards to avoid miscommunication in aviation. The ICAO is an agency of the United Nations, whose goal is to set principles and standards for international aviation safety and security. In 1951, the ICAO recommended having English as the universal language of air travel. The universal language was widely accepted and known as aviation English. Later, the ICAO added

a minimum level of English language proficiency for all pilots and controllers to their policy. This proficiency level is added to reduce miscommunication and misunderstanding on international flights (ICAO, 2003).

Another safeguard the ICAO has set are the international standards of phraseology, which are published in ICAO Annex 10 Volume II Chapter 5 (ICAO, 2001) and in ICAO Doc 9432 - Manual of Radiotelephony (ICAO, 2007). Aviation English does not follow the same formal grammar rules as the normal English language, but it follows a standard phraseology with informal grammar to minimize miscommunication. The standard phraseology has a rigid structure of utterances because communication through radiotelephony is not always clear and has a noisy environment. The rigid structure helps pilot and controllers understand each other and send their message to the receiver in a short clear way. The standard phraseology states how pilots and controllers should communicate during all phases of a flight, including pre-departure, taxiing, take-off, cruising, and landing. Not only the word choice is important, but also the pronunciation. Certain numbers and letters can sound similar to each other through a noisy environment. Furthermore, accents and dialects also make it difficult to identify certain numbers and letters. For example, the pronunciation of nine (/ˈnɪnə/) is similar to the German word 'nein' (/naɪn/) which means no. This lead to the standard pronunciation niner (/ˈnaɪnə(ɹ)/) for the digit nine, which is shown in Table 1 together with the other standard pronunciations for digits. The pronunciations for the digits are important, because most numbers should be transmitted by pronouncing each digit separately. For example, the number 29 should be pronounced as 'TOO NIN-er'. One exception where digits do not need to be pronounced separately is when the numbers contain whole hundreds and thousands. For example, the numbers 900 and 9000 are pronounced as 'NIN-er HUN-dred' and 'NIN-er TOUSAND'. Pilots and controllers need to follow the standard pronunciation to reduce confusion.

Numeral or numeral element	**Pronunciation**
0	ZE-RO
1	WUN
2	TOO
3	TREE
4	FOW-er
5	FIFE
6	SIX
7	SEV-en
8	AIT
9	NIN-er
Decimal	DAY-SEE-MAL
Hundred	HUN-dred
Thousand	TOUSAND

Table 1: The standard pronunciation of numbers according to the ICAO. The syllables in capital letters are stressed.(ICAO, 2007)

Despite the existing safeguards to protect against miscommunication, many aviation accidents involve miscommunication. Krifka et al. (2003) even concluded miscommunication in general aviation to be one of the leading causes of accidents. The main reason is that pilots and controllers still make use of non-standard phraseology. As mentioned earlier, one of the biggest aviation disasters happened in 1977 when the KLM Boeing 747 crashed into the Pan Am Boeing 747 on the runway in Tenerife. The main cause of this accident involves miscommunication. The Dutch pilot from KLM communicated "We are now at take-off", while his aircraft started rolling down the runway. In this case the pilot used the non-standard phrase "at take-off" instead of "taking off". This is an example of code-switching, where the pilot uses Dutch grammar with

English words. Verbs that are expressed with the suffix "-ing" in English are equivalent to the use of "at + infinitive" in the Dutch grammar. However, the Spanish controller was not familiar with the Dutch grammar and misunderstood the statement. The controller thought the aircraft was at the takeoff point waiting for instructions and did not know the aircraft was already taking off. Therefore, the controller did not warn the pilot about another aircraft that was already on the runway. This example shows a simple communication mistake can lead to a fatal aviation accident. According to a study on phraseology conducted by the **?**, 44 percent of the pilots experience non-standard phraseology at least once per flight. Even with a communication protocol, it is difficult for pilots and controllers to always use the standards phraseology.

1.2 Keyword spotting

Keyword spotting (KWS), also known as trigger word detection, is an audio mining technique used for automatically detecting predefined keywords in speech signals. A basic representation of a KWS system is illustrated in Figure 1. The KWS system uses a speech signal and a list of keywords as input and will return the positions of the keywords. Keyword spotting is also related to speech recognition, because both analyze speech signals to transcribe audio to text. The main difference is the number of known words in the transcription vocabulary. Speech recognition has a big number of known words and it occasionally contains a few unknown words in a stream of known words. However, keyword spotting has a small number of known words in a stream of unknown words. Thus, keyword spotting is a special case of speech-to-text transcription (Thambiratnam, 2005).

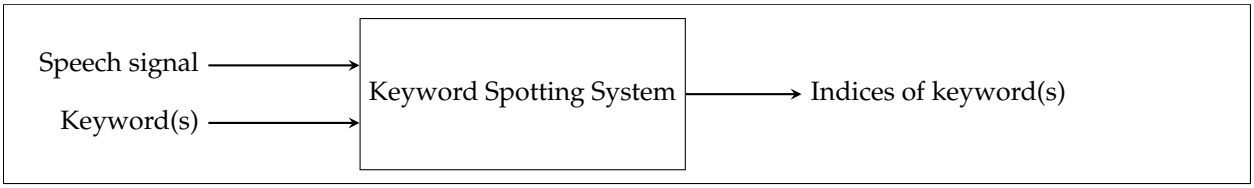


Figure 1. A basic representation of a keyword spotting system

Keyword spotting can be useful and is used in many applications. A well-known and commonly used application is the virtual assistant, such as Apple Siri, Google Assistant and Amazon Alexa. These virtual assistants are command controlled devices. Users need to say "Hi, Siri", "Hello, Google" or "Hey, Alexa" to get attention from the virtual assistant. However, keyword spotting is also implemented in many other different applications, like telephone routing, spoken password verification and audio document indexing (Thambiratnam, 2005). In this thesis, keyword spotting will be used to detect the pronunciation of the word 'nine' by pilots and controllers. The keywords are 'nine' and 'niner', because these are the two pronunciations being considered. All the other words are grouped with the same category 'non-keyword'.

There exist various approaches to implement keyword spotting. One of the first keyword spotting systems was built using Hidden Markov Models (HMMs) (Rohlicek et al., 1989). From then on, HMMs have been implemented in many other works of keyword spotting as well. The fundamental idea behind the Hidden Markov Model approach is having a model for keyword speech and a model for non-keyword speech. The last model is also referred to as filler or garbage HMM in literature. The probability for each utterance is calculated to see if the utterance is closer to the keyword or non-keyword. Many researchers have successfully implemented keyword spotting using HMM-based approaches (Rose and Paul, 1990; Wilpon et al., 1990; Xu et al., 2004). However, HMM is very time-consuming and requires a large amount of training data. Furthermore, HMM has low flexibility because it requires retraining when new keywords are added (Wöllmer et al., 2009). These drawbacks of HMM has led to the use of Dynamic Time Warping (DTW) for keyword spotting. Barakat et al. (2011) showed with their experiments that DTW performs better than

HMM when only a few examples of the keyword are available in the training data. The Dynamic Time Warping approach does not require any information about the language or any modelling and training. In the basic DTW approach, each keyword is represented by a template. The segments of utterances are compared against the templates to get the similarity between the keyword and the spoken utterance. Although DTW overcomes some of the limitations from HMM, the drawbacks of DTW are its computation complexity and estimation of the similarity threshold (Wu and Liu, 2003).

In recent years, the advances in deep learning lead researchers to use Artificial Neural Networks (ANN) to achieve keyword spotting. ANNs are networks inspired by the brain. An ANN has an input layer, one or more hidden layers and an output layer. Each layer consists of interconnected processing nodes that loosely represents the neurons in a brain. In ANN approaches, the speech signal is first preprocessed and feature extraction is applied. Feature extraction is the process where discriminative and dominant characteristics of the signal are collected. These features are used as input for the ANN. The ANN will be trained to get optimal weights for the nodes and predict the probabilities of the output layer. These probabilities are used to create a confidence score that corresponds to the keyword. Different types of Neural Networks have been applied to keyword spotting, like Deep Neural Network (DNN) (Chen et al., 2014), Convolutional Neural Network (CNN) (Sainath and Parada, 2015) and Convolutional Recurrent Neural Network (CRNN) (Zeng and Xiao, 2019). ANN approaches have shown significant improvement in keyword spotting. However, the limitation in ANNs is the need for a large amount of keyword training data to have a good performance.

Next to the approaches for KWS mentioned above, other approaches exist as well. Some are hybrid approaches that combine the strengths of two approaches. All approaches have their benefits and limitations. Depending on the dataset and goal of the application, some approaches might be better suited than others.

1.3 TF-CRNN for Keyword Spotting

Kim and Nam (2019) have proposed a model for keyword spotting using the TF-CRNN architecture. Their model successfully performs keyword spotting on speech commands from the Speech Commands dataset (Warden, 2018) with an accuracy rate of 96.00%. Furthermore, TF-CRNN has been compared to some other models and outperforms all of them. The main difference between TF-CRNN and previously proposed KWS systems is that TF-CRNN works more like the human auditory system. In the human auditory system, signal processing is bidirectional. There are afferent and efferent connections. The afferent connections are the feedforward pathway from the ear to the brain. The efferent connections are the feedback pathway from the brain to the ear (Lyon, 2017). Most of the previous works only implement the feedforward pathway of the auditory system, while TF-CRNN implements the feedback pathway as well. The way the feedback pathways are implemented resembles the mechanism of the outer-hair cells. Mainly due to the biologically-inspired aspects in TF-CRNN, this thesis will focus on this approach to implement keyword spotting.

The overall architecture of TF-CRNN is shown in Figure 2. The architecture is made up of a Convolutional Recurrent Neural Network (CRNN) with temporal feedback connections. By combining Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN), the model gains the advantages from both networks. CNN is used to extract local features and RNN is used to learn the temporal dependency in the audio signal. The CNN module is a deeply stacked CNN, which is based on the SampleCNN by Lee et al. (2019). This deeply stacked CNN with very small sizes of 1D convolution filters allows the CNN module to take raw speech signals as input. Therefore, prior feature extraction on the speech signal will not be necessary. The CNN module feeds the output of its last convolutional layer to the RNN module. The RNN module will produce hidden states at every time step. These hidden states are used as temporal feedbacks to scale the feature activations in the convolution blocks. At every time step, the hidden state will be fed

into the convolutional at the next time step to create scaled features. The output of the RNN module is connected to a fully connected layer that predicts whether the audio segment contains the keyword. A more detailed explanation of the implementation of this architecture is described in the Methods section.

