Voice Conversion Semester Project

EURECOM
Sophia Antipolis

Valentin Nelu Ifrim
Marco Vinai
Elaa Mansour

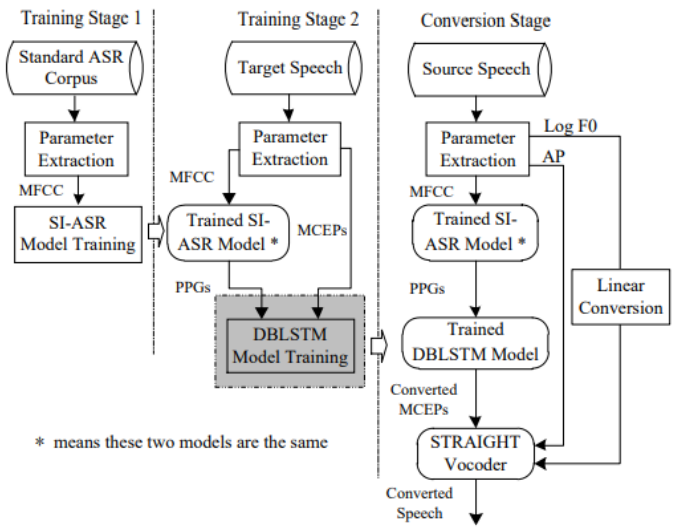
Intro about VCC2020

Voice conversion is a NLP challenge that has gained more and more consideration in the last years due to the promising results achieved in the deep learning field.
Voice Conversion is the means of converting one’s voice to sound like that of another without changing the linguistic content: a technique to modify speech waveform to convert non-linguistic information while preserving the linguistic information.
In this project our aim is to start from a sentence spoken by a source speaker and make our system reproduce it as it was told by a target speaker.

In order to monitor the improvements in technologies and methods for this task, a challenge is created every two years: the VCC (Voice Conversion Challenge), that collects the best models ranking them according tothe following metrics: **naturalness of speech** and **target speaker similarity.**
Starting from the results achieved in the VCC2020, we considered the different possible approaches and tried to replicate one of them (T11).

T11

Among the various systems in the last VCC edition, one in particular outperformed the others, so we decided to take it as reference and try to reproduce it.
Another advantage of this model is that it does not require parallel data to be trained, therefore we are not bound to any specific dataset.
This system relied on a 3-step model for the feature transformation part and a WAVEnet vocoder to reproduce the audio. Since we have multiple target users and creating and training a specific vocoder for each of them would be time consuming, we decided to focus on the first model, using a classical signal-processing vocoder to reproduce the audio.
The source model, with the various stages can be represented by the following graph:



We decided to replicate this structure, changing only the final part: we used the pyworld vocoder, instead of the STRAIGHT one, so we were able to keep everything on python.

In the following sections we will get deeper on the details of each section of our model, explaining the functions we used and how to make them work, so that everyone is able to replicate our work, using our code at https://github.com/IAmNelu/DeepVoiceSP.

MFCC2PPG

For this first model we took as reference the thesis Timo_van_Niedek___4326164___Phonetic_Classification_in_TensorFlow.pdf (ru.nl), that exploited the TIMIT dataset. The goal of this first step is to extract from the input audio the MFCCs and to convert them to the corresponding PPGs.

Parameters

The configuration file for this first part (*mfcc2ppg_config.json*) contains the major parameters that can be tuned for this first part. Those parameters are the following:

- **TRAIN_PATH**: Path to train dataset
- **TEST_PATH**: Path to test dataset
- **FOLDING_DICT**: Path to phoneme reducer dictionary (i.e *foldings61-39.json,* present in */config_files*)
- **CONFIG_RUN**: dictionary of configuration for the run: all *boolean*
    - **Process_data**: if True, extract MFCCs and phonemes, performing all the preprocessing steps, else load data from pickles called “train_data.pickle” and “test_data.pickle”, avoiding all the preprocess part.
    - **Process_phonemes:** if True, process the .PHN files to reduce the number of target phonemes we want to recognize, saving two dictionaries; else just load them
    - ***[LEGACY] propTRAIN***: if True perform training (used mainly for debugging)
    - ***[LEGACY] propLOAD:*** if True the preprocessing will not be computed but loaded from pickle which path is in ***Model_to_load***
    - **propRMSilencePostLoad:** if True, remove starting and ending silence from the inputs
- ***[LEGACY] Model_to_load***: only used if ***propLOAD*** is True and indicates the path for the data already preprocessed.
- **MFCC_DATA:** Dictionary of configuration for mfcc extraction
  Those values should be consistent through the various config files **[IMPORTANT!]**
    - **Sampling_frequency:** sampling frequency value to be used in librosa functions.
    - **Order_mfcc:** Number of mfccs to extract per frame
    - **N_fft:** Length of the FFT window in librosa functions. Since win_length is not specified, this will be default value also for that parameter
    - **Hop_length:** Number of frames to jump when extracting MFCCs. Set to Sampling_frequency/100, in order to get values for every 10ms.
- **NETWORK_PARAM:** Dictionary of configuration for the MFCC2PPGs network
    - **Batch_size:** batch size for the input (number of sentences to be classified in parallel)
    - **Epochs:** Number of epochs for train
    - **Dropout:** Dropout probability
    - **Hidden_units:** Number of hidden units
    - **Order_mfcc:** Number of MFCCs per frame
      (SAME as MFCC_DATA[“**Order_mfcc**”])
    - **Lr:** Learning rate

