Audio files are present in various file formats such as .mp3, .wma, .wav etc. Python offers some great libraries for audio processing. Librosa is one such library which offers audio processing. Scipy and torchaudio are some other libraries that offer this feature but they are not as famous as librosa.

The file can be loaded in Librosa in the following way:

|  |
| --- |
| import librosa |
|  |  |
|  | # Load the audio file |
|  | AUDIO\_FILE = './audio.wav' |
|  | samples, sample\_rate = librosa.load(AUDIO\_FILE, sr=None) |

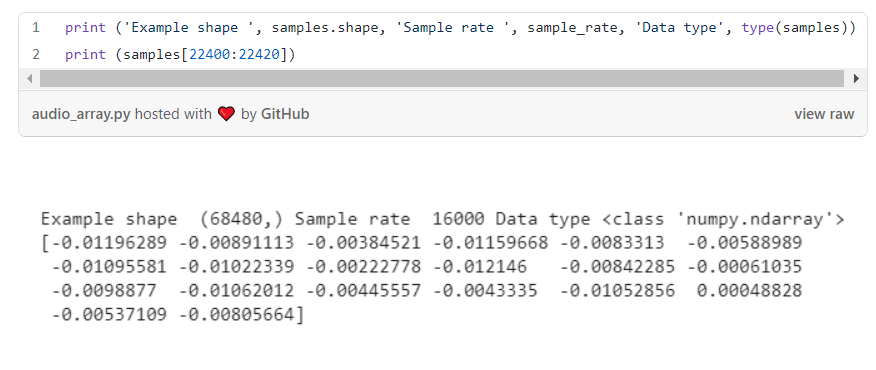
The following code can be used to visualize the voice:

|  |
| --- |
| import librosa.display |
|  | import matplotlib.pyplot as plt |
|  |  |
|  | # x-axis has been converted to time using our sample rate. |
|  | # matplotlib plt.plot(y), would output the same figure, but with sample |
|  | # number on the x-axis instead of seconds |
|  | plt.figure(figsize=(14, 5)) |
|  | librosa.display.waveplot(samples, sr=sample\_rate) |

In a jupyter notebook the following code can be used to play the song:

|  |
| --- |
| from IPython.display import Audio |
|  | Audio(AUDIO\_FILE) |

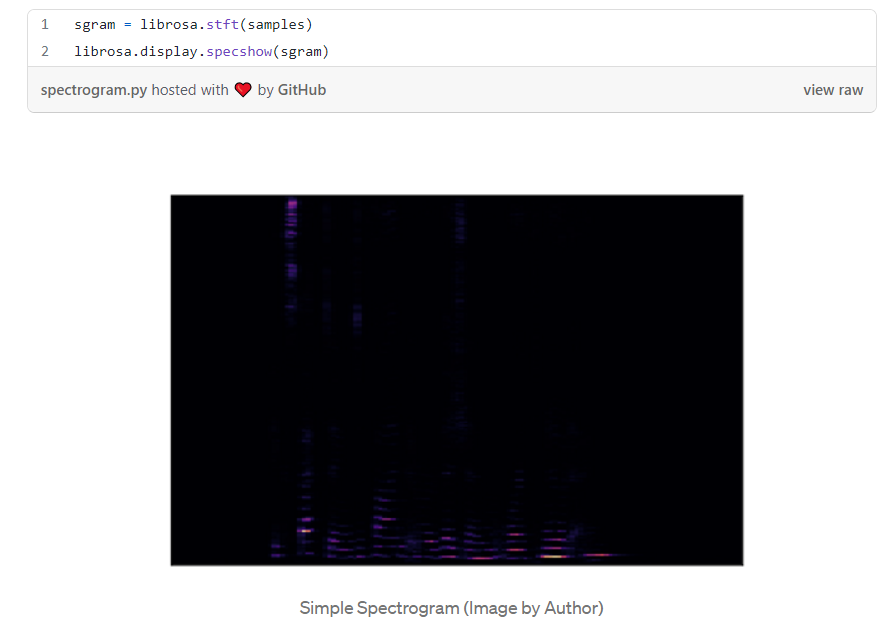
Since audio is continuous variations in the air pressure we sample a certain number of points from the audio file which are taken at fixed intervals. The processed audio file is then converted into an array which contains the amplitude of signals at various points at certain time intervals.



