Training a TTS model and implementing it on a robot

Table of Contents

Introduction

This short manual is intended both to relay the experience of developing, for the first time, a synthetic voice in Grunnegs, and its implementation on a robot. Additionally, it should also serve as a guide or a reference for those who might want to develop a synthetic voice for their own regional variety.

This project came about, as many do, at the request from stakeholders that were concerned that elderly citizens at a housing corporation were not engaging with technological solutions. As the elderly citizens are speakers of the regional variety of Dutch Low-Saxon spoken in the province of Groningen, broadly referred to as Grunnegs, besides Dutch, our team designed a study to look into the effect of using synthetic speech in their home language (Grunnegs) when compared to using synthetic speech in the state language (Dutch).

Although it might appear as though synthetic speech is a rather luxurious allocation of resources, synthetic speech comes to serve a number of purposes. It can be applied to aid the blind, to aid the deafened and vocally handicapped. Additionally, it can be used in educational applications, for example to aid dyslexic students. And, as in our case, there are multimedia applications for synthetic voices, which in combination with automatic speech recognition can be applied to achieve interactive multimedia platforms.

State-of-the-art speech synthesis is currently data driven (instead of e.g., rule-based). This means that to develop synthetic voices we need to collect appropriate data. Speech synthesis has been achieved following different techniques, that have been improving the intelligibility, naturalness and currently the expressiveness of the synthetic voices, during the last hundred years. With the application of neural networks, the required amount of data diminishes significantly, when compared to previous techniques such as unit selection. This is extremely promising for the communities of speakers of languages that did not have the resources previously to develop synthetic voices (nor other speech technologies in their language).

This manual comes to serve both as an account of the process that led to the training of the (to our knowledge) first Grunneger synthetic voice, and as a way to replicate our process, so that anyone who would want to train a voice in another variety can do so, granted that they have the necessary access to the required hardware. We take for granted that if anyone has access to the required hardware, they will have access to the required software, as the algorithms we have used are open source and the software that we used for the recording and editing is either available for free on the internet, or there are viable alternatives that are.

This manual is organized in two chapters. The first chapter "Grunneger synthetic voice" includes two main sections. "Building a corpus" describes both the process by which we planned the corpus and how it was recorded. "Training the algorithms" describes the process of using the data in the corpus to let the algorithm produce a model as a result of the process of training. The second chapter titled "Robot" includes a section that describes the process of selecting the robot for this project. The second part of this chapter is titled "Implementing the synthetic voices in the robot", where the process of getting the synthetic voice's trained model to work on the robot.

Grunneger synthetic voice

In this first section we will discuss how the corpus was planned, assembled and recorded. Additionally, some remarks will be included regarding recommendations derived from the experience picked up during this project. In the second section the training process is described.

Building a corpus

For this project we decided to adapt the methodology described in the Google AI blog, where their team describes the process of developing a Bangla synthetic voice. This seemed ideal, as the target language is an under-resourced language, such as Grunnegs, our target language.

An algorithm such as the one we chose to use, the Tacotron algorithm, is data-driven. This means that a corpus was needed to be assembled. A corpus comprised of both aural and textual data. Following the experience described in the Google AI Blog, we set out to assemble a corpus of two thousand sentences in the target language. This corpus would be then used to train the algorithm, which in turn would produce a model. The resulting model is a function that, given a certain input, in this case written Grunnegs (e.g., "Moi, ik bin Tammo"), produces a desired output, which in this case is spoke Grunnegs.

Planning the corpus

However, to do this we have to take some considerations. The corpus cannot be assembled in a random way. Some considerations are strictly related with the kind of data that the algorithm accepts. We could call these technical considerations. Because the algorithm is designed to find patterns between the audio files and the text files, which ultimately produces a model that accurately can process text into intelligible speech. Therefore, the first important consideration is to have the aural data and the textual data aligned, which means that there is a correspondence between the text files and the audio files, that is, that what is said in the audio files is exactly what is written in the text files. This is extremely important as we are going to input this data into the algorithm with the objective to produce a model that can accurately represent the regularities that exist between the spoken word and the written word. There are automated procedures to force align data. However, given the fact that the corpus we collected was relatively small, and forced aligned data still has to be checked, we decided to check the correspondence of the data manually.

For the same reason that we need the aural and textual data to correspond to each other, that is, allowing for the algorithm to establish regularities, there cannot be any background noise, which includes also background music. This might seem trivial, especially if seen from the perspective of human beings, which are very adept at differentiating speech from background noise or music. But a computer, a robot or an algorithm are very different.

There are potentially four standard ways to come up with the data (and of course all possible combinations of these four to different degrees). We need text and the corresponding audio. The first option is to write all the sentences and then record them. This might seem a straightforward option, but is problematic, as it is uncertain if the text produced is representative of the current language use of the community. A second possibility is to find existing pairs, for example, audio books are great. Most under-resourced languages do not have these or are very limited. Sometimes these exist but are not easily available, even when they are in the hands of public institutions. This is actually the case with our project. We were made aware of the existence of audio books recorded in Grunnegs in the 90's, but the public institution that holds these recordings has decided to constantly ignore our requests. A third option is to use radio or TV recordings. This is usually referred to as found data. However, transcription can be time-consuming, and there is no guarantee that there are no noises in the background of the audio that can be detrimental to the training process. A fourth option is to, on the basis of texts found (e.g., on the internet), record the audio files. This is, incidentally, the path we chose.