    - ***[LEGACY]* Sq:** for the first implementation of the network indicates the sequence length as the number of frames of each utterance processed at time
    - **Lr_decay:** parameter to set lr decay speed also referred as decay rate for the implemented exponential decay of the lr
    - **Decay_steps:** number of steps before a full learning rate decay
    - **Num_phoneme_classes:** Number of target labels (Must be consistent with FOLDING_DICT used)
    - **Optim:** optimizer used
    - ***[LEGACY] Preload*:** if True loads the weight of the network from a previous checkpoint
    - ***[FUTURE] Validation_stop*:** if True implements early stopping in the training
    - ***[FUTURE] Validation_stop_value*:** if ***Validation_stop*** is True indicates which value will the training should stop
    - ***[LEGACY] Preload_path*:** if ***Preload*** is True indicates the path from where to load the weights of the network
    - **Checkpoint_path:** path used to save model of each epoch
    - **Log_file:** path to file where output log
    - **Final_checkpoint:** path used to save model of last epoch
    - **Best_checkpoint:** path used to save best model
    - **VERBOSE:** Boolean for more info during test-runs, printing statistics about validation accuracies. It is used mostly for debugging, the same infos can be retrieved from the Tensorboard Data.
- **PADDING_SIL:** number of silence frames left at beginning and end of sentence.

In the configuration files some features are not currently used. In the list above they are marked in italic with the following tags:

- ***[LEGACY]***: properties used for previous implementations that we decided to leave if wanted to recreate older tests and scenarios
- ***[FUTURE]***: properties not yet implemented that could be used to improve model performance

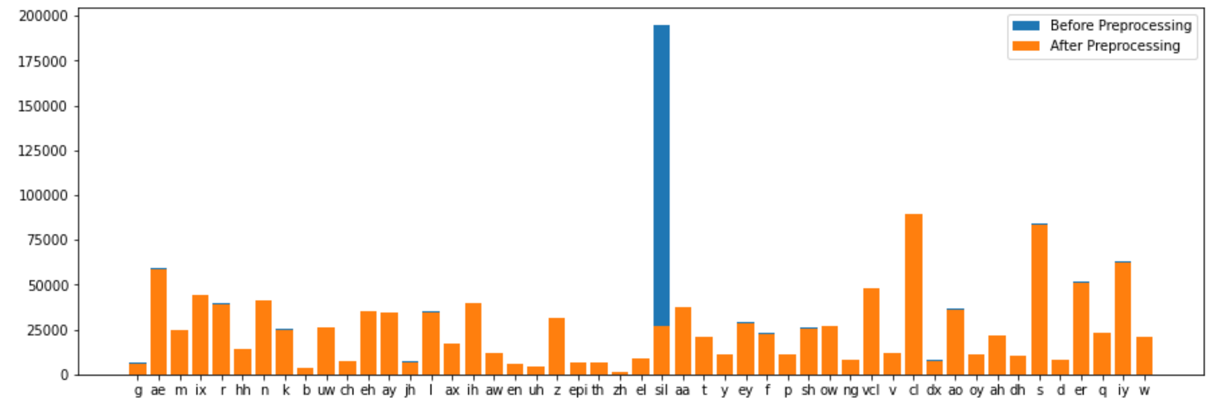
Dataset

Since this is a supervised learning problem, we need a dataset with the notation of the utterances given the sentence, so we decided to use the TIMIT dataset. This dataset is already split in train and test and for each sentence has the literal and utterance transcription. The utterance transcription, stored in ".PHN" files, indicates each phoneme with the frame indices of its duration.

Preprocessing

In order to be coherent with the dataset used, an analysis was made on how the phonemes were computed in the TIMIT dataset. Following this analysis, it was decided to extract the MFCCs features using librosa with a sampling rate of 16 kHz, as in the reference, and to set a hop-size of 160, in order to get a MFCC vector for each 10ms.