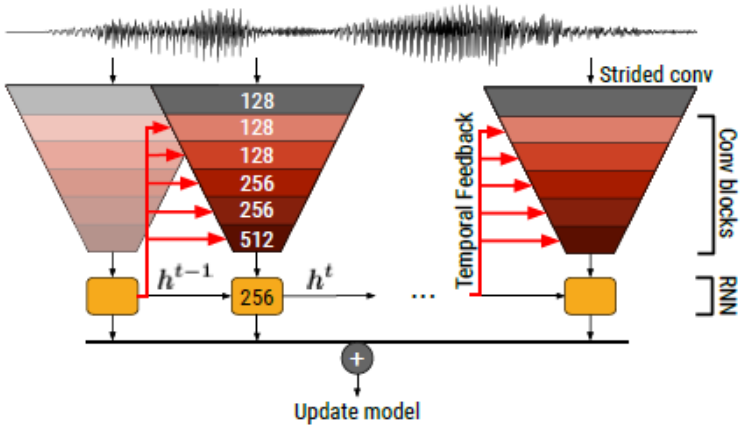


Figure 2. The temporal feedback convolutional recurrent neural network (TF-CRNN) architecture. The number in each convolutional block represents the number of filters. The hidden states of the RNN from the previous time step h^{t-1} are used as temporal feedbacks that are fed to the convolutional blocks in the current time step t. (Kim and Nam, 2019)

Kim and Nam (2019) have shown that TF-CRNN can classify keywords on speech commands adequately when the input audio signals only contain utterances of one spoken word. It would be interesting to see whether TF-CRNN can also be used to detect keywords when the input audio signal contains utterances of several spoken words. More specifically, this thesis will explore if the TF-CRNN model can classify a short ATC communication audio fragment on the existence of the keywords 'nine' and 'niner'. Therefore, the main question this thesis will investigate is whether it is possible to build a model with TF-CRNN that detects the mispronunciation of nine in ATC communication. Considering the high accuracy rates by Kim and Nam (2019), the hypothesis is that using TF-CRNN a keyword spotting system can be built to detect the mispronunciation of nine with an accuracy rate above chance level. Further questions that will be explored in this thesis are whether a similar accuracy as Kim and Nam can be achieved and whether the system is better at classifying one pronunciation over the other. For the first question, the accuracies should not be significantly different from each other, because the same architecture is being used. For the second question, the system should classify both pronunciations equally good because the system is not biased towards one pronunciation.

2 Methods

In order to build a model that detects the mispronunciation of the number nine in aviation, a dataset with ATC communication is needed as input (see Section 2.1). Several preprocessing steps on the dataset are necessary before the data can be used as input for the model (see Section 2.2). As mentioned before, the architecture of the model is the same as proposed by Kim and Nam (2019). However, the source code for the implementation of their TF-CRNN model has not been shared yet. Therefore, this thesis reimplements the TF-CRNN model based on the information provided in their paper (see Section 2.3). Furthermore, different approaches have been tried to improve the model performance(see Section 2.4). At last, the performance of the model is analyzed using evaluation metrics and statistical tests (see Section 2.5).

The model is built and trained using the open-source machine learning library PyTorch (Paszke et al., 2019),

which is based on the Python programming language. Additionally, the most relevant functions that are implemented in this thesis are attached in the Appendix B.

2.1 Dataset

The dataset used in this thesis is The Logan International (BOS) Corpus which is part of the Air Traffic Control Corpus (ATC0) dataset Godfrey (1994). The Logan Corpus contains approximately 20 hours of voice communication traffic between different pilots and controllers. The recorded communication is collected at Logan International Airport in Boston, Massachusetts. There are eleven audio files and each of them contains about 1 to 2 hours of audio signals. The audio signals are without silence elimination and have a sampling rate of 8 kHz, which means every second contains 8000 samples. The dataset also provides full transcripts for each audio file.

2.2 Dataset Preprocessing

The audio signals require preprocessing before it can be used as input for the model. An overview of the preprocessing steps is shown in Figure 3. The audio files need to be converted into a readable format and split into smaller audio files. The smaller audio files are subsequently labelled and transformed.

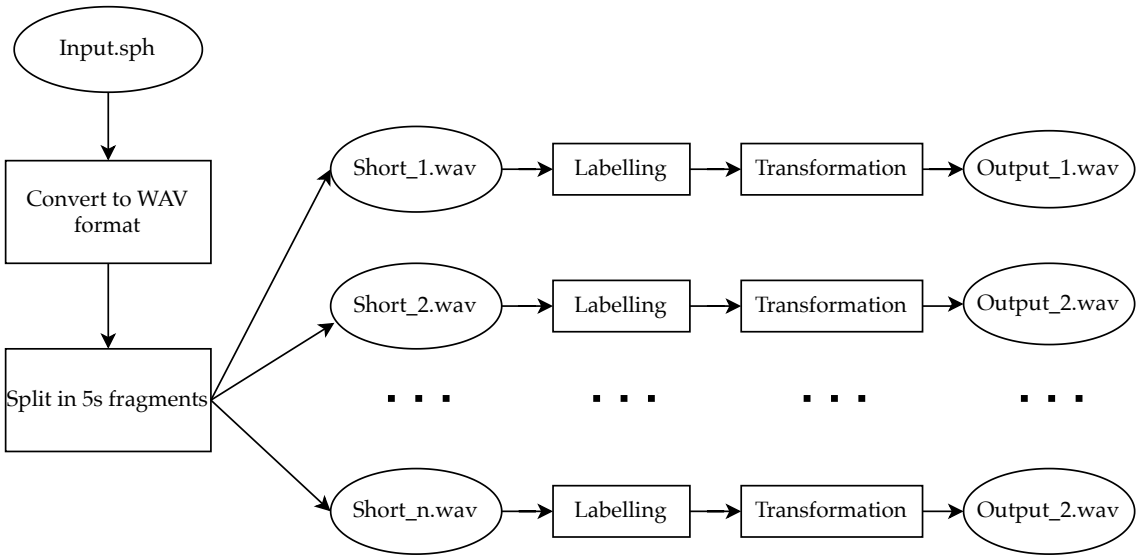


Figure 3. The preprocessing steps that every audio file goes through. The circles represent the audio files and the squared blocks represent the preprocessing steps. The "n" in the audio files denotes the number of audio fragments which are created after the splitting phase. This amount differs per input audio file because every input audio file has a different length.

The first two preprocessing steps are both performed using version 2.1.3 of Audacity® recording and editing software (2020). Audacity easily converts the audio files from the NIST sphere format (SPH) into the Waveform Audio File (WAV) format. Converting the audio files into the WAV format is necessary for the PyTorch library to read the audio signals. The second step is splitting the long audio files of 1-2 hours in smaller audio fragments of 5 seconds. An audio fragment which is too long can contain more utterances or noise which would obfuscate the pronunciation of the digit 'nine'. On the other hand, a very short audio fragment cannot capture enough utterances to detect the digit 'nine' in between the other words. Therefore

a length of 5 seconds is chosen, which would capture approximately one or two ATC messages. Not all fragments end up being exactly 5 seconds. Only seven fragments are shorter and these have been removed from the dataset. Due to the insignificant amount of these fragments and the fact that none of them contains the word 'nine', it is justifiable to remove them from the dataset. Hence after the first two preprocessing steps, there are in total 14,391 audio samples.

The third step in preprocessing is labelling all the audio samples manually. An audio sample can belong to three different categories: nine, niner and non-keyword. The 'nine' category applies to audio samples in which the digit nine occurs and is pronounced as "NINE". The 'niner' category applies to audio samples in which the digit nine occurs and is pronounced as "NIN-er". Thus, nine and niner are the two keywords the model should detect. The category 'non-keyword' applies to audio samples in which other utterances occur except the two keywords. Every audio sample is only assigned to one of the categories. Therefore, the model deals with a multiclass classification problem with three classes as categories Considering that multiclass classification algorithms work better with numerical values, integer labels are used instead of categorical string values to label the audio samples. The integer '0' is used for the category 'non-keyword'. The integer '1' is used for the category 'nine'. In the same way, the integer '2' is used for the category 'niner'. The audio samples are labelled by analyzing each audio sample together with its corresponding transcript. The collected data set contains 13,010 audio samples of the category 'non-keyword', 724 audio samples of the category 'nine' and 549 audio samples of the category 'niner'. This shows that the dataset is extremely imbalanced. The class imbalance is reasonable, because nine is generally not a frequently spoken word in the ATC communication. Nevertheless, different approaches to deal with the class imbalance problem exist. The techniques that are attempted in this thesis will be discussed in a further section (see Section 2.5).

There are some cases where the audio sample is not labelled as '1' or '2', even though the keywords are present. One of the cases is when both keywords – nine and niner – appear in one audio sample. These audio samples have been removed from the dataset, because adding them would make the classification over-complicated. It is not feasible to simply assign these audio samples with the labels '1' and '2'. By assigning multiple labels to an audio sample, the multiclass classification problem will be changed into a multilabel classification problem. This means that the audio samples will be assigned with a set of target labels instead of one label. Moreover, all audio samples then need to be assigned with the label '0' as well, because non-keywords occur in all samples. It is redundant to classify all samples as non-keywords as well. Additionally, multilabel classification draws the focus away from the main objective which is to detect a keyword in an audio fragment. Furthermore, it is also not reasonable to create a fourth category for audio samples that contain both keywords. There is only an insignificant part of the entire dataset – less than 0.008% – that contains samples with both keywords. Including these samples as a fourth category would make the already imbalanced dataset even more imbalanced. Considering the complication and the relatively small amount of audio samples involved, it seems acceptable to remove these samples with both keywords from the dataset.

Another case where the labelling deviates is when the pronunciations 'NINE-TEEN' and 'NINE-TY' occur in the audio sample. The pronunciation of these numbers contains the keyword 'nine'. However, the main mistake with the pronunciation of these numbers is not that the digit 9 is pronounced as "NINE" but that the digits are not separately pronounced. The number 19 should be pronounced as "WUN NIN-er" and the number 90 should be pronounced as "NIN-er ZE-RO". As a consequence of not separately pronouncing the digits, the digit 9 is pronounced incorrectly. The numbers nineteen and ninety are unlikely to be pronounced as "NIN-er-ty" and "NIN-er-TEEN". Furthermore, the mistake this thesis focusses on is when the digit 9 is pronounced as "NINE" instead of "NIN-er". Because the main mistake with 'NINE-TEEN' and 'NINE-TY' differs from the focused mistake, audio samples with these cases are not labelled with the class label '1'.

The last step of data preprocessing is transforming the data. The transformations of the data are imple-

mented in the ATCDataset class (Appendix B.2). The audio samples are first converted into a NumPy array of 32-bit float. Afterwards, each audio sample is framed into several short frames of 50 ms with an overlap of 50%. According to Kim and Nam (2019), a time step size of 50 ms with 50% overlap shows the best performance for TF-CRNNs. As mentioned before, every second in the audio samples contains 8000 samples. Thus, one audio frame of 50 ms contains $8000 * 0.05 = 400$ samples. The framing of the audio samples is illustrated in figure 4. The resulting shape of one audio sample is (199, 1, 400). Each audio sample contains 199 frames and each frame contains 400 samples. The framed audio samples are used as input for the model, where one frame is the input at one time step. Further transformations on the audio samples are not necessary because the model takes in raw waveforms.