Due to the fact that Python can only deal with 16-bit audio files, this is the bitrates needed for the audio files. This means that either the audio files have to be recorded with this bitrate, or they have to be subsequently transformed from any other bitrate (e.g., 32-bit) to 16-bit. Additionally, the algorithm only works with tracks recorded mono, so if the audio files were recorded in Stereo, these have to be modified. A way to do this efficiently is to use software such as Adobe Media Encoder to do so, which might take from a couple of minutes to a couple of hours depending on your hardware and size of the corpus.

There are also linguistic considerations to be taken. Since the algorithm is designed to build a model that reflects the regularities between text and audio the corpus has to be as homogenous as possible. This means that the corpus assembled should be composed of a single speaker recording a single variety. Of course, the speaker can be proficient in multiple varieties, but during the recording of the corpus, it should all be as homogenous as possible. Or, if it is found data, the audio files selected should all be of one variety. We chose to work with Hoogelaandsters because we got in touch with a potential speech donor who had previous experience recording audiobooks in Grunnegs. We then selected from the corpus we had assembled, texts that reflected the spelling of this variety.

Regardless of whether the corpus was assembled by crawling the internet for texts or whether the texts were written by the team, it is necessary to use a consistent orthography. This is extremely important, not because of some prescriptive obsession, but because the algorithm will be looking for regularities. Therefore, unnecessary variation in aspects of the dataset that could be homogenous (and the algorithm expects to present a certain uniformity), will result in a low-quality model, which will very likely perform poorly when synthesizing speech. We decided to record texts we retrieved from the website of a literary journal. The use of the literary journal was beneficial, as editors tend to ensure a certain degree of uniformity in spelling, which is crucial in this case, as the algorithm is set to find regularities between the spoken and textual corpora one assembles.

A crucial aspect is to get in contact with potential “speech donors”. To do this it is important to survey the speech community and look for potential institutions that might have members that might be willing to collaborate with the project. We approached several theater associations which replied positively, although only one had someone available that was both willing to collaborate with our project and was available. In case of having multiple candidates, carrying out a short interview to ensure that we can choose the person that best fits the desired profile is an advisable step to take.

Recording the data

As mentioned in the previous section, to retrieve the necessary data to train the models needed for speech synthesis there are namely two sets of data that have to be produced: a written corpus and spoken corpus. Ideally, these two parallel corpora are available in the form of audio books, accompanied by the digitalized texts.

However, under-resourced languages such as is the case of Grunnegs at the moment, do not have the available infrastructure nor the existing resources to develop neither speech technology nor to localize software. The lack of resources means that we had to produce (at least) part of the data ourselves. Following the example of the experiment carried out at Google, and described in the Google AI Blog, recording the audio ourselves on the basis

To produce the required audio files, we recorded at the studios at the Faculty of Arts and at Campus Fryslân. Ideally, it has to be a room without noise, which also means no eco (anechoic studios).

We used Adobe Audition to edit the audio files (Audacity is also a possibility). To record we used SpeechRecorder (Clark & Bakos, 2015) developed at the University of Edinburgh. There are other options, such as Praat, which with a plug-in developed by Wilbert Heeringa presents similar functionalities to SpeechRecorder. SpeechRecoder allows to efficiently streamline the process of recording, automatically generating unique clips of each recording with a unique name and the corresponding text.

It took about 10 studio hours to record what resulted in 1,5 hours of material. Once the recordings were done, all the data had to be prepared for the next stage. We also recorded the same amount data in Dutch to use as control in our experiments.

To record the Grunneger corpus we got in contact with a woman of about 60 years old, who has lived her whole life in the northern area of the province of Groningen where Hoogelaandsters is spoken. She identified herself as a speaker of this variety.

The first session of recordings took place at the Faculty of Arts of the University of Groningen on August 6th, 2019. The second recording session took place on August 14th, 2019. The audios in these sessions were recorded in a proper studio, so there is no echo in the recordings, and the "speech donor" was in a soundproof room, with whom I could only communicate by pressing the talkback button on the console. In these first sessions, the audios were recorded in batches of fifty sentences, which were presented to the donor one sentence at a time on a power point, while the audio was recorded using Adobe Audition. This is a licensed program. A suitable option would have been to have used Audacity. This software is available on the internet to be downloaded free of charge.

Additionally, for our experiment we also needed a control condition, so we did a similar procedure to record a Dutch corpus. While the project was advancing, we still researched into existing software that might streamline recordings in a better way. Therefore, to assemble the Dutch corpus we decided to use SpeechRercorder, a software especially designed to assemble corpora that has speech synthesis for its objective. The advantage of this software is that it significantly reduced the amount of time that has to be destined to ready the data for the training stage.