The MFCCs extracted are then normalized by MFCC order for each sentence, so that each input phrase has 0 mean and 1 standard deviation for each MFCC level.
Those steps are done exploiting the *compute_mfcc* function in the *data_preparation* file.
After this analysis, a new function [*pair_data*] sets the corresponding label for each frame analized: it takes it from the corresponding ".PHN" file and parses the sequence number on a 10ms basis, creating a one-hot vector with the 1 value on the correct phone. In particular in this step two additional helping functions are used: [*match_data*] is an utility function created to deal with dictionary entries that we are using in the pipeline and [*read_phn*] makes the one hot encoding of the phonemes.
When the length of the feature extracted and the reference mismatched, a reduction was performed, setting the final sequence length to the shortest one.
The ".PHN" files are first preprocessed with the function [*substitute_phonemes*] in order to reduce the number of phones possible, from 61 to 49, substituting equivalent phones with a single one (ex: h# -> sil), then a mapping with all the possible phones was made using two dictionaries (*phonemes_dict* and *phonemes_dict_inv*), to have a common reference in the one-hot labels.
All files are processed in this way and the extracted data is saved in ".pickle" files, to have them ready to use.
Once extracted, the MFCCs are matched with the corresponding phoneme labels.
The high number of silence frames brought a high unbalance in the dataset: we decided to remove most frames of silence at the end and at the beginning of an utterance leaving just a few of them. The effect of the function [*remove_silence*] can be seen in the histogram below that shows the cardinality of each phoneme across the whole TIMIT training set.

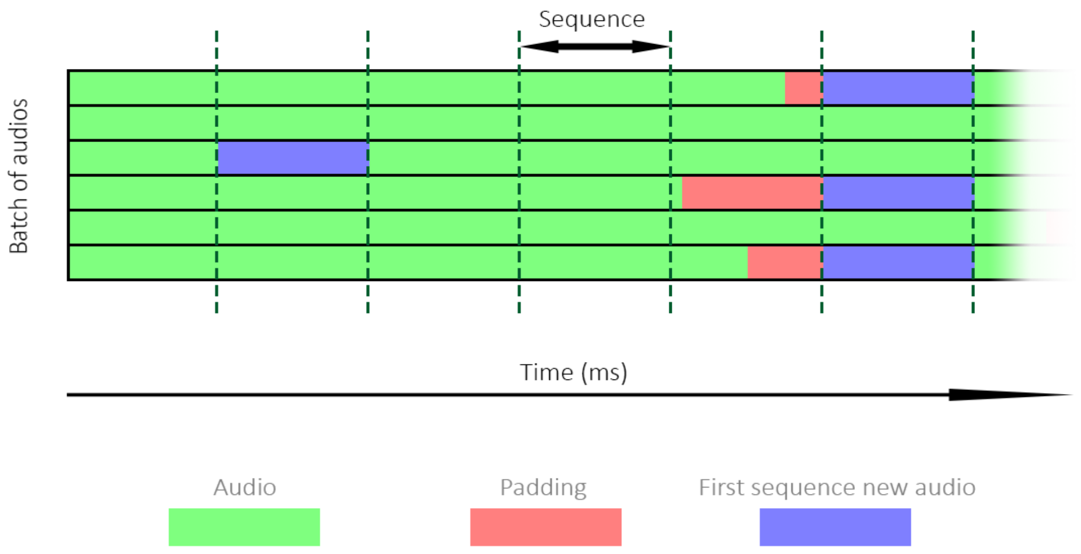


It is immediately noticeable the disproportion of the "sil" phoneme with respect to others: this was in part also caused by the reduction of phoneme we performed before: infact "h#" that indicates the start of a sequence was also mapped to silence.

Model

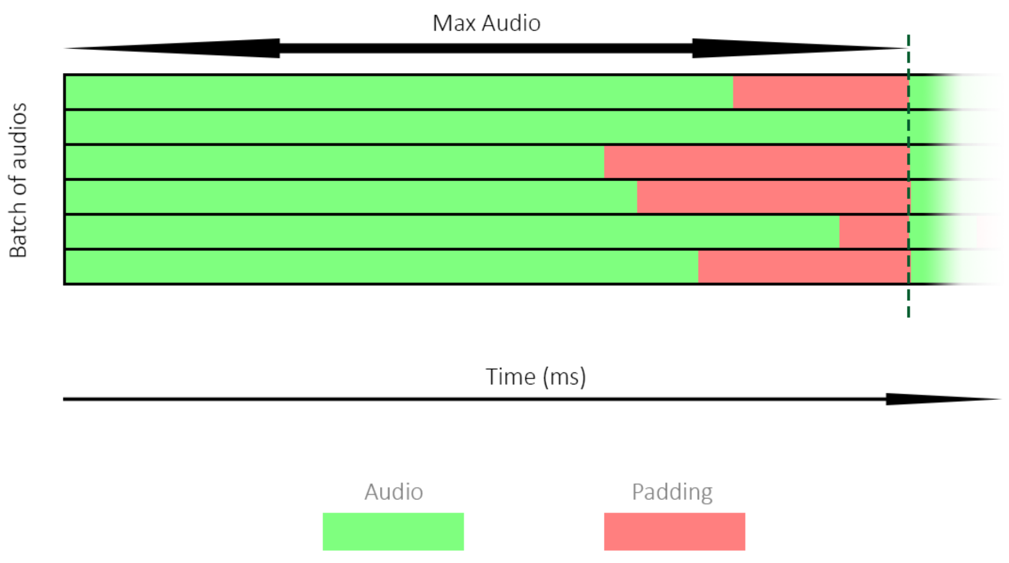
The first approach we tested and then discarded was similar to the one explained in the thesis referenced above. It consists of dealing with a mini batch of audio streams at the time. This converter network would take in input a tensor of size *[batch-size x sequence length x order mfcc].* The sequence length is a parameter that can be set in the configuration file and it indicates the number of frames to be processed at the time. By **frame** we referred to an array of MFCCs (shape *[1 x order mfcc]*) that corresponds to a time window of 10ms. The main advantage of this approach is the padding size. In fact the utterances do not have the