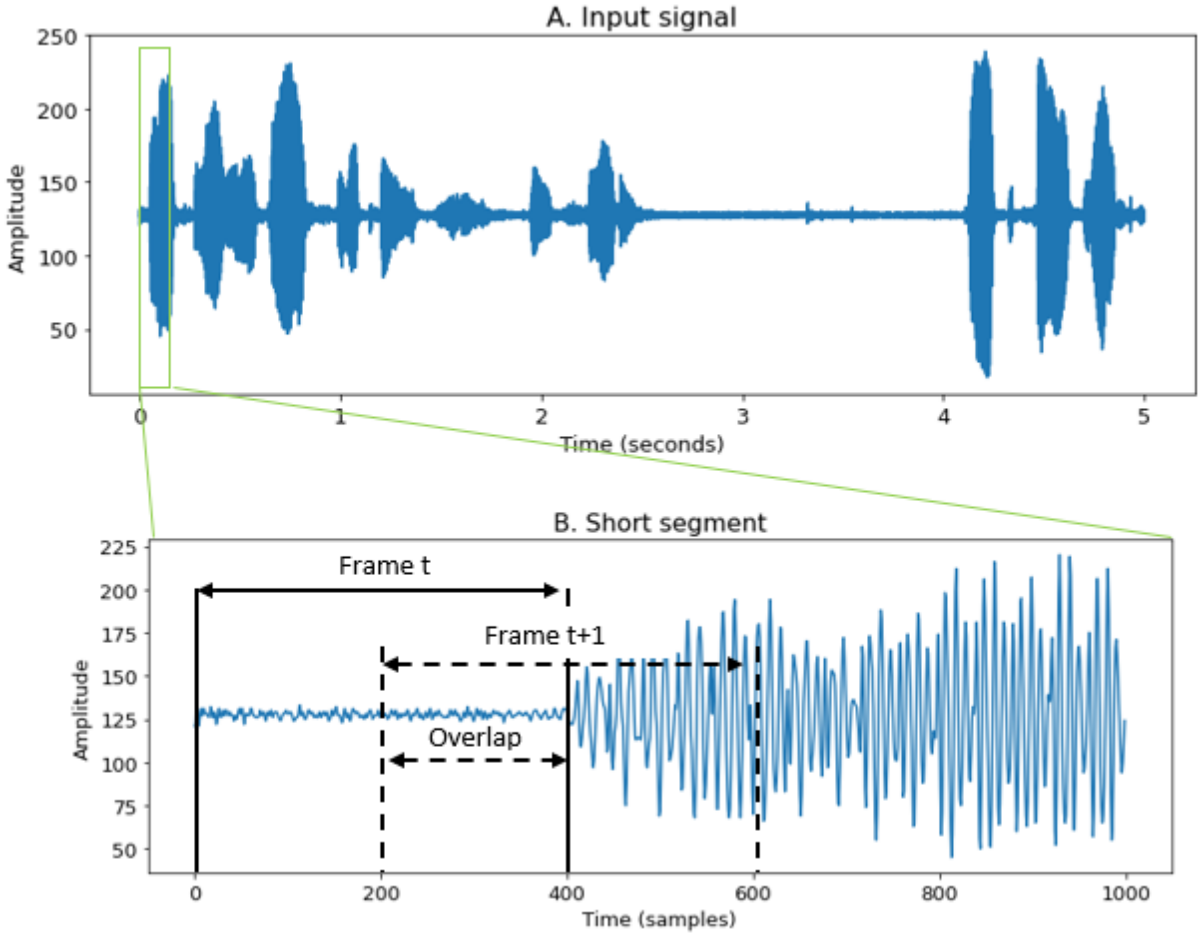


Figure 4. Plot A illustrates an example audio sample waveform. Plot B illustrates the framing of the audio samples. Every frame contains 400 samples and overlaps 200 samples with the preceding frame. Frame *t* is used as input for the current time step. Frame *t+1* is used as input for the next time step.

After preprocessing, the dataset is split into training, validation and test subsets. The training set is used to train the data. The model will see and learn from this data, which means the validation set is used to evaluate the trained model and fine-tune the model hyperparameters during the training process. The model will only see this data, but will not learn from it. The test set is used to evaluate the final trained model. In contrast to the validation set, the test set is only used when the model is completely trained. The dataset is split with a ratio of 80:10:10, 80% is for training, 10% is for validation and 10% is for testing. The corresponding number of data samples for each dataset is 11,439 for training, 1,415 for validation and 1,429 for

testing. Because the dataset is imbalanced, there is a high chance that the target class 'nine' or 'niner' is not present in the validation and/or test set when the dataset is being split randomly. Therefore, it is important to make sure each target class is equally represented in the training, validation and test sets. This is implemented in the `split_data()` function (Appendix B.1) by first keeping the different classes separate. Then, each class is randomly split into training, validation and test sets with the 80:10:10 ratio. After splitting, the subsets for training are concatenated together and the same is applied to the validation and test sets. In this way, the three subsets have a similar distribution of the different target classes (Appendix A). To use the training, validation and test sets in the model, the `ATCDataset` class is instantiated for each subset. Then, data loaders to iterate over the data in batches. These data loaders also shuffle the data and make sure the data is passed to the model.

2.3 Model implementation

There exist different deep learning frameworks that can be used to build the neural network. At first, Tensorflow (Abadi et al., 2016) has been used to build the model because it is one of the most popular frameworks. However, while implementing the model it has been realized that the model cannot be completely implemented with Tensorflow. The reason for this is that neural networks are static computation graphs in Tensorflow. This means the neural network has a fixed layer architecture and the entire computation graph of the model is only defined before running it. This aspect of Tensorflow makes it difficult to implement the TF-CRNN model because the TF-CRNN is a dynamic neural network. Thus, the temporal feedback aspect cannot be implemented using a static computation graph. On the contrary, PyTorch (Paszke et al., 2019) uses dynamic computational graphs, which allows the computation graph to be defined and modified. Therefore, the decision is made to change the framework by using PyTorch instead of Tensorflow.

The reimplementation of the TF-CRNN model contains a CNN module which is followed by an RNN layer in the TF-CRNN module(Appendix B.3). This is based on the information provided in the paper by Kim and Nam (2019). Most aspects of the model implementations were described distinctly. Their model consists of a CNN module and an RNN module. There are 6 convolution blocks in the CNN module. Every convolution block consists of a 1D convolution layer, Rectified Linear Unit (ReLU) activation function, batch normalization and max-pooling, except for the first block. The first convolution block does not have max-pooling but has a strided convolution layer with stride size 3 instead. Additionally, the convolutional layers have multiple filters with size 3 and max-pooling has pooling size 3. There are some components of the CNN module that has not been specified clearly. First of all, the order of the layers in the convolution block is not specified. Because the CNN module from original TF-CRNN is based on SampleCNN (Lee et al., 2019), the same sequence of layers as stated in the implementation of the SampleCNN is used. In SampleCNN, the 1D convolutional layers are followed by batch normalization, ReLU and max-pooling. Also, there are two hyperparameters in the convolution blocks that have not been specified explicitly. The first hyperparameter is the padding in the convolutional layers. The padding size is set to 1, which means zero is added on both sides of the input. Zero-padding is necessary for the convolution layers with stride size 1 to ensure that the input and the output dimensionality match. The second hyperparameter that has not been specified is the stride size in the max-pooling layer. Max-pooling requires a stride size to apply a max filter on subregions of the input representation. These subregions are commonly non-overlapping. To ensure non-overlapping regions, the stride size should be the same as the kernel size. Therefore the stride size is set to 3. Furthermore, the last convolution block does not contain max-pooling which contradicts with the original TF-CRNN. Max pooling is used to down-sample the input representation. However, the size of the input of the last convolution block is already reduced to size 1. Therefore, down-sampling any further is not possible. These changes and clarifications on the CNN module of the original TF-CRNN model are also visualized in Figure 5. This figure illustrates the complete model architecture, including its layers and parameters.

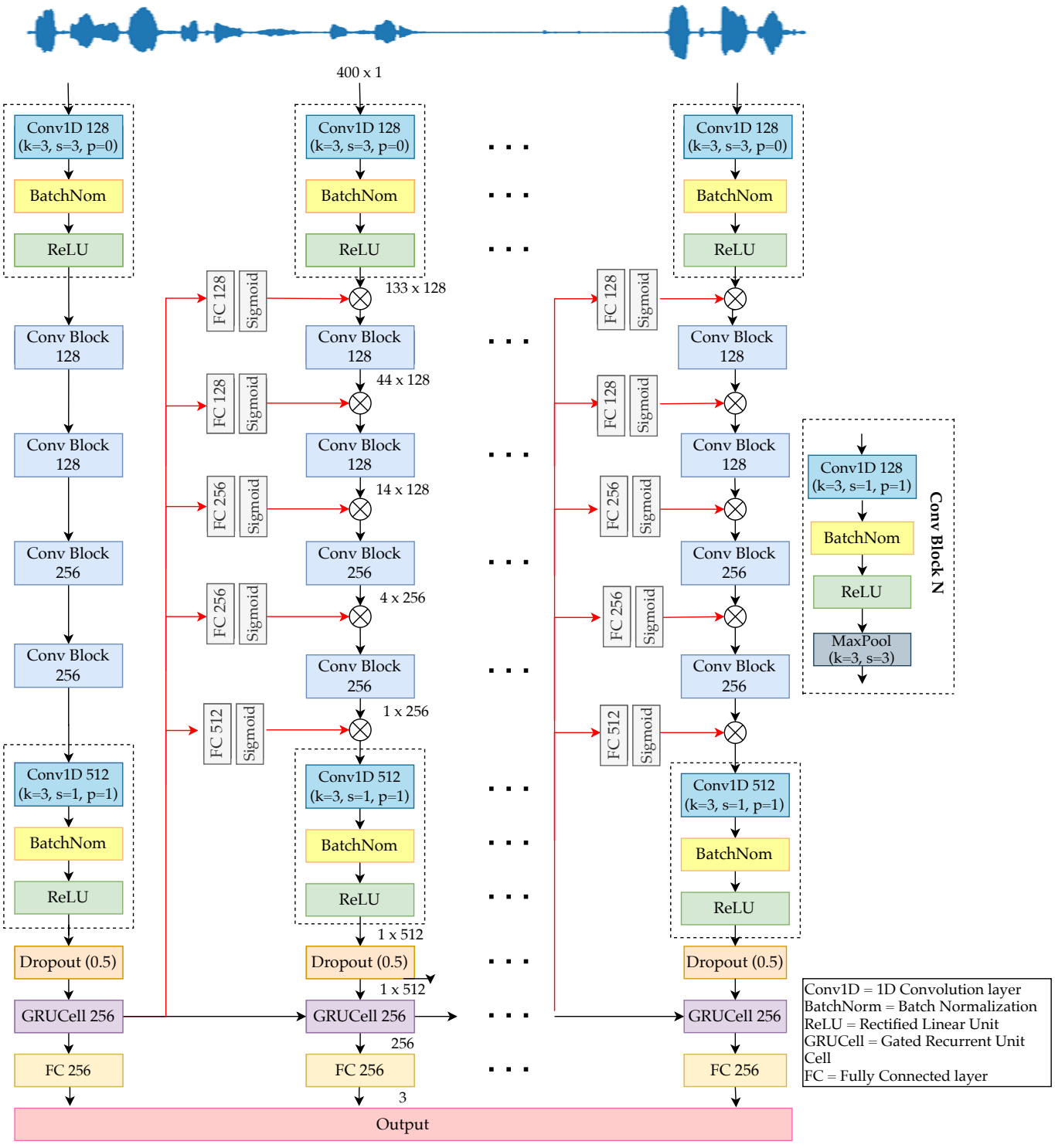


Figure 5. A schematic visualization of the adapted TF-CRNN architecture from Kim and Nam (2019). Every Conv1D has multiple filters, which is indicated by the number in the box. Additionally, it has a kernel size k, stride s, and padding p. Similarly, the max-pooling layers have a kernel size k and stride s. The dashed boxes represent the convolution blocks. The number in the GRUCell and FC denotes the number of features in these layers. The dimensions above the layers indicate the output shapes. Furthermore, the red arrows represent the temporal feedbacks used for feature scaling and the $\otimes$ denotes element-wise multiplication.