The "speech donor" for the Dutch corpus was a woman of about 70 years old, who was born and raised in Amsterdam. The recordings took place at the Studio in the faculty building of Campus Fryslân, in Leeuwarden. This studio has been intended to film videos, and the acoustics are not ideal for audio recordings, but since the donor lived in Leeuwarden, this studio provided a noiseless environment where to carry out the recording sessions. Because of the availability of the donor the sessions were much shorter than the two fulltime sessions carried out to assemble the Grunneger corpus. Recordings took place on February 17th and 19th, and March 2nd, 5th, 9th and 12th 2020.

Once the data had been collected, we had to run a quality check, which basically meant listening again to all audios and make sure that what was being said in the audios was exactly what was written in the text files. Part of this process was being done while the recordings are taking place, as while the donor is reading out loud the sentences that appear on the screen, the person responsible for the recordings should be controlling this correspondence by wearing headphones and simultaneously listen to the input while it is being recorded.

Training a model

Introduction

To train a TTS model for the target speech variety, in this case, Grunnegs, certain steps have to be taken. First of all, is to have access to hardware that has enough computing power to process the amount of data in a reasonable amount of time. Because the training phase of achieving speech synthesis using Tacotron requires an enormous number of computations we started using GPUs instead of the CPUs, which if the algorithm allows for it, computes much faster.

We worked mainly with either the traditional Tacotron algorithm that produces a model that generates Mel wave spectrograms which are then used to synthesize speech by the vocoder, the Griffin-Lim algorithm. This algorithm needs no training, so the performance of this algorithm is very reliable. However, this vocoder does not synthesize the most natural sounding voices.

Additionally, we worked with the Tacotron2 implementation that combines the Tacotron algorithm that generates Mel spectrograms together with the WaveNet vocoder. This vocoder does need training on the data, which means more time computing, which is a downside when comparing it with the Griffin-Lim vocoder, but it produces a more pleasing output, that is also more natural sounding.

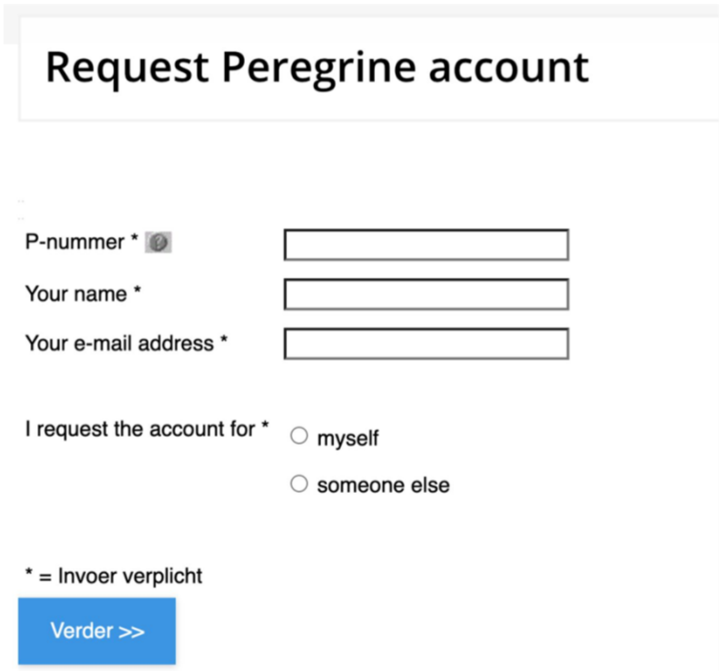
Initially we intended to train solely using the data that we had collected in Grunnegs, following the aforementioned Google AI post. It was very satisfying to be able to get results fairly quickly. However, the results were somewhat disappointing, as the intelligibility of the samples was very low, and it was a very unpleasant experience to the ear. Therefore, we looked at other options. Inspired by the promising results of a paper (link) that used data in different languages to develop a multilingual model, we decided to use a corpus in English (popularly known across the community dedicated to speech synthesis as LJ Speech), that is comprised of 24 hours of recorded speech, with its corresponding text. By doing so our system remarkably improved its performance. Not only did we end up with a robust text-to-speech system, but it was also remarkably more intelligible and sounded more natural like.

What we have been able to observe is that taking a trained model of the English data, in the form of the checkpoint files that the algorithm produces, and the continue the training process with the Grunneger corpus, even though the relationship in the amount of data is enormous (24 hours English data versus 1.5 hours Grunneger data), the resulting model, after a sufficient amount of training steps, is able to synthesize a voice that is intelligible and natural-sounding, which only rarely produces speech that might be tagged as English-sounding.

The challenge when training the model with the Grunneger (or the Dutch) data is to avoid **over fitting** (letting the algorithm to "over-train" the model). Over fitting means training the data to a point in which it perfectly represents the dataset. But since the data is a sample of the population, a perfect representation of the data would not be a which result in the model fitting perfectly the existing data but failing to perform well outside the words that compose the corpus. And without this being linear, as a general rule of thumb, the smaller the dataset, the bigger the number of inaccuracies that an over fitted model will present when applied on real world data, in this case words outside of the training set. The resulting output of the training stage are models that the algorithm creates in the form of checkpoints that can either be used to synthesize text, or as checkpoints continue the training procedure.