same length, but to speed up the process we need to evaluate them in batches and each batch must have the same size. Since our training batch as second dimension has *sequence length*, in the worst case scenario we would have a misalignment of *sequence length -1* frames. Unfortunately there is a huge drawback with this approach.
In the phoneme prediction domain, the next and previous phoneme has a huge impact on the current one, therefore the classifier network (referred to also as converter) has to take into account the context of each phoneme. For this task the Deep Bidirectional LongShort Term Memory (DB-LSTM in short) network is a good fit: the problem is that this architecture stores an internal value that would represent the current state depending on the context. We want to reset the state for each new utterance. The default tensorflow implementation does not allow resetting the state only for only some selected streams in the batch size. In the first implementation, based on the Github Tensorflow code we coded a function to reset the state when needed.



The picture above shows an example of this first intuition. As it is possible to see, we need to pad the last sequence of an audio (we repeat the last phoneme, “sil”) in order to have all utterances with a size (in terms of number of frames) multiple of the sequence length. This approach had another underlying problem. As it is possible to see for the third stream of audio in the second sequence, even if there is no need for padding, a new utterance is processed and only the third inner state of the DBLSTM has to be reset. Apart from the custom function we implemented, we also need a custom class Dataset that would give us information also on the duration of each audio.
A fundamental hyperparameter to be tuned would be ***sequence length*** since it directly controls the amount of padding added into our dataset.
With this approach we reached satisfactory results (around 69% of accuracy on the test set) following the guidelines of the aforementioned thesis.
This approach though was overly complex and complicated requiring a special dataset class and more special functions, so we decided to try another approach.

We took inspiration from an example of deep speech recognition found on MATLAB documentation. The main difference with this approach is that the new network will have as input a batch of full utterances at the time.



As it is possible to see from the picture above there is a fundamental problem with this second approach: the number of padding frames increases depending on the randomness in shuffle of the data. To limit this number, the first idea would be to sort the audios depending on their length. The issue doing that is that we lose randomness which could affect the learning process. In the end we opted for a middle ground approach. We first sorted all the data and audios by length, then depending on the batch size, we would pad them accordingly: for example if we have 7 audios, with the shortest of 54 frames and the highest two of 65 and 123 frames, with a batch size of 7 we would pad all to utterances to 123 frames, making the 1 first audio from 54 to 123, but with at least 69 frames of silence at the end; choosing instead a batch size of 6, we would pad the first 6 audios to 65 frames and the last audio would be padded according to the highest in his batch.

The final model is therefore a DBLSTM with 4 layers each with 512 **hidden units**: after each layer was used a Dropout Layer with **probability** of 0.6 to reduce overfitting. At the end there is a Dense layer with softmax activation function.