For the RNN module, Kim and Nam (2019) use a many-to-many setup (i.e., a synced sequence input and output) by replicating the label at every time step. According to the experiments by Kim and Nam (2019), a many-to-many RNN shows better results than a many-to-one RNN. Therefore, a many-to-many RNN is used in this thesis as well. Note that the many-to-many is only used in the training and validation phase. In the test phase, the many-to-one setup is used to predict the class label. Furthermore, they use Gated Recurrent Units (GRUs) to implement the RNN layer and initialize the hidden states with zeros. At each time step, the hidden states are used as temporal feedbacks. They will be fed into a fully connected layer and a sigmoid activation function to compute the scaling values. The fully connected layer ensures the scaling values to match with the dimensionality of the features in the convolution layer. The scaled values are then multiplied element-wise with the output of the convolution blocks at the next time step to create scaled features. This thesis uses GRU cells to implement the RNN module, because GRU cells are more flexible and allow the hidden state from the current time step to be used in the convolutional layer in the next time step. The outputs of the GRU cells are fed into a fully connected layer, which returns logits corresponding to the three classes. These logits are the unnormalized predictions of the model. The logits from all the time steps together form the output layer. The output will later be normalized using the softmax activation function, which converts the logits in the output layer into a probability distribution.

After building the model, the model is trained using the `train_model()` function (Appendix B.4). Similarly as Kim and Nam (2019), a stochastic gradient descent (SGD) with Nesterov momentum of 0.9 is used to optimize the model. However, they have not specified the loss function that is minimized by SGD. Since the thesis is dealing with a multi-class classification problem, the chosen loss function is the cross-entropy loss. This function calculates the model error. In Pytorch, the cross-entropy loss already includes the softmax function. Hence, it is unnecessary to implement the softmax activation function explicitly in the model architecture. Other training parameters based on Kim and Nam (2019) are the batch size of 23 and an initial learning rate of 0.1. The learning rate is decayed by factor 5 when the validation loss does not decrease for 3 epochs. Moreover, the training is stopped when the validation loss does not improve after ten epochs to avoid overfitting. Additionally, the best performing model is saved for testing, which is the model with the lowest validation loss.

2.4 Improving model

As mentioned earlier, our dataset is extremely imbalanced. The majority class is 'non-keyword' and the minority classes are 'nine' and 'niner'. This class imbalance can cause the model to perform poorly. Therefore, three common techniques for class imbalance are implemented to improve the model performance. The first technique is applying random oversampling on the training set. Random oversampling involves randomly selecting samples from the two minority classes with replacement and add them to the training dataset. This process continues until the number of samples in each minority class are equal to the number of samples in the majority class. As a result, the newly created training dataset is larger and balanced. In this thesis, the `RandomOverSampler` class from the Python library imbalanced-learn is used to implement random oversampling. It is crucial to apply random oversampling after the dataset is split. Otherwise, there is a high possibility that the validation and test sets contain the same samples as the training set. This will result in a model that does not learn how to classify but memorizes the data sample with its corresponding class label instead. Furthermore, the validation and test set need to be representative of real-life data, in which 'nine' and 'niner' are rare classes. In general, random oversampling is a good approach to address the class imbalance because the imbalance is eliminated. Instead of balancing the dataset, it is also possible to balance the batches. Therefore, the second technique to deal with the class imbalance is using a weighted random sampler in the data loaders. The `WeightedRandomSampler` class from Pytorch is used to implement this. This approach uses class weights on the data samples to rebalance the class distributions while sampling from the imbalanced training set. The class weights are computed as the reciprocal of the class amount. This results in class weights 9.49E-5, 1.71E-3 and 2.25E-3 for respectively 'non-keyword',

'nine' and 'niner'. Hence, the majority class will get a lower weight than the minority classes. The class weights do not need to sum to one, therefore they are not normalized. After computing the class weights, a list of weights corresponding to the training samples is created. The `WeightedRandomSampler` uses this list of weights to randomly sample on the training set with replacement. This results in balanced batches of data samples. This approach does not increase the training set, because it makes use of oversampling and undersampling. The minority classes are oversampled and the majority class is undersampled simultaneously until the size of the original training set is reached. The third technique to handle class imbalance is using class weights in the cross-entropy loss function. The same class weights as computed with the `WeightedRandomSampler` are used. Furthermore, applying weights to the loss penalizes the model for not classifying the minority classes correctly. The results of these three different approaches are compared to find out which approach is better suited for our model and dataset.

Also other approaches regarding the dataset have been implemented as an attempt to improve the model. One of these approaches is normalizing the training data. Kim and Nam (2019) mention that normalizing data is not necessary. Nevertheless, it is interesting to examine whether the model will perform better if the data is normalized since it usually is an essential preprocessing step. Another approach is exploring different ratios for splitting the dataset. Aside from the 80:10:10 ratio for the training, validation and test set, the ratios 70:15:15 and 60:20:20 are explored as well. Furthermore, approaches concerning the training hyperparameters are explored too. One hyperparameter is the initial learning rate which has been decreased to 0.01. The other hyperparameter is the batch size which has been increased to 32. At last, dropout layers have been added after each convolution block in the model architecture to reduce overfitting. These approaches are applied to investigate whether the model can be optimized further.

2.5 Analysis

During the training and validation phase, the loss and classification accuracy of the model on the training and validation set are saved using the `train_model()` function (Appendix B.4). The loss indicates how poorly the model predicts the samples. Furthermore, the classification accuracy is the ratio of correct predictions to total predictions made. These metrics are used to analyze the learning and generalization performances of the model. The loss is computed at every time step, because the model returns a prediction at every time step. However, the different time steps of one audio sample belong to the same class label. Therefore, the losses are averaged over the time steps to obtain the mean loss of one audio sample. To gain the final loss of the model at one epoch, the mean losses from all the audio samples are averaged. This final loss is saved and plotted in the loss curve. Similarly, the classification accuracy of the model is computed. At every time step, the predicted class label is compared to the true class label. Then, the classification accuracy of one audio sample is computed. The classification accuracies from all audio samples are averaged to obtain the final classification accuracy of the model for one epoch. This final classification accuracy is saved and plotted in the accuracy curve. The two learning curves show how the loss and classification accuracy of the model evolves over epochs for the training set and validation set.

After the training and validation phase, the model is evaluated in the test phase using the `test_model()` function (Appendix B.5). This will compute the classification accuracy of the model in the following way. First, the logits in the output layer of the model are converted into probabilities by using the softmax activation function. Then, the mean of these probabilities is computed to obtain the final probabilities of the three different class labels. The class label with the highest probability is the normalized prediction for the audio sample. From these predictions, the classification accuracy of the model is computed. The classification accuracy is a valid evaluation metric to use when the class imbalance is properly handled. However, the classification accuracy can be misleading when the class imbalance is not properly handled. Additionally, classification accuracy only gives a broad view of how our model performs. Therefore, some other evaluation metrics are computed as well to analyze the performance of the model. One of these metrics

is the unnormalized confusion matrix, which summarizes the correct and incorrect predictions. Furthermore, it provides an insight into the errors being made by the model. The normalized confusion matrix is also computed, where each class is represented as having 1 sample. When the dataset is imbalanced, the normalized confusion matrix gives a better insight into the errors. However, the precision and recall value of the model can be computed from the unnormalized confusion matrix. These metrics are useful as well when the dataset is imbalanced. These values are usually computed for a binary classification problem where one class is the negative class and one class is the positive class. To compute these values for an imbalanced multiclass classification problem, the majority class is the negative class and the minority classes are the positive class (i.e., 'non-keyword' is the negative class, while 'nine' and 'niner' are the positive class). Precision represents the ability of the model to return only relevant instances. It is the percentage of correct positive predictions from all positive predictions. On the other hand, recall represents the ability of the model to identify all relevant instances. It is the percentage of correct predictions from all positive predictions that could have been made. Thus, recall indicates the positive predictions that are missed by the model. The precision and recall values are not enough on its own to give insight into the performance of the model. Therefore, the F1 score is also computed, which is the harmonic mean of precision and recall. Ideally, the precision, recall and F1 score should be close to 1.0. These evaluation metrics will provide a good insight into how well the model performs.

Moreover, a statistical analysis is done on the classification accuracy of the model using a significance test. A one-sample t-test (two-sided) is used to determine the statistical significance of the model. The one-sample t-test compares the mean of a single group against a known mean. The null hypothesis H0 is that the classification accuracy is equal to the chance level accuracy of 33.33%. Additionally, the alternative hypothesis HA is that the classification accuracy is different from the accuracy at chance level. The one-sample t-test from the SciPy library Virtanen et al. (2020) is used to implement the t-test. To use this t-test, the model is evaluated 100 times on a different test set. The test sets are created by drawing a subset from the test dataset with replacement with the same length as the original test set. As a result, 100 distinct subsets are created on which the model will classify. The classification accuracy of the model is computed for every subset. The list with the classification accuracies will be passed on to the t-test. The one-sample t-test will then compute the mean classification accuracy and test it against a mean accuracy of 33.33%. The one-sample t-test returns a t-value and a corresponding p-value. In general, the t-value is the ratio of the difference between two groups and the difference within two groups. Thus, a bigger t-value means there is a bigger difference between the groups that are being compared. The results are more likely repeatable when the t-value is high. The p-value is the probability of obtaining the observed results by chance with a significance level $\alpha = 0.05$. If the p-value is less than the significance level ($p < .05$), the null hypothesis is rejected and the result is statistically significant. On the contrary, if the p-value is greater than the significance level ($p > .05$), the null hypothesis cannot be rejected and the result is statistically nonsignificant. Ideally, a permutation test should be used to determine the statistical significance of the model. However, the model is not trained on a null distribution. Therefore, the permutation test cannot be done. Nevertheless, comparing the results of our model to a chance level accuracy of 33.33% using one-sample t-test is almost similar and justifiable.

A one-sample t-test is also used to check whether the classification accuracy of our model is significantly different from the accuracy achieved by Kim and Nam (2019). This is implemented in a similar way as for testing the statistical significance of the model. Furthermore, an independent sample t-test (two-sided) is used to investigate whether the mean percentage of correctly classifying the classes 'nine' and 'niner' are significantly different (i.e., is the model better at classifying one class than the other class). The null hypothesis H0 is that there is no difference between the mean percentages and the alternative hypothesis HA is that there is a difference between the mean percentages. Comparable to the previous t-tests, the precisions of the two classes over 100 subsets are passed to the t-test. The independent sample t-test also returns a t-value and p-value. A p-value lower than the significance level ($p < .05$) indicates there is a significant difference in the results of the two classes.