Requesting access to the HPC

The University of Groningen has a high-performance computing (HPC) cluster called Peregrine. This is available for employees that might need to use it in their research. An account has to be requested. The link for the form to request an account is: https://www.rug.nl/society-business/centre-for-information-technology/resear ch/services/hpc/facilities/request-peregrine-account.



Request Peregrine account

P-nummer *	
Your name *	
Your e-mail address *	
I request the account for *	○ myself ○ someone else

* = Invoer verplicht

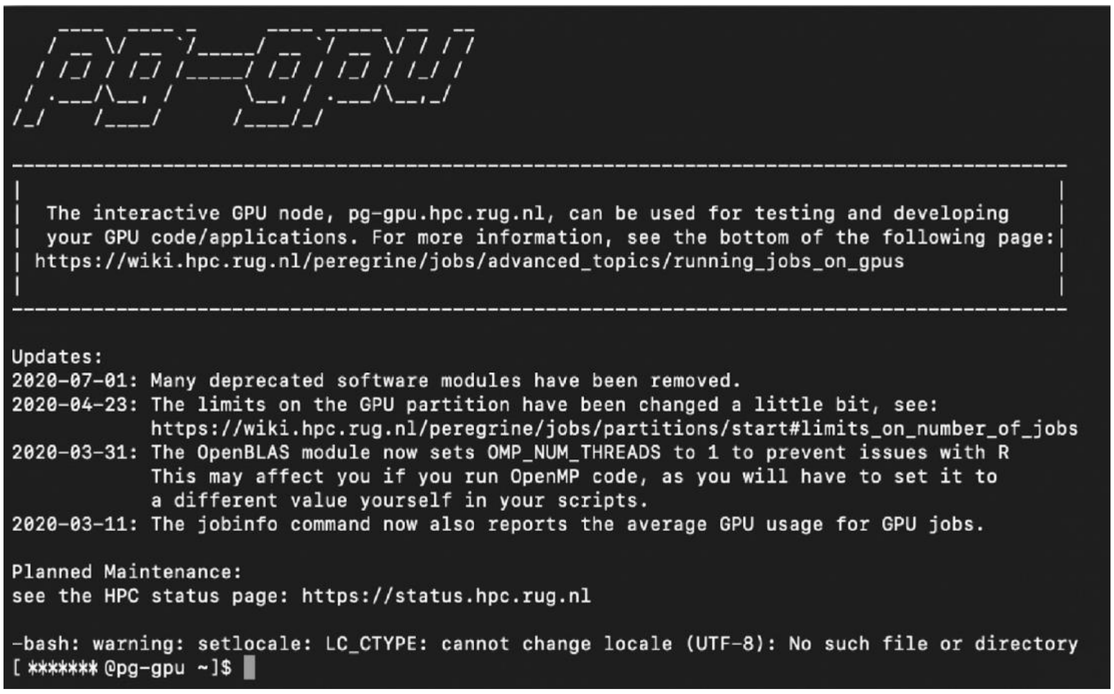
Figure 1. Screenshot of the online form to request a Peregrine account

Students are also allowed to request accounts given that they are part of a research project. More information on the Peregrine computing cluster can be found in their wiki: https://wiki.hpc.rug.nl/peregrine/start.

Once you have an account, to log in, type:

```
ssh username@peregrine.hpc.rug.nl
```

into the command line, replacing "username" for either your s-number or p-number. Once you are logged in you should see your username left of the command-line (see Figure 2).



```
    ____  ____ _      ____ _____  __  __
   / __ \/ __ `/_____/ __ `/ __ \/ / / /
  / /_/ / /_/ /_____/ /_/ / /_/ / /_/ /
 / .___/\__, /      \__, / .___/\__,_/
/_/    /____/      /____/_/

-----------------------------------------------------------------------------------------
|                                                                                       |
|  The interactive GPU node, pg-gpu.hpc.rug.nl, can be used for testing and developing  |
|  your GPU code/applications. For more information, see the bottom of the following page:|
| https://wiki.hpc.rug.nl/peregrine/jobs/advanced_topics/running_jobs_on_gpus           |
|                                                                                       |
-----------------------------------------------------------------------------------------

Updates:
2020-07-01: Many deprecated software modules have been removed.
2020-04-23: The limits on the GPU partition have been changed a little bit, see:
            https://wiki.hpc.rug.nl/peregrine/jobs/partitions/start#limits_on_number_of_jobs
2020-03-31: The OpenBLAS module now sets OMP_NUM_THREADS to 1 to prevent issues with R
            This may affect you if you run OpenMP code, as you will have to set it to
            a different value yourself in your scripts.
2020-03-11: The jobinfo command now also reports the average GPU usage for GPU jobs.

Planned Maintenance:
see the HPC status page: https://status.hpc.rug.nl

-bash: warning: setlocale: LC_CTYPE: cannot change locale (UTF-8): No such file or directory
[ ****** @pg-gpu ~]$ █
```

Figure 2. After successfully logging in you will see a similar screen.