Fine Tuning / training

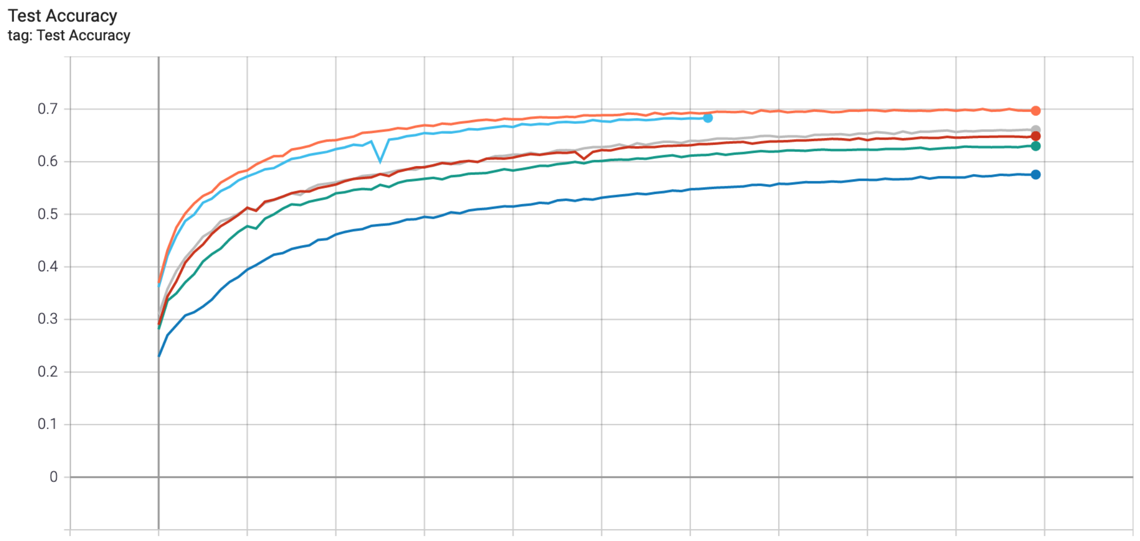
For debugging and logging purposes we used Tensorboard Library, which allowed us to see in real time the performance of our model.
In both approaches explained before the objective function that we wanted to minimize is a Categorical Cross entropy, used usually for classification. In order to keep the idea of phonetic posteriorgrams we did not one-hot encode our results but we kept them as output of the sigmoid function.
Focusing on the last model we focused our tuning mostly on the **Learning Rate** (with decay) and **batch size** (for the reasons explained in the previous paragraph). The formula we used for the learning rate at epoch ***i*** is the following:

$Lr_i = 0.93 Lr_0^{i/10}$ where 0.93 is **NETWORK_PARAM["lr_decay"]** and 10 is **NETWORK_PARAM["decay_steps"],** from the config file.
We used that formula for a continuous decay, but this can be easily implemented for a step decay, setting the division on the exponent to an integer division.

The main result can be seen in the table below.

Optimizer	Starting LR	Batch Size	Epochs	Test Accuracy
RMS PROP	2.00E-05	12	100	62.97%
Adam	2.00E-05	20	100	57.55%
Adam	2.00E-05	12	100	64.86%
Adam	1.00E-05	6	100	66.01%
Adam	2.00E-05	6	62	68.33%
Adam	2.50E-05	6	100	70.02%

The plots below instead depicts the test accuracy of the models shown in the above table evolving during the training.



How to reproduce

In order to train this first model reproducing our results it is enough to download the TIMIT dataset, clone our repository and run the *mfcc2ppg.py* script with *mfcc2ppg_config.json* as argument. This config file is already set with our latest parameters and takes around 13 hours to train for 100 epochs (on a GTX 1050), reaching an accuracy of more than 70%.

In order to run, some libraries are required, such as ***librosa, tensorflow, numpy*** and ***pandas.***

PPG2MCEP

This model is the second phase in our pipeline: it takes as input the PPGs produced by the previous system and gives as output the corresponding MCEPs values for the selected target speaker.
In this regression problem we do not need any specific dataset, so we can use as reference target whichever data we want, as long as it is specific to the same speaker.
We had to create a specific system for each one of the targets in the *VC_dev_target* list, that will result in multiple training sessions.

Parameters

The configuration file for this first part (*ppg2mcep_config.json*) contains the major parameters that can be tuned for this first part.
[IMPORTANT] Those parameters must be consistent with the ones of the previous system
Those parameters are the following:

- **PATH_TO_DATA:** path to dataset folder folder
- **PATH_TO_LIST:** lists of audios to be loaded
- **SPEAKER:** speaker code used for the vc dataset
- **SCALER_PATH:** path to folder where to store the scaler data or the given target
- **NETWORK_PARAMS:** Dictionary of configuration for the PPG2MCEP network
    - **Batch_size:** batch size for the input (number of sentences to be classified in parallel)
    - **Epochs:** Number of epochs for train
    - **Dropout:** Dropout probability
    - **Hidden_units:** Number of hidden units
    - **Dim_ppgs:** input size (number of phoneme classes, MUST be consistent with previous system, therefore same of MFCC2PPG["Dim_ppgs"])
    - **Dim_mceps:** number of MCEPs predicted for each frame (MUST be consistent with MCEP["Order_mcep"])
    - **Lr:** Learning rate
    - **Lr_decay:** parameter to set lr decay speed
    - **Decay_steps:** number of steps before a full learning rate decay
    - **Optim:** optimizer used
    - ***[FUTURE] Validation_stop*:** if True implements early stopping in the training
    - ***[FUTURE] Validation_stop_value*:** if ***Validation_stop*** is True indicates which value will the training should stop
    - **Check_points:** path used to save model of each epoch
    - **Final_checkpoint:** path used to save model of last epoch
    - **Best_checkpoint:** path used to save best model
    - **Log_file:** path to file where output log
    - **VERBOSE:** Boolean for more info during test-runs, printing statistics about validation accuracies. It is used mostly for debugging, the same infos can be retrieved from the Tensorboard Data.
- **MCEP** Dictionary of configuration for mfcc-mceps extraction. MUST be consistent with first part configuration.
    - **Sampling_frequency:** sampling frequency value to be used in librosa