3 Results

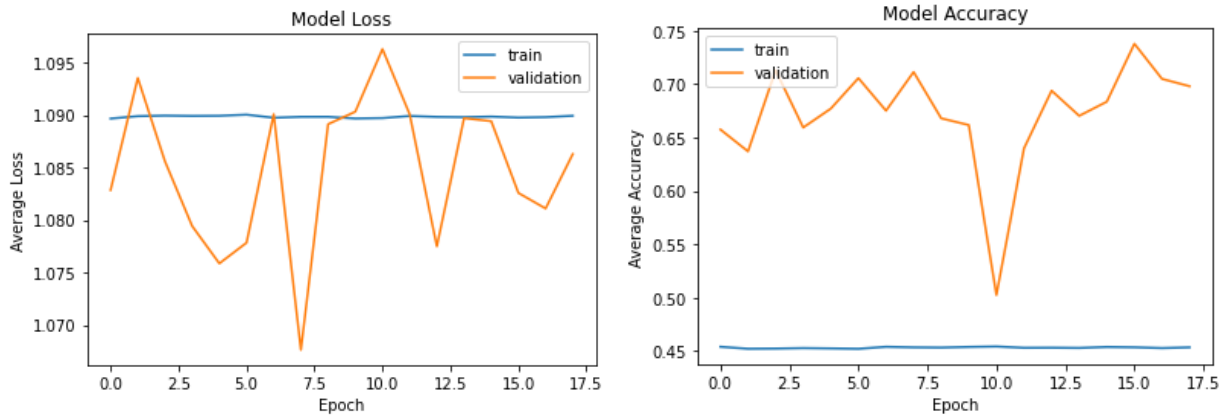
The performances of the model are shown in this section. The results of the model that is trained with and without the different class imbalance techniques are displayed in separate subsections (see Section 3.1, Section 3.2, Section 3.3, Section 3.4). These results include the learning curves and classification matrices. Furthermore, the three techniques are compared (see Section 3.5) to decide which technique is handling the class imbalance better. At last, the results of the significance tests on the best performing model are shown in Section 3.6.

The results from the different approaches to improve the model are not discussed thoroughly in this section, because they did not show any improvement of the model. Moreover, the results were not significant and do not add any value to this thesis.

3.1 Without class imbalance technique

The model that is trained without any class imbalance techniques stops training after 18 epochs. As shown in Figure 6, the model does not show any improvement with regards to the loss and classification accuracy. The training loss remains flat at a high loss value of approximately 1.09 and the validation loss fluctuates around 1.08 (Figure 6a). Similarly, the training accuracy remains stable around an accuracy value of 0.45 and the validation accuracy fluctuates around 0.67 (Figure 6b).

The model achieves a classification accuracy of 85.17% on the test dataset. However, the classification accuracy metric is misleading in this case, because the class imbalance is not dealt with. Therefore, the model has a strong bias towards classifying the data samples as the majority class 'non-keyword'. The strong bias is also visualized in the confusion matrices in Figure 7, which show the predictions of the model. 'Non-keyword' is correctly predicted for 1209 data samples, which is 93% of 'non-keyword'. Moreover, the label classes 'nine' and 'niner' have high probabilities of being misclassified as 'non-keyword', with percentages of 92% and 80% respectively.



(a) **(b)**

Figure 6. Graphs visualizing the learning curves of the model that is trained without any class imbalance techniques. (a)The evolution of the model's training and validation loss over epochs. (b)The evolution of the model's training and validation accuracy over epochs.

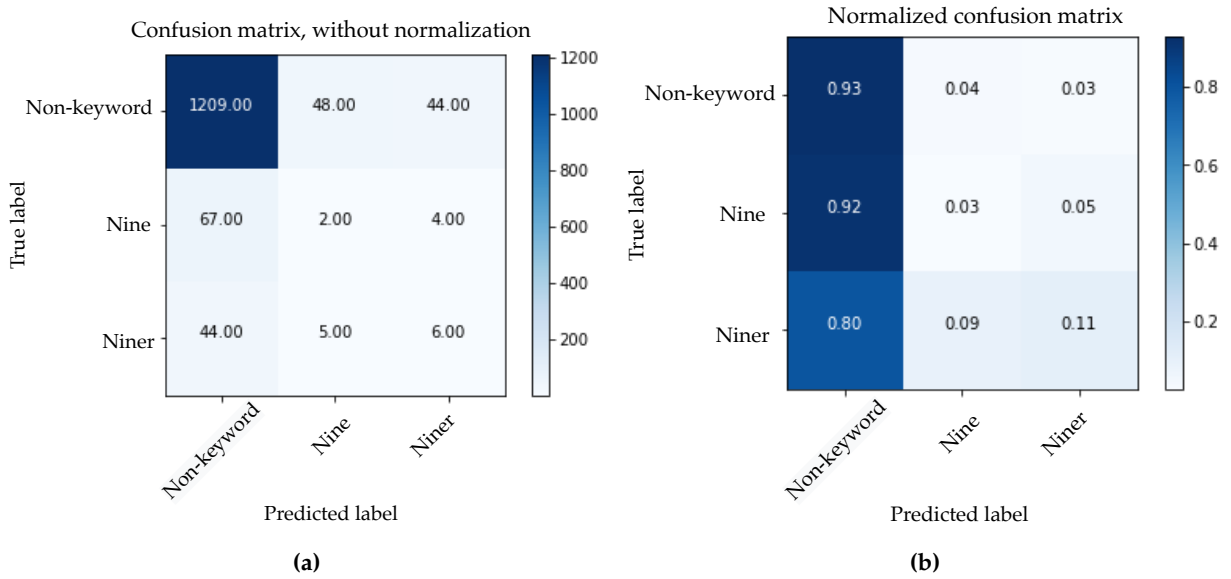
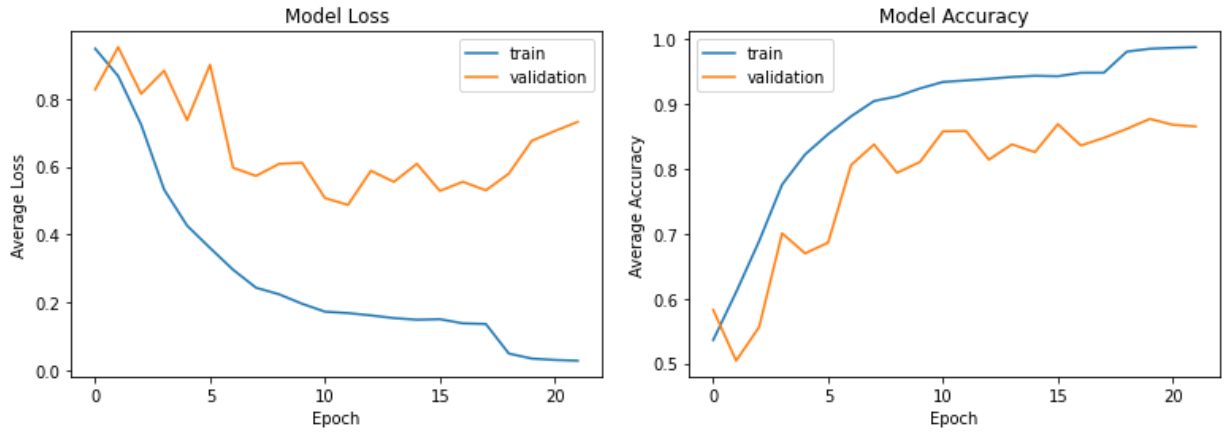


Figure 7. The unnormalized (a) and normalized (b) confusion matrix visualization for the model that is trained without class imbalance techniques over 1429 test samples.

3.2 Random oversampling

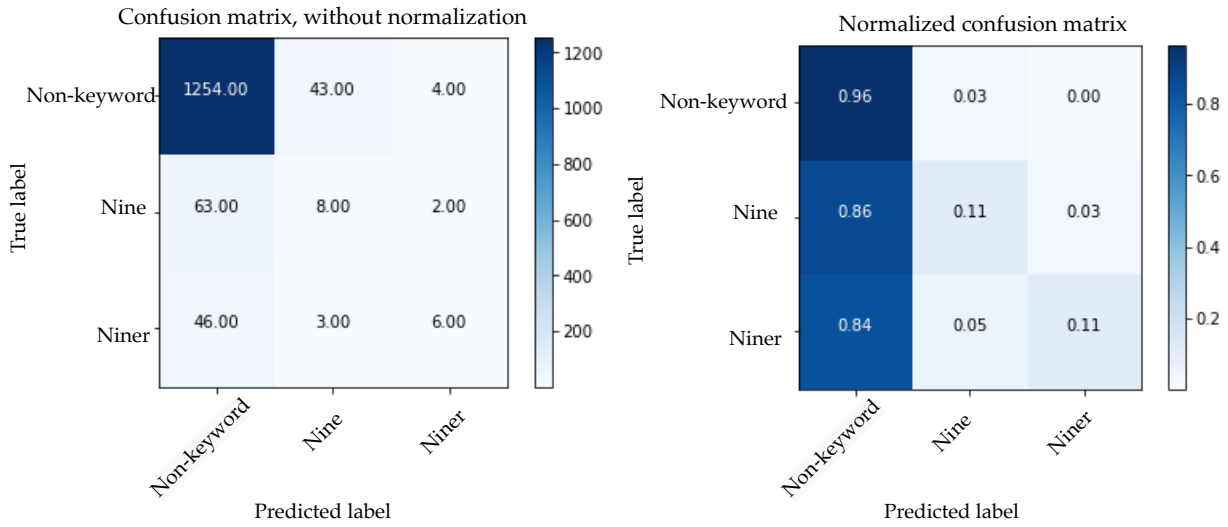
The model that is trained with random oversampling stops training after 22 epochs. As shown in Figure 8a, the training and validation loss are both decreasing over epochs and seem to stabilize. However, from epoch 18 onwards the training loss continues decreasing while the validation loss increases again. This is a sign of overfitting, where the model has learned the training set too well and is less good in generalizing to new data. Additionally, the validation loss is the lowest at the epoch 12, which indicates the model is best optimized at the epoch 12. Furthermore, Figure 8b shows that the training and validation accuracy are both increasing over epochs. Similar to the loss, the accuracy does not improve anymore from approximately epoch 12 onwards. Moreover, both graphs show a gap between training and validation results, where the training results are showing more improvement than the validation results. This could indicate an unrepresentative training dataset.

The model achieved a classification accuracy of 88.73% on the test dataset. The predictions of the model are visualized in the confusion matrices in Figure 9. Both confusion matrices show that the class 'non-keyword' is classified very well with 1254 correct predictions, which is 96% of 'non-keyword'. However, for the classes 'nine' and 'niner' only 11% are correctly predicted. Furthermore, the normalized confusion matrix shows that the class labels 'nine' and 'niner' are often misclassified as 'non-keyword' with respectively 86% and 84%. Hence, the model classifies almost all data samples as 'non-keyword'. This implies a strong bias of our model towards classifying audio samples as 'non-keyword'. Additionally, this indicates that the class imbalance has not been dealt with properly. Therefore, the high classification accuracy is a misleading metric for measuring the class performance in this case.



(a) **(b)**

Figure 8. Graphs visualizing the learning curves of the model that is trained with random oversampling. (a)The evolution of the model's training and validation loss over epochs. (b)The evolution of the model's training and validation accuracy over epochs.