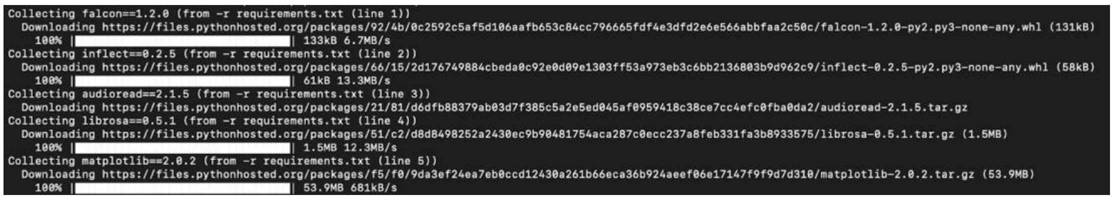
Installing dependencies

One of the requirements to run Tacotron2 to train our TTS model is Python3. The Peregrine cluster already has Python3 installed, so this step is already done. Should you run this on a cluster other than Peregrine, check that the correct version of Python is available.

The python scripts in this repository require TensorFlow 1.4. If you are working on your own HPC cluster, you can download the latest version of TensorFlow. It is advisable to install it with GPU support. Otherwise, you can ask your clusters admin whether the required version is installed.

To install the requirements needed to run the python scripts, type the following into the command-line:

```
pip install -r requirements.txt
```



```
Collecting falcon==1.2.0 (from -r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/92/4b/0c2592c5af5d106aafb653c84cc796665fdf4e3dfd2e6e566abbfaa2c50c/falcon-1.2.0-py2.py3-none-any.whl (131kB)
    100% |████████████████████████████████| 133kB 6.7MB/s
Collecting inflect==0.2.5 (from -r requirements.txt (line 2))
  Downloading https://files.pythonhosted.org/packages/66/15/2d176749884cbeda0c92e0d09e1303ff53a973eb3c6bb2136803b9d962c9/inflect-0.2.5-py2.py3-none-any.whl (58kB)
    100% |████████████████████████████████| 61kB 13.3MB/s
Collecting audioread==2.1.5 (from -r requirements.txt (line 3))
  Downloading https://files.pythonhosted.org/packages/21/81/d6dfb88379ab03d7f385c5a2e5ed045af0959418c38ce7cc4efc0fba0da2/audioread-2.1.5.tar.gz
Collecting librosa==0.5.1 (from -r requirements.txt (line 4))
  Downloading https://files.pythonhosted.org/packages/51/c2/d8d8498252a2430ec9b90481754aca287c0ecc237a8feb331fa3b8933575/librosa-0.5.1.tar.gz (1.5MB)
    100% |████████████████████████████████| 1.5MB 12.3MB/s
Collecting matplotlib==2.0.2 (from -r requirements.txt (line 5))
  Downloading https://files.pythonhosted.org/packages/f5/f0/9da3ef24ea7eb0ccd12430a261b66eca36b924aeef06e17147f9f9d7d310/matplotlib-2.0.2.tar.gz (53.9MB)
    100% |████████████████████████████████| 53.9MB 681kB/s
```

Figure 3. Installing required dependencies.

The list of requirements includes the following dependencies:

falcon==1.2.0
inflect==0.2.5
audioread==2.1.5
librosa==0.5.1
matplotlib==2.0.2
numpy==1.14.0
scipy==1.0.0
tqdm==4.11.2
Unidecode==0.4.20
pyaudio==0.2.11
sounddevice==0.3.10
lws
keras

These are currently available to be used on the Peregrine cluster given that, to be able to run Tacotron on Peregrine and train our models, we requested the cluster's admin to check whether these were available and installed. However, it is always advisable to email the cluster's admin to check whether the dependencies in the requirements list are all installed on the server.

Cloning the repository

Once you are logged in, the first step is to clone the repository from GitHub to the desired folder. To do this type the following into the command line:

```
git clone https://github.com/Rayhane-mamah/Tacotron-2.git
```

This will download all the python scripts and will replicate the folder structure of the repository, ready to be used.

Bear in mind that there are multiple repositories on GitHub that offer different implementations of the Tacotron algorithm. The one we chose was the one that best suited the HPC cluster, as all dependencies required were compatible.

Download a dataset

To train our model we will use a technique that is called bootstrapping. Bootstrapping is a technique that uses resources available for well-resourced languages, in this case, English, and makes use of these to develop resources for under-resourced languages, in this case, Grunnegs. For this reason, we will have two datasets. One in English and one in Grunnegs. However, for experimental purposes, we also compiled a third corpus, of spoken Dutch.

The English dataset that we will be using is called LJ Speech. The link to download this dataset is: https://keithito.com/LJ-Speech-Dataset/. This dataset consists of 13,100 short audio files, recorded by a single speaker, reading texts from 7 different non-fiction books. There is a transcription available for each file. This amounts to up to 24 hours of spoken data.

The Grunnegs dataset that we used has been produced by this team. It comprises two thousand sentences, recorded by a single speaker, reading texts retrieved from the internet, mainly short fiction, with some non-fiction excerpts. It is about 1,5 hours long.

The Dutch dataset, also produced by this team, comprises about two thousand sentences recorded by a single speaker, reading texts retrieved from the internet, in this case, Wikipedia articles. It is about 2,5 hours long.