Bit-depth tells us the possible values that the amolitude in the sample can take. For example, a bit-depth of 16 means that the amplitude number can be between 0 and 65535 (2 ¹⁶ - 1).

**Spectrograms**

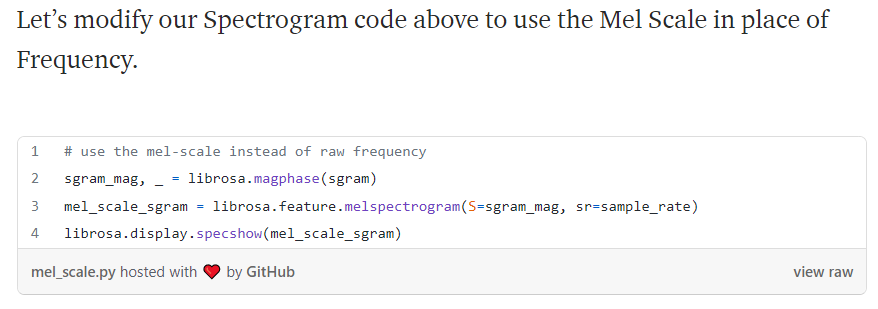
Spectrograms are generated from sound signals using Fourier Transforms. A Fourier Transform decomposes the signal into its constituent frequencies and displays the amplitude of each frequency present in the signal. A Spectrogram chops up the duration of the sound signal into smaller time segments and then applies the Fourier Transform to each segment, to determine the frequencies contained in that segment. It then combines the Fourier Transforms for all those segments into a single plot. It plots Frequency (y-axis) vs Time (x-axis) and uses different colours to indicate the Amplitude of each frequency.



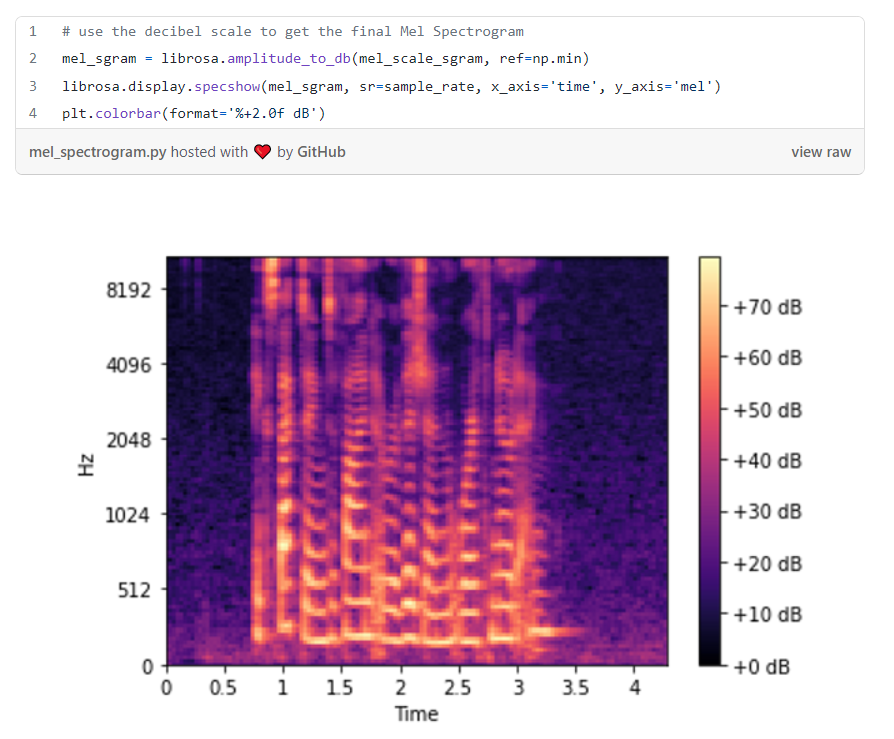
Humans perceive frequencies on a logarithmic scale rather than a linear scale. The Mel Scale was developed to take this into account by conducting experiments with a large number of listeners. It is a scale of pitches, such that each unit is judged by listeners to be equal in pitch distance from the next. Human perception of amplitude is that of the loudness and that too we perceive on a logarithmic scale. For this we use the Decibel Scale.