    functions.
    - **Hop_length:** Number of frames to jump when extracting MFCCs. Set to Sampling_frequency/100, in order to get values for every 10ms.
    - **N_fft:** Length of the FFT window in librosa functions. Since win_length is not specified, this will be default value also for that parameter
    - **Order_mcep:** Number of mceps to extract per frame
    - **Order_mfcc:** Number of mfccs to extract per frame
- **MFCC2PPG:** Dictionary of configuration to load and set up the first model pre-trained, paying attention to be consistent with the training done at the previous step
    - **Batch_size:** batch size for the input (number of sentences to be processed in parallel)
    - ***[LEGACY] Sequence_length*:** indicates the number of frames processed at time by the first MFCC2PPG converter
    - **N_mfcc:** Number of MFCCs per frame
    - **Hidden_units:** Number of hidden units
    - **Dim_ppgs:** Number of target labels
    - **Path:** path to pre-trained weight to load the model

Dataset

Since we do not have constraints, as the PPG extraction is done using the first system, we can use the dataset we want, as long as it consists of sentences said by the same target user: this allows us to target whoever we want.
In this case we used the ***VCTK*** dataset that had on average for each speaker 700 seconds of audio.

Preprocessing

In order to extract the data for the target speaker, all the audios are processed by extracting the MFCCs and MCEPs, with the latter that will be used as target for our regression. The extracted data is then normalized and the MCEPs values of normalization, which are computed considering the whole dataset, are stored to scale up the results.
In this process we are relying on two different libraries to extract those features: we use ***librosa*** to extract the MFCCs in the same way as before, while regarding the MCEPs computation we use ***pySPTK***.
Since we are using a different library that does not automatically implement the frame embedding, in order to get the MCEPs we have to create the frames manually, adding some zero padding in order to achieve the expected results. This process is done in the function *load_mfcc_mceps_VCTK,* while extracting those features.
The PPGs are then predicted using the pretrained model of the first phase, using the *get_ppgs_mceps* function and they will be the features to feed to our network.

Model

The problem we are facing is a regression one since we are training to get real values (of MCEPs) given a set of features that are the posterior grams predicted by the first stage of

the whole pipeline. Once again we use a DB-LSTM since the context is important. We followed an approach similar to the one used for the MFCC2PPG with two main differences.
The padding was done with zeros in order to have all audios in the batch of the same length and we do not use a sigmoid as the last activation function since we are performing a regression.

Fine Tuning / training

To train the model we focused on minimizing the MeanSqauredError. Since this is an intermediary task, to evaluate our model we split the data in Train and Test and as reference metric we used the MSE on the test split.
Again one of the most important hyperparameters to be tuned is the learning rate that we used keeping the same optimizer (**Adam**).
In this scenario we tried to focus also on the importance of the **DropOut probability**.
In an ideal scenario we should have tuned the parameters for each speaker but we focused on the speaker **p260** and used the same configuration for all the training sessions. We could not perform more optimization due to time constraints.
The table below shows some of the most important parameter tuning mentioned.

Speaker	Learning Rate	DropOut	Test MSE
p260	5.00E-05	0.5	0.6628
p260	7.00E-05	0.5	0.6618
p260	7.00E-05	0.7	0.78
p260	3.50E-05	0.7	0.78
p260	3.50E-05	0.4	0.6368

How to reproduce

In order to train this first model reproducing our results it is enough to have already trained the first system and have a target dataset quite big, as the generalization will get better with more data and we can exploit only the audio from a single speaker.
After cloning our repository and training the first system, run the *ppg2mcep.py* script with *ppg2mcep_config.json* as argument. This config file is already set with our latest parameters and takes around 40 minutes to train for 100 epochs with a target speaker dataset of around 11 minutes of audio. The resulting loss is highly dependent on the speaker, so it is therefore hard to use a general metric to evaluate the model.

In order to run, two libraries are further required with respect to the previous ones: ***pysptk,*** which allows us to extract the mceps and ***sklearn*** to split the dataset in train and test.

VOCODER

The vocoder is the third phase in the baseline that we are working on. It is responsible for the generation of target speech waveforms from converted features.