It is possible to use other datasets. To do this it is important to either follow the same format that is expected by the python scripts. You can either follow the expected formats and format your dataset accordingly or, if your dataset is already formatted in a particular way, you might want to revise the code and make the required adjustments, if this might be less work than reformatting your corpus.

Preprocessing

In order to start the training, it is important to preprocess the loaded text and audio files. This requires the following command:

```
python preprocess.py
```

The process of preprocessing prepares the corpus to be ready to be processed by the training process.

Training

Given that we are bootstrapping, the training process is not one step, but at least two steps. First, we will train a model for the English data, and then we will use this model as a checkpoint in order to train, together with the Grunnegs data, a Grunnegs model.

To improve the quality of the model we trained the English data all the way up to about 600,000 steps. In this way, we have allowed the model to become accurate enough. On this data we then trained the Grunneger data between 30,000 and 100,000 steps. As from 30,000 steps the synthetic voices started producing desirable results. For the Dutch data we followed the same pattern, training the Dutch data on the English checkpoint.

Training from scratch. To start the training process, now that the corpus is in the corresponding folder, is to type into the command line:

```
python train.py --model='Tacotron'--
tacotron_train_steps=500000
```

If you want to train both models (Tacotron and Wavenet):

```
python train.py --model='Tacotron-2'
```

This implementation of Tacotron generates checkpoints every 5000 steps and stores them in the "logs-Tacotron".

After training the English model. After the English model has been completed it is important to run the Groningen version. For this, we use the previously discussed bootstrapping method.

Step 1: Remove the English files present in the LJSpeech folder and load the Groningen version of the dataset in the LJspeech folder
Step 2: Move the Groningen zip files inside the folder and extract the zip document.
Step 3: Preprocess the files (python preprocess.py)
Step 4: Continue training the model using

```
python train.py --model='Tacotron'--
                tacotron_train_steps=750000
```

Training using a pre-trained model. To train a model starting from a previous checkpoint, type into the command line:

```
python train.py --model='Tacotron'--
                tacotron_train_steps=700000
```

Explored Venues. Our last development was to train a model using a variation of Tacotron that allows for the manipulation of prosody, called the General Style Token (GST). This came about through feedback that was given after Professor van Doorn's inaugural lecture, where a sample of synthesized speech was played, some of the attendants reflected and considered that the sample was played too fast. This pushed us to look into the possibilities of controlling the prosody of our models. We first looked into Speech Synthesis Markup Language (SSML). This is an industry standard when it comes to controlling prosody of the synthesized voice. It works similarly to how html works to format webpages. Tags are added to the text that is to be synthesized and these tags provide instructions on how to synthesize the speech. Unfortunately, the models produced by the Tacotron algorithm do not support SSML.

Advice. The Peregrine cluster sets limits to the amount of time jobs are allowed to run. Therefore it is important that when a job is set to run on the cluster, that the required calculations are made in order to avoid the job being cancelled before it actually gets to yield results, i.e. usable checkpoints. If you are working on your own HPC cluster or you have no restrictions you can, of course, ignore this remark.

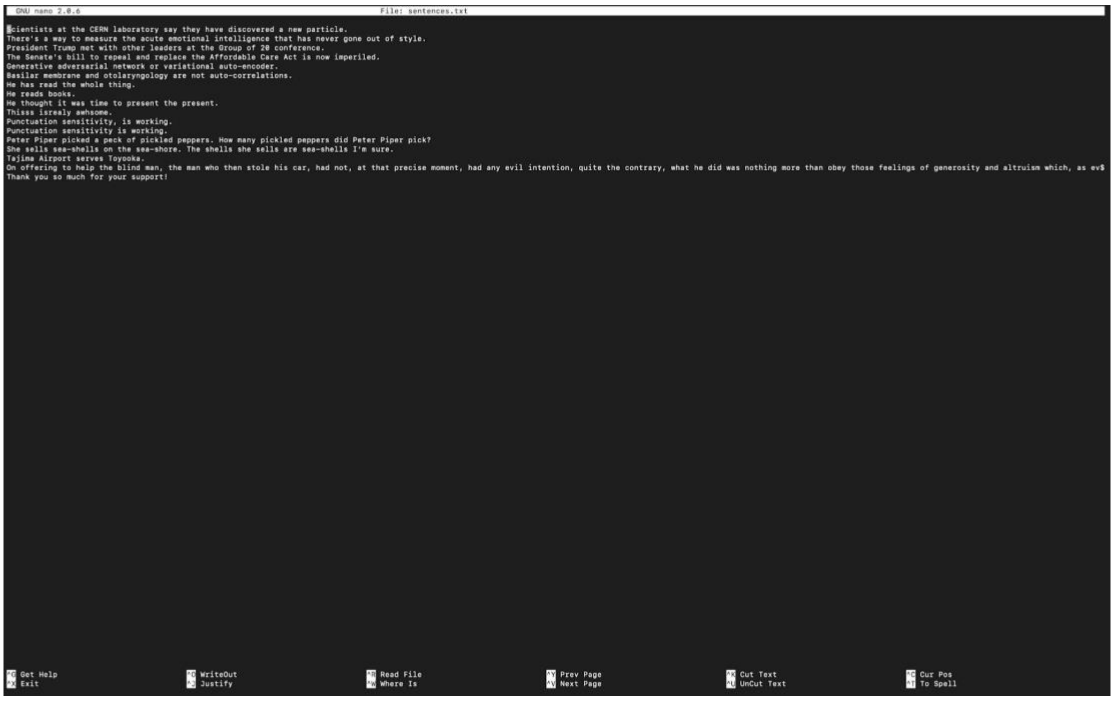
Synthesizing text

After having trained your model, you are ready to produce synthesized utterances. To do this, you first need to write your text. Be sure that the text is spelt according to the spelling you used to train the model because the model can only accurately synthesize fragments of text that follow the same spelling conventions that the texts used to train the model have used.