**Mel Spectrogram**

A Mel Spectrogram uses the above described properties instead of the basic spectrogram properties to describe the data. Instead of using frequencies it uses Mel Scale and instead of using the colours as an indicator of loudness it uses Decibel scale.



So let’s modify it to use the Decibel Scale instead of Amplitude.



A mel spectrogram

A spectrogram is prepared by using the techniques of Fourier Transform. We could have used DFT(discrete fourier transform) but it is very expensive. Therefore we use FFT (fast fourier transform) but it cannot take in account how the frequencies change with time. Therefore we finally use STFT (short time fourier transform) which first samples the data at a fixed rate and then applies FFT individually to all the samples and the stacks them up to form the spectrogram. Sometimes we need to use the techniques of MFCC (mel frequency cepstral coefficients) which works better with **human voices**.

**Data augmentation**

**Specaugment**

There are various techniques that can be used for the process of data augmentation. In case of spectrogram we can apply frequency masking (horizontal lines to block a particular frequency) and time masking (vertical lines to block all the frequencies of a particular time).

**Raw data augmentation**

Raw data can be augmented by using the following techniques:

* Time Shift — shift audio to the left or the right by a random amount.
* Pitch Shift — randomly modify the frequency of parts of the sound.
* Time Stretch — randomly slow down or speed up the sound.
* Add Noise — add some random values to the sound.

**Data pre-processing and building the model (full code) for classification of voice:**

Refer the link below:

https://towardsdatascience.com/audio-deep-learning-made-simple-sound-classification-step-by-step-cebc936bbe5

**NAUTILUS**

Speech synthesis refers to generation of speech it can be through the use of a VC (voice converter) or TTS (text to speech). The NAUTILUS is based on the simple idea that both the VC and TTS are more or less the same things. Their input is different but when seen in light of voice cloning they are basically the same. Using this as the base the NAUTILUS networks aims at combining the two into one.

The main difference between voice cloning and speech synthesis is that the former aims producing the exact same sound as that of the target speaker whereas the latter aims at jut producing sound in a way to minimize the loss and therefore the output is not so natural.

The conventional methods used for voice cloning are text-dependent ie they expect a parallel input of both the speaker and the target speaker. Obtaining the data is expensive and hence this limits the length of data to be used to about 5 min and hence effects the output quality of the speaker. Therefore it is common to use non-parallel methods. One of such methods that formed the basis of the design of the NAUTILUS was the use of phonetic posteriorgrams (PPG) obtained from a ASR model. Since an ASR (Automatic speech recognition (ASR) refers to the task of recognizing human speech and translating it into text) is speaker independent, theoretically by using PPG we can convert the speech of arbitrary speakers into any desired speaker.

Next they try to adapt the system to an unseen target. The approach they follow is to train a speaker-adaptive model conditioned on a speaker representation extracted from speech. The speaker representations are the byproducts of the speech recognition systems.

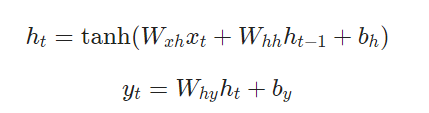
The ASR and TTS are generally thought to be independent systems with different architectures. Many tries have been made to combine the two. A similar kind of attempt has been made with the ASR and the VC. Hypothetically there is no difference between VC and TTS systems. Specifically a PPG based VC is a essentially a TTS stacked on top of a ASR.