A traditional parametric vocoder is used in our implementation which is pyworld vocoder. It takes as input: **F0, aperiodicity, spectral envelope** and outputs converted waveform. Features used are :

- **F0** of source speaker: Fundamental frequency: lowest frequency of a periodic waveform
- **Aperiodicity (ap)**: speech is decomposed into two components: a periodic or quasi-periodic component which takes into account the quasi periodic segments of speech produced by regular vibrations of the vocal cords and an aperiodic component (or noise component). The aperiodic component corresponds to two main physical situations in speech production: additive noises and structural noises
- **Spectral envelope**: smooth spectrum of tones, derived from the MCEPs processed by the previously explained system

Pyworld vocoder is very similar to the straight vocoder since they have the same inputs and operate in the same manner but we worked with ***Pyworld*** because it has a python wrapper which is not the case with straight which has only a matlab implementation and also pyworld vocoder has a richer library than straight vocoder . This choice helped us keep all the work in python.

Parameters

Functions used from the **pyworld library** :

- **dio** : used to extract F0 contour of the input sound
  - *x* : Input signal
  - *fs* : Sampling frequency (we used 16000)
  - It **outputs** a vector containing F0 values and a vector t containing temporal positions of F0 values
- **stonemask :** used to refine the estimated contour F0 by dio function . stone mask help to improve the robustness to noise
  - *x* : Input signal
  - *F0* : F0 contour
  - *fs* : Sampling frequency
  - *time_axis* : The vector t outputted by *dio* function containing temporal positions of F0 values
  - It **outputs** vector containing F0 values refined
- **d4c:** used to calculate the aperiodicity of a signal
  - *x* : Input signal
  - *alpha*:float: All pass constant (we used the default value equal to 0.35)
  - *fs* : Sampling frequency
  - *time_axis* : The vector t outputted by dio function containing temporal
  - It outputs a two-dimensional array
- **synthesize :** synthesize the voice based on f0, spectral envelope and aperiodicity

- *F0*: F0 value
- *sp*: spectral envelope resulting from the *mc2sp* function
- *ap*: aperiodicity estimated using *d4c* function
- *fs*: sampling frequency

Functions used from **pysptk library** :

- **mc2sp:** Convert mel-cepstral to spectral envelope
    - *mc*: array of Mel-cepstral coefficients
    - *alpha*: float:All pass constant (we used the default value equal to 0.35)
    - *fftlen*:int: FFT length which is the length of the FFT window (we used 512)
    - It outputs a two-dimensional array containing the spectral envelope

Processing

First we have loaded the source speech signal
Then the source audio waveform has to be manipulated by extracting its features :

- F0 is extracted using the function *dio* of pyworld vocoder: sampling frequency=16000 and a frame =10ms which are the same parameters used in the training step and also we got a vector containing F0 Values
- F0 extracted is refined using the stone mask
- Aperiodicty is extracted using the function D4C using previously specified parameters

For the spectral envelope, we took the converted MCEP generated from the previous phase and converted them to a spectral envelope using the function **mc2sp** but we had a size incompatibility problem between the spectral envelope, the aperiodicity and the F0 vector despite using the same configuration in all phases (same frame size, same fftlen, same sampling frequency).
To overcome this problem, we decided to double the size of the MCEPs array by repeating it two times. This helped us achieve size compatibility .
Then we passed all those features as input to the function synthesize which outputs the generated sound from the vocoder

Results

When listening to the resultant sound from the vocoder synthesization , the words spoken in the speech are very noisy and the voice is somehow “robotic” and “buzzy” . One reason could be because the vocoder is using signal-processing mechanisms which cause the loss of phase information and temporal details, which results in marked speech quality degradation

Putting all together:

In order to run the trained models on a selected audio, and convert it to the target voice, another python file was created for the ***VCTK*** dataset: *vc_convert_audio.py*
This file has to be used with the *vc_config_converter.json* configuration file and it is able to convert all the requested audios for a specific target speaker. The parameters are listed in the next section, while the proper operations are just a repetition of the ones explained in the previous sections.