(a) **(b)**

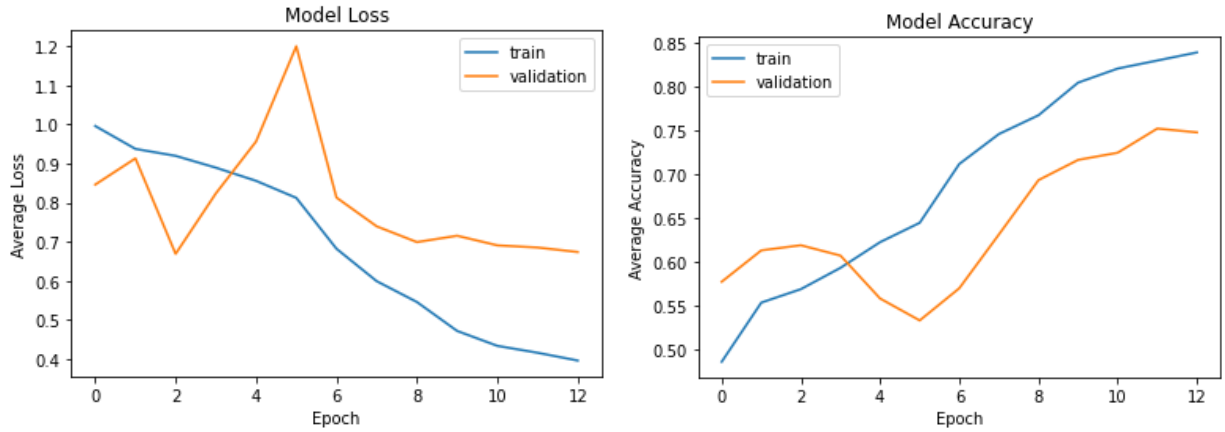
Figure 9. The unnormalized (a) and normalized (b) confusion matrix visualization for the model that is trained with random oversampling over 1429 test samples.

3.3 Weighted random sampling

The model that is trained with weighted random sampling stops training after 13 epochs. The validation loss does not decrease at the same speed as the training loss, as shown in Figure 10a. The training loss decreases smoothly, while the validation loss fluctuates a bit over the first few epochs. However, the validation loss shows small improvement overall. Additionally, the lowest validation loss is at epoch 2, which

indicates the model is best optimized at the epoch 2. Furthermore, the accuracy learning curve shows similar improvements as the loss learning curve. Figure 10b shows that the training accuracy is increasing smoothly, while the validation accuracy fluctuates a bit over the first few epochs. Moreover, both graphs show a gap between training and validation results, where the training results are showing more improvement than the validation results. This could indicate an unrepresentative training dataset.

The model achieved a classification accuracy of 62.56% on the test dataset. The predictions of the model are visualized in the confusion matrices in Figure 11. Both matrices show that the class 'non-keyword' is classified relatively well with 833 correct predictions, which is 64% of 'non-keyword'. Furthermore, the normalized confusion matrix also shows that approximately 68% of the class 'nine' is predicted correctly. However, only 20% of the class 'niner' is predicted correctly. The normalized confusion matrices also show that data samples are often misclassified as 'non-keyword' or 'nine', with percentages of respectively 54% and 90%. This implies a stronger bias of the model towards classifying audio samples as 'non-keyword' or 'nine' rather than 'niner'. Moreover, this indicates that the class imbalance has only been dealt with partially.



(a) **(b)**

Figure 10. Graphs visualizing the learning curves of the model that is trained with weighted random sampling. (a)The evolution of the model's training and validation loss over epochs. (b)The evolution of the model's training and validation accuracy over epochs.

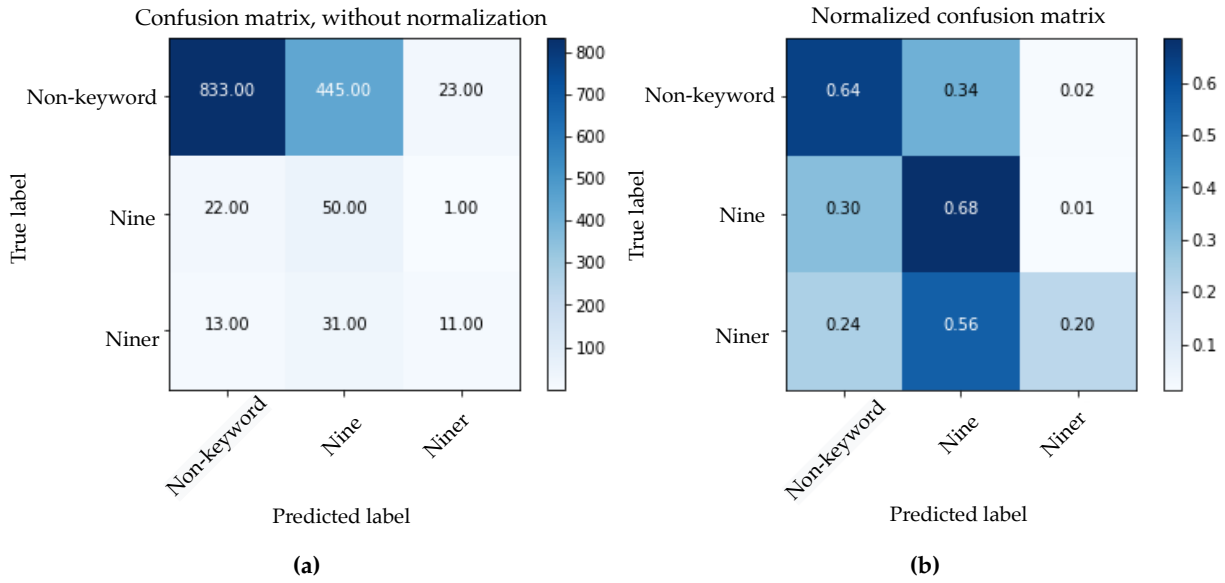
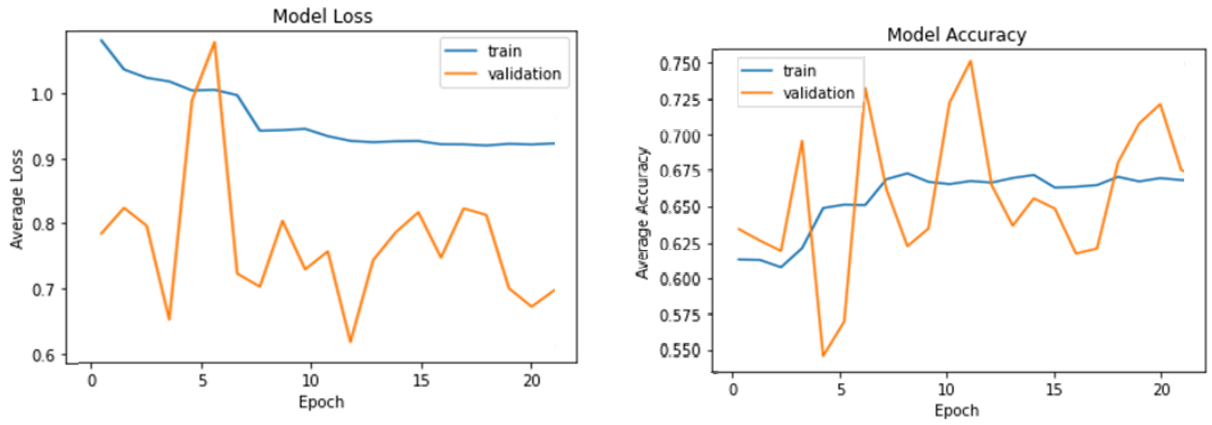


Figure 11. The unnormalized (a) and normalized (b) confusion matrix visualization for the model that is trained with weighted random sampling over 1429 test samples.

3.4 Weighted cross-entropy loss

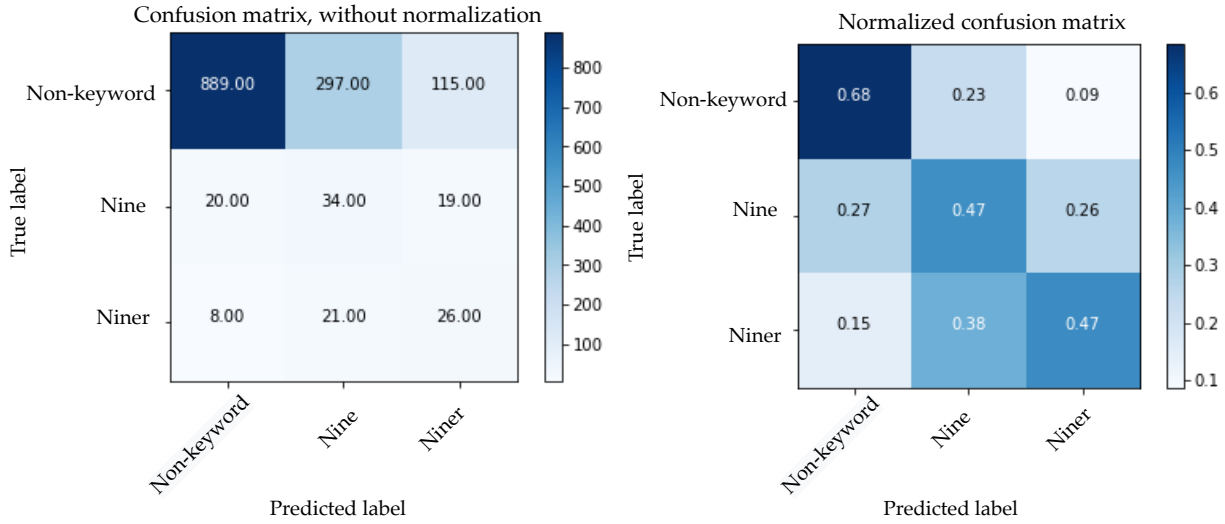
The model that is trained with weighted cross entropy-loss stops training after 21 epochs. As shown in Figure 12a, the training loss decreases smoothly and then stabilizes over epochs. On the contrary, the validation loss fluctuates a lot. Furthermore, there is a big gap between the training loss and validation loss, where the validation loss is lower than the training loss. This could imply that the validation dataset is unrepresentative. Additionally, the validation loss is the lowest after 11 epochs, which indicates the model is best optimized after 11 epochs. Furthermore, Figure 12b shows that the training accuracy gradually increases over epochs and stabilizes. Similar to the loss learning curve, the validation accuracy fluctuates a lot. Nevertheless, validation results are showing a small improvement in general.

The model achieved a classification accuracy of 66.41% on the test dataset. The predictions of the model are visualized in the confusion matrices in Figure 13. Both matrices show that the class 'non-keyword' is classified relatively well with 889 correct prediction, which is 68% of 'non-keyword'. Furthermore, the normalized confusion matrix also shows that approximately 47% of the classes 'nine' and 'niner' are correctly predicted. Even though, the percentage of correct predictions are higher than the chance level percentage (33.33%), the model still misclassifies a lot of samples. Additionally, percentages of the misclassifications in the lower triangle of the matrix is higher than the upper triangle. Approximately 42% of the audio samples are misclassified as 'non-keyword' and 61% is misclassified as 'nine'. This implies that our model is slightly biased towards classifying audio samples as 'non-keyword' and 'nine'. Moreover, this indicates that the class imbalance is being dealt with but there is still room for improvement.