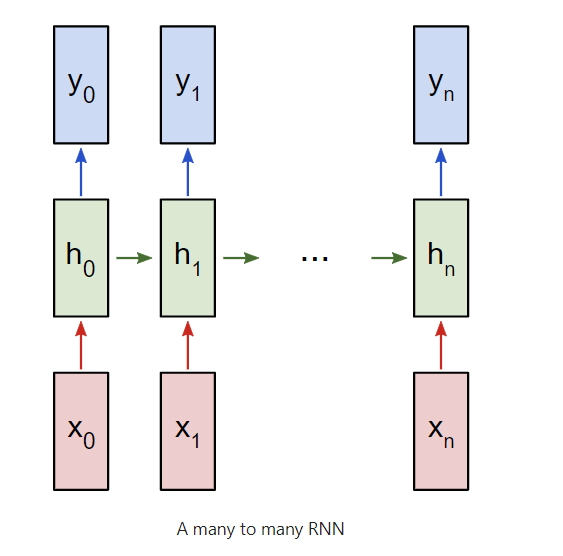
Architecture

The system that has been proposed can be used both as a VC and TTS. When the network encounters speech from an unknown data source, the core concept that is used is to train a latent linguistic encoding (LLE). This is what makes it similar to the PPG based VC design. In the latter case they trained their phonetic representation extractor with the VC but in this case the latent features are trained jointly with the speech generation model.

Recurrent Neural Networks (RNNs)

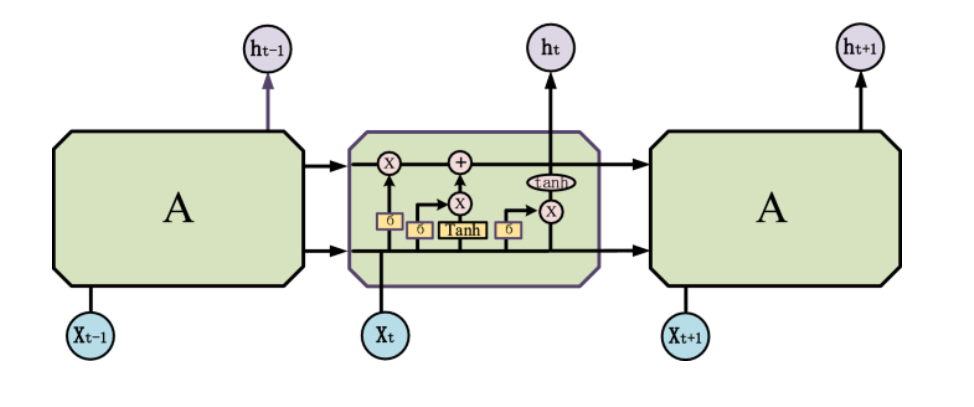
A recurrent neural network is designed to solve the problem of memory in neural networks. RNNs contain a feedback loop which is then passed into them. When this feedback loop is opened in time domain what we get is the following network. Here all the y\_i’s are the output of the RNN, x\_i’s are the input vector to the RNNs and finally the memory structures called the hidden inputs, h\_i’s. It is clearly visible that the hidden inputs are also passed along with the input vectors. The unfolding of the RNN along the time domain (that is shown below) does not mean that it is a series of networks as shown. RNN consists of a single block which has its weights and biases as parameters. But this time a single RNN unit has 3 weights – W\_xh, W\_hh and W\_hy. Also it has two biases, b\_h and b\_y. The outputs are given below:





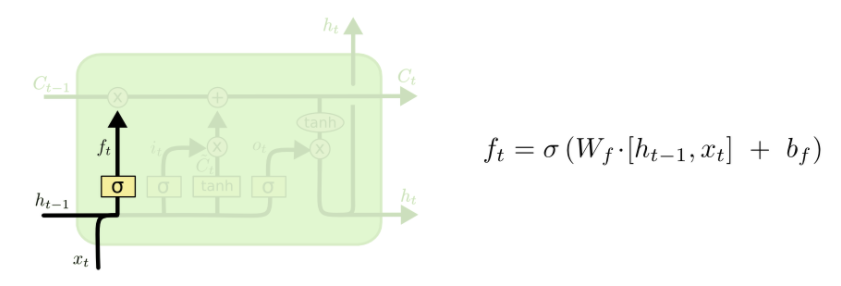
LSTM (Long Short Term Memory)

These are the special types of recurrent neural networks which have the ability to overcome the shortcomings of the traditional types of neural networks. The structure of the LSTMs is given below:

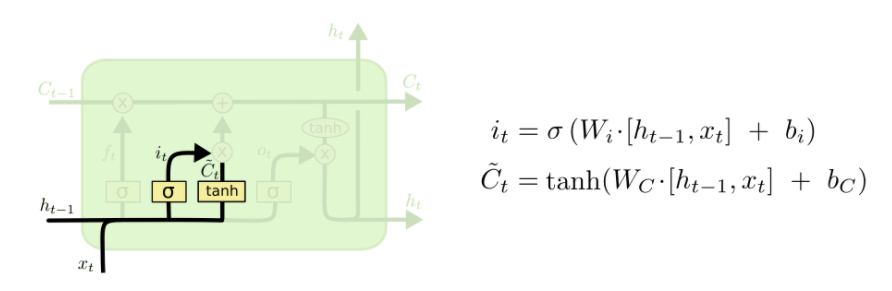


The major shortcoming of the RNNs is the problem of vanishing and exploding gradients. As more and more number of words are passed as input to the network the gradients that are calculated during the backprop get multiplied together by chain rule which could lead to the words that are input later to have less access to the memory and hence the output will not be as desired. These problems of learning the sequences of the long sentences can be solved by LSTMs. The LSTMs consists of 3 gates that help to solve this problem:

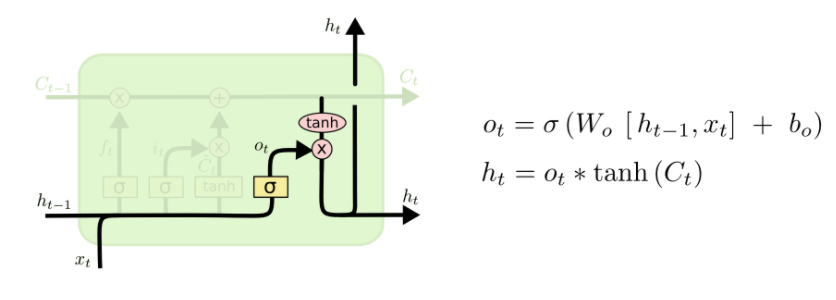
* Forget gate: This gate is used to tell how much information should be retained from the previous cell state. Hidden inputs along with input vectors are passed into the sigmoid activation function which basically squishes everything between 0 and 1. This in a sense contains the probabilities that the previous layer inputs (h\_i) will be retained. This is then multiplied with the previous cell state and this helps in filtering out of the unrequired information.



* Input gate: This gate tells us that after retaining the previous cell state, how much information should be added to the current cell state. In this gate there is an input gate that consists of a sigmoid activation function that squishes that input data between 0 and 1. And then that data is multiplied by the output of a tanh activation function that determines which of the values should be added or removed from the current state. This is done adding the cell state with the output of the input gate.



* Output gate: This gate helps to generate the output of the current LSTM cell. What values are to be outputted that is decided by passing the input values through the sigmoid activation function and then these values are multiplied by the cell state values which are first passed through a tanh activation function to decide which values are going to passed out.



Sequence to Sequence (seq2seq) Model

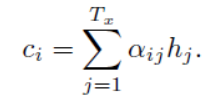
Attention

Attention mechanism is the method that is used to mimic the way humans perceive things. This is generally is done by focusing or by paying attention to a particular features and not looking at all things for making a simple prediction. For example - suppose we have a school photograph and in the photo we are asked to identify the how many students are present in the photo. For doing the above described task, looking at all the features and then giving the output, would be computationally expensive and unnecessary. For this we could simply look at the features that correspond to the face for each person. This is where attention comes in.

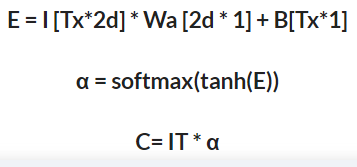
In early networks RNNs/LSTMs were used to predict the next word by producing the context vector which in this case was generally taken to be the output of the last hidden unit of the RNN/LSTM. This was done because it was expected that the last output of the hidden unit would be a good summary of the entire system. This was accompanied by ignoring all the intermediate hidden outputs from the other units.