To input the text you are intending to synthesize you need to open a terminal window and log yourself into Peregrine. After you have successfully logged yourself in you need to go to the model's home folder (where the model is saved, easily recognized as the folder where the file synthesize.py can be found).

Here you will find a file called 'sentences.txt'. To edit the text in this file you have to use some text editor that runs in a Linux OS. There are several available. However, the most basic tool that can be used is 'nano'. So to edit the file, you just have to type into the command line:

```
nano sentences.txt
```



```
  GNU nano 2.0.6                                  File: sentences.txt

Scientists at the CERN laboratory say they have discovered a new particle.
There's a way to measure the acute emotional intelligence that has never gone out of style.
President Trump met with other leaders at the Group of 20 conference.
The Senate's bill to repeal and replace the Affordable Care Act is now imperiled.
Generative adversarial network or variational auto-encoder.
Basilar membrane and otolaryngology are not auto-correlations.
He has read the whole thing.
He reads books.
He thought it was time to present the present.
Thisss isrealy awhsome.
Punctuation sensitivity, is working.
Punctuation sensitivity is working.
Peter Piper picked a peck of pickled peppers. How many pickled peppers did Peter Piper pick?
She sells sea-shells on the sea-shore. The shells she sells are sea-shells I'm sure.
Tajima Airport serves Toyooka.
On offering to help the blind man, the man who then stole his car, had not, at that precise moment, had any evil intention, quite the contrary, what he did was nothing more than obey those feelings of generosity and altruism which, as ev$
Thank you so much for your support!
```

Figure 4. When editing sentences.txt for the first time the sample text will appear

This will open the .txt file and will enable you to edit it. Once you input the desired text, you can go on and save it by pressing 'ctrl+o' and then pressing 'ctrl+x' to exit. You can also directly press 'ctrl+x' and if there are any changes from the original it will ask you if you want to save it first, but saving from time to time with 'ctrl+o' is always advisable.

Another step that has to be taken before the text is synthesized is to load all the dependencies that are required. In the model folder, there is a file that contains a list of all dependencies that need to be loaded before you go on and synthesize the text, as, without these dependencies, the program will inevitably render an error message at one point or another. This list can be accessed by
Once this is done you are ready to type:

```
python synthesize.py 'tacotron2' 'sentences.txt'
```

into the command line. This will start the process of loading the model first, and, after a while, begin to synthesize the text. There is a progress bar that shows in percentage how far advanced the process is. The process will render graphic representations of Mel spectrograms and .wav files of the sentences that were included in the sentences.txt file, both using a linear model as well as a Mel spectrogram model. Baring a few exceptions, the Mel-based .wav files are easier to understand and sound better.

Robot

Selecting a robot

Selecting the appropriate robot for our project was a matter of considering the resources we had and looking at the available options in the market. Among the numerous options available in the market, we looked in detail at three options. There were two options from Softbank Robotics, Nao and Pepper, and a third option from Furhat Robotics, Furhat.

Both Nao and Pepper have almost the same operating system and it is possible to use scripts written in python. The design of both robots gives the user a friendly impression. While Nao is rather small and has legs, Pepper has no legs and instead moves with wheels. From previous experiences the team considered Nao to be rather unstable and prone to fall. Pepper on the other hand did not seem to have this problem.

Furhat was a very impressive robot, that is able to project a rather realistic face on its screen. The programming language that is used is Kotlin. The negative aspects of this robot are that it is only a bust, the realistic face could unnecessarily complicate our experiment, and additionally, the programming language is less popular than Python, therefore more resources would be available, when compared to Kotlin, which is relatively recently developed programming language.

Therefore, we finally decided to acquire a Pepper robot, as it was the more stable option between the two Softbank Robotics robots, and it was a better fit for our project when compared to the Furhat robot, as explained in the previous paragraph.

Implementing the synthetic voices on the robot

To get the robot to reproduce the synthetic voice rendered by the models trained following the steps in the previous chapter there are different methods that are available. One solution would be to have a model synthesizing speech live, i.e., in real time. This solution is very versatile, but it is very demanding when considering resources. A server has to be constantly available, with the model running, in order to run the computations to synthesize each new sentence requested. Another solution is to synthesize a given number of sentences that can be previously expected to be used in the context where the robot will be deployed and load these in the robot. These can then be linked to behaviors programmed into the robot using the software Choreographe.