(a) **(b)**

Figure 12. Graphs visualizing the learning curves of the model that is trained with weighted cross-entropy loss. (a)The evolution of the model's training and validation loss over epochs. (b)The evolution of the model's training and validation accuracy over epochs.



(a) **(b)**

Figure 13. The unnormalized (a) and normalized (b) confusion matrix visualization for the model that is trained with weighted cross-entropy loss over 1429 test samples.

3.5 Comparison

The previously discussed results have shown that including a class imbalance technique improves the model performance. Nevertheless, the model keeps a strong bias towards the majority class 'non-keyword' with random oversampling. However, the strong bias towards the majority class seems to decrease when weighted random sampling or weighted cross-entropy loss is applied.

As an additional comparison, the F1 scores of the models are compared to determine which technique results in the best performing model. Table 2 contains the F1 scores and the corresponding precision and recall values. As can be seen in the table, the F1 score for the model with weighted cross entropy-loss (0.19) is a bit higher than the F1 scores for the model with random oversampling (0.14) and weighted random sampling (0.16). This indicates that the model that is trained with weighted cross-entropy loss performs better than the models trained with other class imbalance techniques.

Class imbalance technique	Precision	Recall	F1 score
None	0.07	0.06	0.07
Random oversampling	0.21	0.11	0.14
Weighted random sampling	0.10	0.48	0.16
Weighted corss-entropy loss	0.12	0.47	0.19

Table 2: Precision, recall and F1 scores for the model with the three different class imbalance techniques.

3.6 Significance values

The different significance tests are only applied to the best performing model, which is the model with weighted cross-entropy loss. The result of the one-sample t-test indicates that there is a significant difference between the mean classification accuracy of our model and the chance level accuracy, ($t(99) = 258.15, p < .001$). Furthermore, the one-sample t-test also indicates a significant difference between the mean classification accuracy of our model and the classification accuracy achieved by Kim and Nam (2019), ($t(99) = -229.24, p < .001$). Lastly, the result of the independent sample t-test indicates that there is not a significant difference in the percentages of correctly predicting the classes 'nine' and 'niner', ($t(99) = 0.23, p = .81$).

4 Conclusion

To conclude, the results of this thesis are not entirely consistent with the proposed hypothesis. The results show the best performance when our model is trained with weighted cross-entropy loss. Using the TF-CRNN architecture, the model can classify the correct and incorrect pronunciation of the digit nine with a classification accuracy of 66.41%, which is above chance level. Furthermore, the model classifies the two keywords 'nine' and 'niner' with percentages of correct classifications that are not significantly different. Thus, the model can detect 'nine' and 'niner' equally good, even though there is a slight bias towards classifying data samples as 'nine'. However, the results also show that our reimplementation of the TF-CRNN model achieves a significantly lower classification accuracy than achieved by Kim and Nam (2019).

5 Discussion

The classification accuracy of our best model (66.41%) is significantly lower than the accuracy achieved by Kim and Nam (96.00%). This difference in classification accuracy is mainly caused by the different datasets that are used. Compared to the dataset from Kim and Nam (2019), our dataset is limited and severely imbalanced. Taking these two factors into account, the performance of our model is considered to be relatively good. The dataset being too small and imbalanced are the two main factors causing the low classification accuracy in our model. Our dataset with 11,439 samples for training and 1,415 samples for validation is relatively small to properly train and validate a model. The loss curves in the Results section

also indicate that our dataset is too small, because there is a gap between the training and validation loss. The gap between the training and validation suggests that the training or validation set is unrepresentative. When the training set is unrepresentative, the model does not receive sufficient training samples to learn the problem. When the validation set is unrepresentative, the model does not receive sufficient validation samples to evaluate and generalize the model. Unrepresentative training and validation set makes it more difficult for the model to learn properly and predict correctly.

Furthermore, our dataset is extremely imbalanced which makes our model strongly biased towards classifying data samples as the majority class 'non-keyword'. Three different techniques are explored to deal with the class imbalance. The results indicate that the model performs best with the technique involving weighted cross-entropy loss. The class performance improves when weighted cross-entropy loss is applied. Additionally, the bias towards classifying the data sample as the majority class 'non-keyword' is reduced. However, the model seems to be slightly biased towards classifying a data sample as 'nine'. This indicates that class imbalance is not perfectly handled. An explanation for this could be that the chosen class imbalance technique is not specialized enough for our model and dataset in order to handle the class imbalance perfectly. Nevertheless, using weighted cross-entropy already improves the model performance significantly. More class imbalance techniques could be investigated on our model and dataset to improve the model performance.

Moreover, our data samples contain more utterances than only the keywords. Whereas, the data samples used by Kim and Nam (2019) only contain utterances from the keywords and some background noise. Also, pilots and controllers speak incredibly fast in radio transmission, which makes it already challenging to follow for humans. Hence, the speech rate in our data samples is significantly higher than the speech rate in data samples from Kim and Nam. These aspects can make it more difficult for the model to learn and generalize. Therefore, it can also contribute to the low classification accuracy of our model.

Other further experiments can be explored to improve the performance of our model. First of all, our model could be pre-trained with the same dataset as used by Kim and Nam (2019) to ensure our model learns and generalizes well. Their model has shown outstanding results and this thesis reimplements their model. Therefore, it is expected that pre-training our model with the dataset from Kim and Nam will result in a model that learns and generalizes well. Classifying our dataset with the pre-trained model should give a higher classification accuracy. Additionally, a grid search on different hyperparameters can be conducted to find the optimal hyperparameters for the model.

Further research could also investigate whether other approaches than keyword spotting are better in detecting the mispronunciation of the digit nine. Keyword spotting only focusses on detecting the digit nine. The advantage of this is that the model does not need to know the language and grammar of the input. Thus, the model does not take into account other input information, which makes the task computationally easier. However, it can also be a disadvantage that all other information is discarded. Other information provided in the input could potentially help with the detection. For example, words like 'flight level', 'heading' and 'runway', usually are followed by a series of digits in ATC communication. Therefore, it could be interesting to investigate whether a model could detect the digit nine better when it knows the grammar and language of the input. In that case, the context will be taken into account as well in order to detect the digit nine.

Although this thesis only touches upon a small fraction of all the possible ATC communication mistakes, it provides a domain of possibilities for further research on this subject.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., (...), and Zheng, X. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *12th Symposium on Operating Systems Design and Implementation*, pages 265–283.

Audacity Team (2020). Audacity® software is copyright © 1999-2020 Audacity Team. The name Audacity® is a registered trademark of Dominic Mazzoni.

Barakat, M. S., Ritz, C. H., and Stirling, D. A. (2011). Keyword spotting based on the analysis of template matching distances. In *2011 5th International Conference on Signal Processing and Communication Systems (ICSPCS)*, pages 1–6, Honolulu, HI, USA. IEEE.

Chang, W., Wang, E. M., Tsai, W., Hsu, W., Yen, J., and Ho, H. (2007). A Human Factors Analysis of Miscommunication Between Pilots and Air Traffic Controllers in Taiwan. *2007 International Symposium on Aviation Psychology*, (():128–132.

Chen, G., Parada, C., and Heigold, G. (2014). Small-footprint keyword spotting using deep neural networks. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4087–4091, Florence, Italy. IEEE.

Godfrey, J. J. (1994). Air Traffic Control Complete LDC94S14A.

ICAO (2001). *Annex 10, Volume II*. ISBN: 5149548022, 6 edition.

ICAO (2003). Eigth Meeting of Civil Aviation Authoroties of the Authoroties of the Sam Region (RAAC/8). Technical report, Buenos Aires.

ICAO (2007). *Manual of Radiotelephony*. ISBN: 5149548022, 4 edition.

ICAO (2014). Annual Report of the ICAO Council: 2014. Technical report.

Kim, T. and Nam, J. (2019). Temporal Feedback Convolutional Recurrent Neural Networks for Keyword Spotting.

Krifka, M., Martens, S., and Schwarz, F. (2003). Group Interaction in the Cockpit: Some Linguistic Factors. pages 75–102.

Lee, J., Park, J., Kim, K. L., and Nam, J. (2019). Sample-level deep convolutional neural networks for music auto-tagging using raw waveforms. In *14th Sound and Music Computing Conference 2017 (SMC)*, pages 220–226, Espoo, Finland.

Lyon, R. F. (2017). *Human and Machine Hearing: Extracting Meaning from Sound*. Camebridge: Cambridge University Press, ISBN: 9781107007536.

Molesworth, B. R. and Estival, D. (2015). Miscommunication in general aviation: The influence of external factors on communication errors. *Safety Science*, 73:73–79.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., (...), and Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems 32*, pages 8024–8035.

Rohlicek, J. R., Russell, W., Roukos, S., and Gish, H. (1989). Continuous hidden Markov modeling for speaker-independent word spotting. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, volume 1, pages 627–630, Glasgow, UK. IEEE.

Rose, R. C. and Paul, D. B. (1990). A hidden Markov model based keyword recognition system. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, volume 1, pages 129–132, Albuquerque, NM, USA.

Said, H. (2011). Pilots/Air Traffic Controllers Phraseology Study 1. Technical report, IATA.

Sainath, T. N. and Parada, C. (2015). Convolutional neural networks for small-footprint keyword spotting. In *Annual Conference of the International Speech Communication Association, Interspeech*, pages 1478–1482, New York, USA.

Thambiratnam, A. J. K. (2005). *Acoustic keyword spotting in speech with applications to data mining*. PhD thesis, Queensland University of Technology.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R., Jones, E., Kern, R., Larson, E., (...), and Vázquez-Baeza, Y. (2020). SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods*, 17(3):261–272.

Warden, P. (2018). Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition.

Wilpon, J. G., Rabiner, L. R., Lee, C. H., and Goldman, E. R. (1990). Automatic Recognition of Keywords in Unconstrained Speech Using Hidden Markov Models. In *Transactions on Acoustics, Speech, and Signal Processing*, volume 38, pages 1870–1878. IEEE.

Wöllmer, M., Eyben, F., Schuller, B., and Rigoll, G. (2009). Robust vocabulary independent keyword spotting with graphical models. In *2009 IEEE Workshop on Automatic Speech Recognition and Understanding, ASRU 2009*, pages 349–353, Merano, Italy. IEEE.

Wu, Y. D. and Liu, B. L. (2003). Keyword spotting method based on speech feature space trace matching. In *2003 International Conference on Machine Learning and Cybernetics*, pages 3188–3192, Xi'an, China. IEEE.

Xu, M., Duan, L. Y., Cai, J., Chia, L. T., Xu, C., and Tian, Q. (2004). HMM-based audio keyword generation. In *Advances in Multimedia Information Processing - PCM 2004*, pages 566–574. Sprnger, Berlin, Heidelberg.

Zeng, M. and Xiao, N. (2019). Effective combination of DenseNet and BiLSTM for keyword spotting. *IEEE Access*, 7:10767–10775.