The LSTM was used to convert the sentence to produce a sort of summary and on the basis of that summary the outputs were further passed into the decoder LSTM/RNN network which would work on the basis of the summary that was produced by the encoder. So naturally if the summary produced by the encoder was not good enough, the output of the decoder network will also be affected. RNNs cannot perform well on long sentences because of the exploding and vanishing gradient problem. Although the LSTMs can perform better with long sentences but they become forgetful of specific instances and they also pays same attention to all the words in the sentence. This calls for attention.

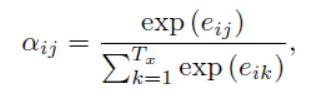
The first attention network was implemented by performing a weighted sum of all the hidden states this sum is now the context vector. These weights are learned during the feed forward neural network. The context vectors are generated by the equation.



The alpha\_i in the above equation are calculated by performing a simple matrix multiplication between the input vectors and the weight matrix of appropriate dimensions a d adding a bias term to the resulting expression. The result of the above described transformations is then passed into a hyperbolic tan function and this is then followed by passing the output into a softmax function.



Here I is the input vector, Wa is the weight matrix and B is the bias term. This produces the embedding score, E. This is then passed into the tanh function which is the passed into a softmax function. Which is then multiplied by the input vector to produce the context vector for the decoder network.



The above architecture is generally called the global attention because we pay attention to all the hidden outputs to generate the context vector. Technically, the term global attention is used when the outputs from both the encoder and decoder are taken into consideration for computing the context vector. However in the model that is described above we have only used the hidden outputs from the encoder network of the model. However the ‘true’ global attention network would be computationally very expensive to implement because of the large number of computations involved. Therefore we use local attention instead of global attention. Local attention as the terms suggests takes in a part of all the encoded inputs (hidden outputs) instead of taking all of them. This is achieved by predicting a position pt in the sequence of embedding of the given input words. Around the position pt it considers a window of 2D i.e. from pt-D to pt+D and the context vector is generated by performing a weighted sum of the vectors which are present in the window. Here D is a hyperparameter. Now the question arises that how do we decide the position of the the point, pt. This can be done in the following two ways:

* + Monotonic alignment – here pt is set to t, assuming that at time t only information around the neighbourhood of t matters.
  + Predictive alignment – In this the model itself predicts the position as described as follows.



Basic working

Speaker Encoder

For speaker verification they use a network that can map a series of log-mel-spectrograms which are computed from the speech utterance from the speaker to a fixed dimensional vector of arbitrary length known as d-vector. The network is trained so that the embeddings from the same speaker have a high cosine similarity whereas embeddings from different speaker are far apart in the embedding space. This is done by stacking 3 LSTMs and passing 40 log-mel-spectrograms. The final embedding is created by L-2 Normalizing the output of the top layer at the final frame. These embeddings are then passed on, which are then concatenated with the embedding coming from the synthesizer network which in this case is the tacotron 2 network.

Tacotron 2

In this the input text is represented by using learned character embedding which is then passed through the 3 layers of 1D convolutions. These convolution layers are used to capture important features from the character embedding which help in capturing the long term context. These are then passed into a bidirectional LSTM which is used to produce a summary of the sentence which can easily be interpreted by the decoder for producing the log-mel spectrograms. The bidirectional LSTM in its way to produce the summary produces hidden units. The weighted sum of the hidden units multiplied with the input are then passed for further processing. The result of the weighted sum of the hidden units leads to the generation of the context vector. This mechanism is called the attention network. In this case instead of using the primitive bidirectional LSTM we can also use the State of the art networks, transformers.

The above steps produce a summary of the input text (context vector) which is then passed into the decoder network which works on the basis of the summary given to it and produces an appropriate log-mel spectrogram. Then there is a network called prenet. This is a 2 layer convolution network which is essential for the task of learning attention. This network takes in the previous time step prediction which is then passed on and concatenated with the output of the attention network. This concatenated output is then passed through 2 stacked unidirectional LSTMs. The output from the LSTMs is then again concatenated with the context vector which is then passed through a linear transform which is used to predict the target spectrogram frame. Finally this spectrogram is then passed through a 5-layer convolution post net. This post net is used to predict the residual which is to be added to improve the overall reconstruction.