Given the resources that the first solution requires, the chosen solution for this project was the second one. In the following two sections this solution is described in more detail.

Transferring the files

To get the Pepper robot to reproduce the sentences we synthesized (either in Dutch or in Grunnegs), we have to transfer the audio files manually from the HPC cluster to the Pepper robot. In order to do so, we first have to start up our preferred file explorer. In most cases a SSH File Transfer Protocol (SFTP) or a Secure Copy Protocol (SCP) is recommended. A popular SFTP client is FileZilla, which we will use for our explanation.

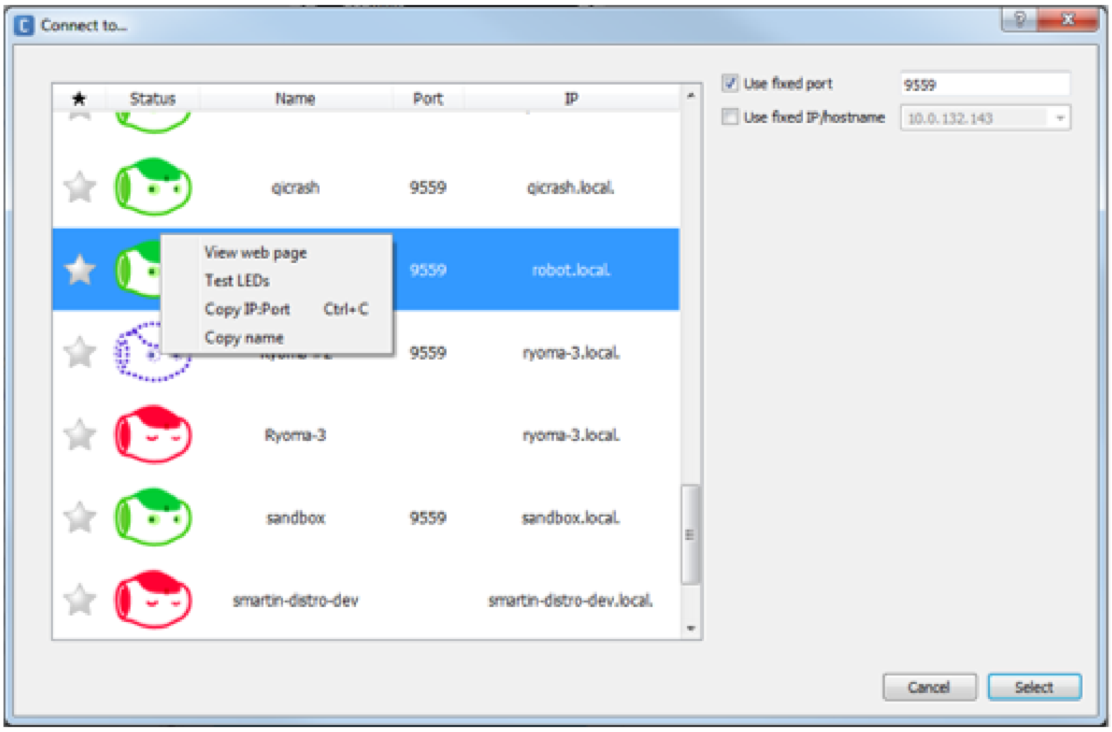
Enter the credentials into FileZilla to connect to the server. Once on the server, select the right Tacotron folder. After the first time running the model to synthesize text into speech, as discussed in the previous chapter, you will find a new folder called “tacotron_output”. Open this folder and select a new folder called “logs-eval”. In this folder select the folder “wavs”. This is the folder where you will find the Grunneger or Dutch voice output (depending on the model you have used). Transfer these files to a local folder on your computer.

Choregraphe

After the files have been transferred to your local computer it is time to start up Choregraphe. The last version of this software can be downloaded for free from the Softbank Robotics website. Be sure to have the Pepper robot on and connected to the same Local Area Network (LAN) as your local computer.

When the software package has been successfully launched, connect the Pepper robot choose the “Connection” > “Connect to” menu or Click the “Connect to” button (see Figure 5). Any of these two actions will prompt the “Connect to” panel (see Figure 6) to be displayed. Select the robot that should have appeared in the list on the left side of the panel and click on the “Select” button on the lower right corner of the panel.

Figure 5. “Connect to” button.



Connect to...

★	Status	Name	Port	IP
☆		qicrash	9559	qicrash.local.
★			9559	robot.local.
☆			9559	ryoma-3.local.
☆		Ryoma-3		ryoma-3.local.
☆		sandbox	9559	sandbox.local.
☆		smartin-distro-dev		smartin-distro-dev.local.

☑ Use fixed port 9559

☐ Use fixed IP/hostname 10.0.132.143

Figure 6. “Connect to” panel.

Once the robot is connected to Choreographe, we can start to upload the audio files to the robot from your local computer. We first have to add a new action. This action can be found under “Multimedia” > “Sound” menu and then select “Play Sound”. Drag this action to the functionality screen and open the settings of the box. After opening the settings select one of the preferred audio files that you have transferred in the previous step to your computer. Then press “OK” and play. The voice should now be pronounced on the Pepper robot if all steps have been finished correctly.