A Dataset distribution

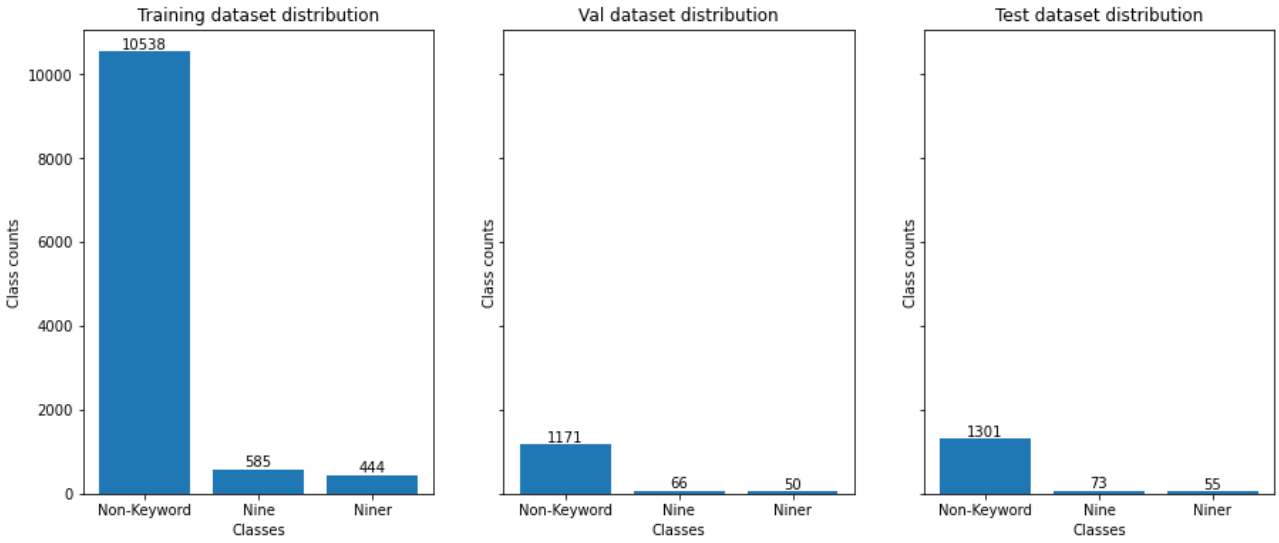


Figure 14. Histrogram visualizing the distributions of the classes 'non-keyword', 'nine' and 'niner' in the training, validation and test set

B Source code

B.1 Split dataset in train, val and test set

```
def split_data(nine_file, niner_file, non_keyword_file):
    #Analyze data in pandas
    nine_df = pd.read_csv(nine_file, delimiter=';')
    niner_df = pd.read_csv(niner_file, delimiter=';')
    non_keyword_df = pd.read_csv(non_keyword_file, delimiter=';')

    #Split data into train, val and test set
    nine_train, nine_test = train_test_split(nine_df, test_size=0.1)
    nine_train, nine_val = train_test_split(nine_train, test_size=0.1)

    niner_train, niner_test = train_test_split(niner_df, test_size=0.1)
    niner_train, niner_val = train_test_split(niner_train, test_size=0.1)

    non_keyword_train, non_keyword_test = train_test_split(non_keyword_df,
                                                           test_size=0.1)
    non_keyword_train, non_keyword_val = train_test_split(non_keyword_train,
                                                          test_size=0.1)

    #Concatenate separate train, val and test sets that belong together
    train_df = pd.concat([nine_train, niner_train,
                          non_keyword_train]).reset_index(drop=True)
    train_df = train_df.sample(frac=1).reset_index(drop=True)
    train_df['file_name'] = train_df['file_name'].astype(str) + '.wav'

    val_df = pd.concat([nine_val, niner_val,
                        non_keyword_val]).reset_index(drop=True)
    val_df = val_df.sample(frac=1).reset_index(drop=True)
```

```
    val_df['file_name'] = val_df['file_name'].astype(str) + '.wav'

    test_df = pd.concat([nine_test, niner_test,
                         non_keyword_test]).reset_index(drop=True)
    test_df = test_df.sample(frac=1).reset_index(drop=True)
    test_df['file_name'] = test_df['file_name'].astype(str) + '.wav'

    return train_df, val_df, test_df
```

B.2 Dataset class

```
class ATCDataset(Dataset):
    def __init__(self, audios, labels):
        self.x = audios
        self.y = labels

    def __len__(self):
        return len(self.y)

    def __getitem__(self, idx):
        #Transform audio
        data = self.x[idx]
        #Read audio
        audio_path = os.path.join(AUDIO_DIR, data)
        fs, audio = read(audio_path)
        #Convert audio file into numpy array float32
        audio = np.array(audio, dtype=np.float32)
        #Reshape
        audio = audio.reshape(1,-1)
        #Slide with 50% overlap and reshape
        audio_slided = np.array([audio[:,i:i+400] for i in range(0, 40000-200, 200)])

        #Transform label
        label = self.y[idx]
        label = torch.LongTensor([label])

        return(audio_slided, label)
```

B.3 TF-CRNN model

```
# Define CNN module
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        # 400 x 1
        self.conv1 = nn.Sequential(
            nn.Conv1d(1, 128, kernel_size=3, stride=3, padding=0),
            nn.BatchNorm1d(128),
            nn.ReLU())
        # 133 x 128
        self.conv2 = nn.Sequential(
            nn.Conv1d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.MaxPool1d(3, stride=3))
        # 44 x 128
        self.conv3 = nn.Sequential(
            nn.Conv1d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.MaxPool1d(3,stride=3))
        # 14 x 128
        self.conv4 = nn.Sequential(
```

```
            nn.Conv1d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.MaxPool1d(3,stride=3))
        # 4 x 256
        self.conv5 = nn.Sequential(
            nn.Conv1d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.MaxPool1d(3,stride=3))
        # 1 x 512
        self.conv6 = nn.Sequential(
            nn.Conv1d(256, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm1d(512),
            nn.ReLU(),
            nn.Dropout(0.5))

        #Temperal Feedbacks
        self.tf1 = nn.Sequential(
            nn.Linear(256, 128),
            nn.Sigmoid())
        self.tf2 = nn.Sequential(
            nn.Linear(256, 128),
            nn.Sigmoid())
        self.tf3 = nn.Sequential(
            nn.Linear(256, 256),
            nn.Sigmoid())
        self.tf4 = nn.Sequential(
            nn.Linear(256, 256),
            nn.Sigmoid())
        self.tf5 = nn.Sequential(
            nn.Linear(256, 512),
            nn.Sigmoid())

    def forward(self, x, h):
        b, _, _ = x.size()
        conv1 = self.conv1(x)
        tf1 = self.tf1(h)
        conv2 = self.conv2(conv1)
        tf_conv2 = conv2*(tf1.view(b, 128, 1))
        tf2 = self.tf2(h)
        conv3 = self.conv3(tf_conv2)
        tf_conv3 = conv3*(tf2.view(b, 128, 1))
        tf3 = self.tf3(h)
        conv4 = self.conv4(tf_conv3)
        tf_conv4 = conv4*(tf3.view(b, 256, 1))
        tf4 = self.tf4(h)
        conv5 = self.conv5(tf_conv4)
        tf_conv5 = conv5*(tf4.view(b, 256, 1))
        tf5 = self.tf5(h)
        conv6 = self.conv6(tf_conv5)
        tf_conv6 = conv6*(tf5.view(b, 512, 1))
        return tf_conv6

```

```
# Define TFCRNN model
class TFCRNN(nn.Module):
    def __init__(self):
        super(TFCRNN, self).__init__()
        self.cnn = CNN()
        self.rnn = nn.GRUCell(512, 256)
        self.fc = nn.Linear(256, 3)

    def forward(self, x):
        batch_size, time_steps, num_channel, width = x.size()
```

```
        h_t = self.__init__hidden(batch_size)
        logits = []
        for i in range(time_steps):
            cnn_out = self.cnn(x[:, i, :, :], h_t)
            cnn_out = cnn_out.squeeze(2)
            h_t = self.rnn(cnn_out, h_t)
            rnn_out = self.fc(h_t)
            logits.append(rnn_out)
        return logits

    def __init__hidden(self, batch_size):
        hidden = torch.zeros(batch_size, 256).to(device)
        return hidden
```

B.4 Train model

```
def train_model(model):
    # Early stopping initializiation
    epochs_no_improve = 0
    max_epochs_stop = 10
    min_val_loss = np.Inf

    # Preallocate memory for losses and accuracies
    train_loss = []
    train_acc = []
    val_loss = []
    val_acc = []

    # Loop over epochs
    for epoch in range(50):

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()
            if phase == 'val':
                model.eval()

            running_loss = 0.0
            running_corrects = 0.0

            # Iterate over data
            for data in data_loaders[phase]:
                # Get the inputs
                audios = data[0].to(device)
                labels = data[1].to(device)

                # Zero the parameter gradients
                optimizer.zero_grad()

                # Forward pass
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(audios)
                    loss = 0.0
                    corrects = 0.0
                    for timestep_pred in outputs:
                        # Get loss of one timestep
                        if phase == 'train':
                            loss += weight_criterion(timestep, labels.squeeze(1))
                        if phase == 'val':
                            loss += criterion(timestep, labels.squeeze(1))
                        # Get number of correct predicted labels of one timestep
                        _, preds = torch.max(timestep_pred, 1)
                        corrects += torch.sum(preds == labels.squeeze(1)).item()
```

```
                    average_loss = loss / len(outputs)
                    average_corrects = corrects / len(outputs)

                    # Backward pass + optimize only in training phase
                    if phase == 'train':
                        average_loss.backward()
                        optimizer.step()

                running_loss += average_loss.item()
                running_corrects += average_corrects
                running_acc = running_corrects / BATCH_SIZE

            epoch_loss = running_loss/len(data_loaders[phase])
            epoch_acc = running_acc/len(data_loaders[phase])

            if phase == 'train':
                train_loss.append(epoch_loss)
                train_acc.append(epoch_acc))

            if phase == 'val':
                val_loss.append(epoch_loss)
                val_acc.append(epoch_acc)
                lr_scheduler.step(running_loss)
                # Save model if val loss decreases
                if epoch_loss < min_val_loss:
                    torch.save(model.state_dict(), 'best-model.pt' )
                    epochs_no_improve = 0
                    min_val_loss = epoch_loss
                    best_epoch = epoch
                # Else increment count of epochs wihout improvement
                else:
                    epochs_no_improve += 1
                    if epochs_no_improve >= max_epochs_stop: model.load_state_dict(torch.load('
    best-model.pt'))
                        return model, train_loss, train_acc, val_loss, val_acc

    model.load_state_dict(torch.load('best-model.pt'))
    return model, train_loss, train_acc, val_loss, val_acc
```

B.5 Test model

```
def test_model(model):
    model.eval()
    correct = 0
    total = 0
    predictions = []
    true_labels = []
    with torch.no_grad():
        for data in test_loader:
            audios = data[0].to(device)
            labels = data[1].to(device)
            outputs = model(audios)

            # Get the average probabilities
            prob= []
            for timestep in outputs:
                prob.append(F.softmax(timestep, dim=1))
            mean_probs = torch.mean(torch.stack(prob), dim=0)

            # Get the predicted classes
            _, preds = torch.max(mean_probs, 1)
            predictions.append(preds)
            true_labels.append(labels.squeeze(1))

```

```
            # Check how many predictions equal to the real class
            correct += (preds == labels.squeeze(1)).sum().item()
            total += labels.size(0)

    # Calculate classification accuracy
    accuracy = 100 * correct / total
    return accuracy
```