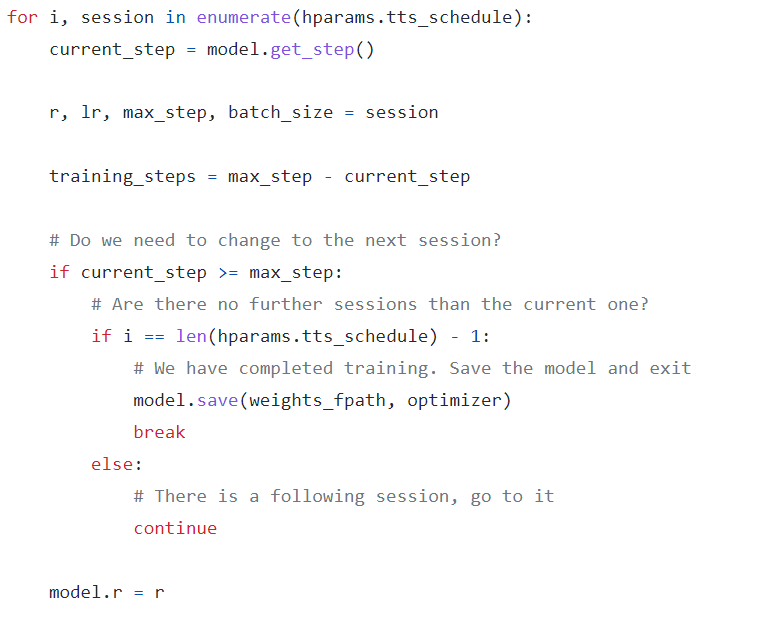
This process of reconstruction is a cyclic process and needs to stop after a certain period. For this there is another network which runs in parallel to spectrogram frame generation. In this the concatenated decoder LSTM output and the attention context vector is passed passed through a linear projection layer which projects it down to a scalar. This scalar is then passed into a sigmoid activation function which predicts the probability that the process of reconstruction is complete. This is then followed by a stop token which helps in addressing the network to stop the process of generation instead of running the process for a fixed amount of time.

Then the predicted log-mel spectrogram is passed through the Wave net vocoder which uses the techniques of inverse short time fourier transform to convert the log-mel spectrograms to the audio waveforms. The wavenet is basically made up of convolution layers specifically dilated convolutions. The use of dilated convolutions allows us to capture a larger receptive field as compared to basic convolution. After the convolutions the architecture also has a tanh-sigmoid gated unit this is then followed by some skip connections followed by a couple of layers of ReLU activation function and then finally a softmax function is applied. Now instead of taking one of the ouputs of the softmax function a weighted sum of the outputs is used. This is done to avoid the network from getting stuck in a sequence of same words causing the prediction to stagnate.