Parameters

- **input_sentence:** path to list of sentences audio codes to be converted
- **Target_scaler:** path to folder of scalers (where scaler of target user will be found)
- **SPEAKER:** code of target speaker (VC dataset used)
- **PATH_TO_DATA:** path to input audio folder
- **RESULT_FOLDER:** Folder to store the converted .WAV audio
- **PPG2MCEP:** Dictionary of configuration to load and set up the second model pre-trained, paying attention to be consistent with the training done
    - **Batch_size:** batch size for the input
    - **Hidden_units:** Number of hidden units
    - **Dim_ppgs:** input size (number of phoneme classes, MUST be consistent with previous system, therefore same of MFCC2PPG[“Dim_ppgs”])
    - **Dim_mceps:** number of MCEPs predicted for each frame (MUST be consistent with MCEP[“Order_mcep”])
    - **Path:** path to pre-trained weight to load the model
- **MCEP** Dictionary of configuration for mfcc-mceps extraction. MUST be consistent with training configuration.
    - **Sampling_frequency:** sampling frequency value to be used in librosa functions.
    - **Hop_length:** Number of frames to jump when extracting MFCCs. Set to Sampling_frequency/100, in order to get values for every 10ms.
    - **N_fft:** Length of the FFT window in librosa functions. Since win_length is not specified, this will be default value also for that parameter
    - **Order_mcep:** Number of mceps to extract per frame
    - **Order_mfcc:** Number of mfccs to extract per frame
- **MFCC2PPG:** Dictionary of configuration to load and set up the first model pre-trained, paying attention to be consistent with the training done
    - **Batch_size:** batch size for the input (number of sentences to be processed in parallel)
    - ***[LEGACY] Sequence_length*:** indicates the number of frames processed at time by the first MFCC2PPG converter
    - **N_mfcc:** Number of MFCCs per frame
    - **Hidden_units:** Number of hidden units
    - **Dim_ppgs:** Number of target labels
    - **Path:** path to pre-trained weight to load the model

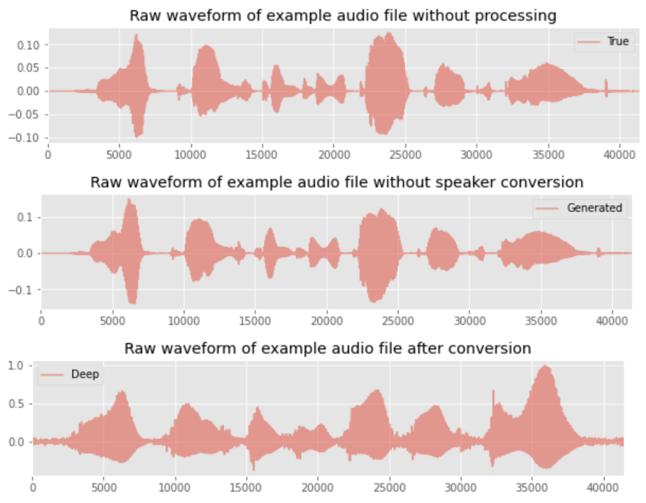
2 Demo Notebooks

Audio2MFCC2PPG.ipynb

To show the results of the first stage of the pipeline we created a python notebook that clones the repository and takes advantage of the ***pydub*** library and some javascript to register some seconds of audio from the microphone. Then the same model described before is created and the weights are loaded (due to size constraints the weights are not available on github, but it is possible to obtain them using the configuration explained in the previous paragraph). Eventually the audio is first converted into MFCCs that are fed to the network. The results will be then associated with the corresponding phonemes and compressed. The compression operation is done for visualization purposes: it would be useless to display the same phoneme if it is repeated and it is enough to show the end and the beginning of suh phoneme.
It is important to keep in mind that this visualization is not precise and it does not give any intuition on the probability distribution of the predicted phoneme.

Audio_Processing_Demo.ipynb

Following the final code of the resulting pipeline, an example notebook was created: it clones our repository and, taking the weights of pretrained networks from the drive (weights for both MFCC2PPGs and PPGS2MCEPS, for a specific target), is able to convert any .WAV file uploaded to the target speaker voice. As already mentioned, due to size constraints the weights are not available on github, but it is possible to obtain them using the configuration explained in the previous paragraph.
Furthermore, it also shows some plots on the generated waveforms, together with the different result quality, using our transformed output and the original one.
Overall performances are noisy and it is very difficult to distinguish the single words, but it can be noticed how different audio sources transformed to the same target have a similar pitch tone, that, although noisy, can be resembled as to a common speaker. At the same time, changing the target speaker leads to a different conversion result, so that even humans are able to distinguish the results, in terms of same/different target speaker.
Analyzing the waveform generated, we can see how after conversion the general shape can be still identified, but in a more noisy way, that is the reason for the low quality results.



Conclusion and Further work

Voice conversion is one of the hardest tasks in natural language processing, as it involves features peculiar to each individual. Our attempt confirms this difficulty, but we have to highlight that we are not using parallel data to train our systems, which usually are costly to obtain. Even though the generated audio is very noisy and hard to understand, it has to be remarked that when we produce audios for two different targets it is possible to recognize the different target by listening to them. Another good result is the fact that listening to audio generated from different source speakers to the same target can be recognized as belonging to the same person.

In order to improve the final audio result, the next step would be to change the implementation of the vocoder, moving from our classical “signal-processing based” vocoder to a neural-network one, which in the VC challenge showed astonishing performance.
Overall, the results achieved are not fully satisfying but they are for sure a good step in the right direction; hopefully this could be enough to fool an automatic voice recognition system and this could be another interesting direction to explore.