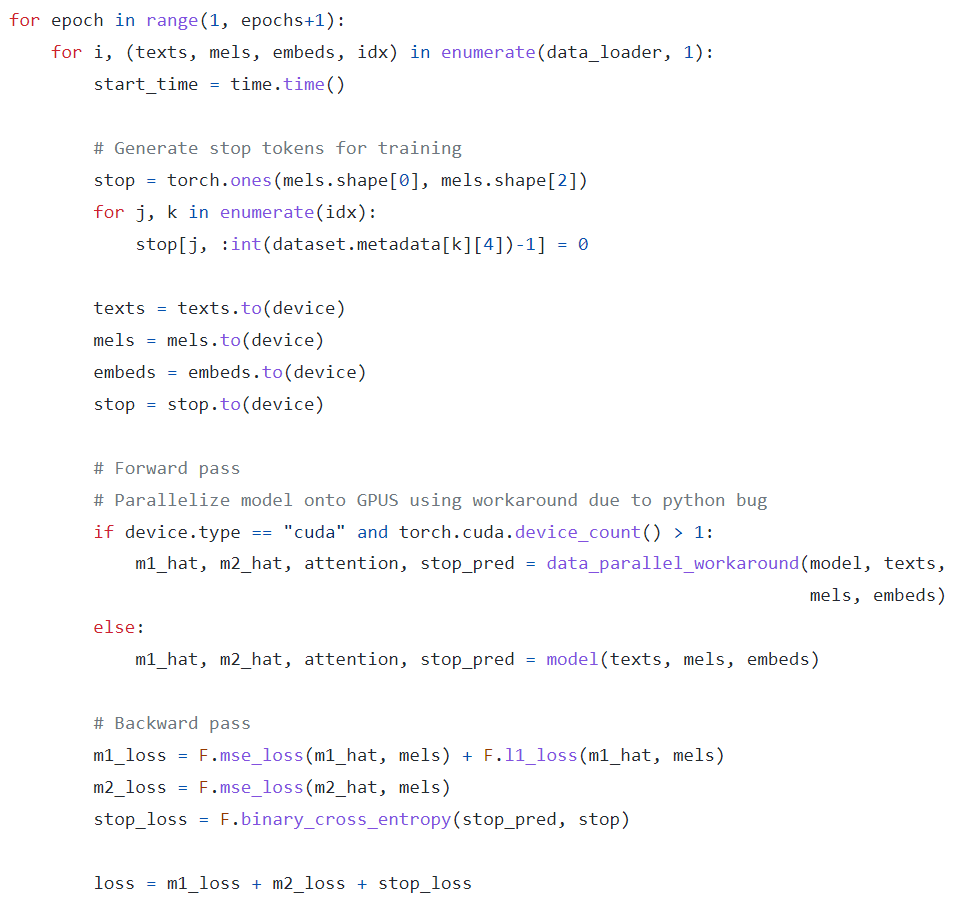
Training cycle

For the training of this architecture, each of the models that are created, are trained separately. So the encoder, synthesizer and the vocoder are trained separately. The encoder takes audio in the form of mel-spectrograms as input and then generates character embedding for that input speech. The encoder in this case consists of a LSTM, Linear layer and ReLU activation. Now for producing the embedding they have only considered the final output of the LSTM cell. This output contains cell state and hidden output. Then the hidden output is used as an embedding for inputs. The embedding is then normalized by dividing the obtained embedding by the L2 norm. Now these embedding are trained.

For training the synthesizer network, all the required data is imported from the respective directories and loaded into the models. Then the appropriate devices are loaded this is followed by instantiating the model. The we initialize the optimizer then if the weights are saved, they are loaded along with the metadata of the embeddings. If that is not the case then the weights are initialized. This is followed by initializing the dataset and the dataloader. This is followed by a normal training loop which contains



Why is there any extra loop (above)?



Main training loop (above)

The last training loop is normal training loop.

Datasets Used

|  |
| --- |
|  |
|  |
|  |

* Librispeech
* voxceleb1
* voxceleb2
* LJSpeech-1.1
* VCTK-Corpus/wav48

Data Augmentation

Adding noise

wn = np.random.randn(len(data))

data\_wn = data + 0.005\*wn

plot\_time\_series(data\_wn)

*# We limited the amplitude of the noise so we can still hear the word even with the noise,*

*#which is the objective*

ipd.Audio(data\_wn, rate=16000)

Shifting the sound

data\_roll = np.roll(data, 1600)

plot\_time\_series(data\_roll)

ipd.Audio(data\_roll, rate=16000)

Stretching the sound

def stretch(data, rate=1):

input\_length = 16000

data = librosa.effects.time\_stretch(data, rate)

if len(data)>input\_length:

data = data[:input\_length]

else:

data = np.pad(data, (0, max(0, input\_length - len(data))), "constant")

return data

Changing Pitch

import librosadef manipulate(data, sampling\_rate, pitch\_factor):  
 return librosa.effects.pitch\_shift(data, sampling\_rate, pitch\_factor)

Changing speed

import librosadef manipulate(data, speed\_factor):  
 return librosa.effects.time\_stretch(data, speed\_factor)

Data Preprocessing

The encoder requires takes in as audio signal in the form of a spectrogram. For doing this firstly we load the file using librosa.load. In this we also pass the argument of sampling rate. If this is passed as None then the audio file will be sampled at native sapling rate (librosa documentation). Now we check for the need of resampling the data i.e. we check if the sampling rate that was initialized by the load function is equal to the sampling rate that the user wants. If these are not the same then the resample function is applied to sample the audio file at the desired rate. Next they normalize volume and trim the parts with silences which are longer than a certain time period. In the part of the code where they trim the longer silences, they have also ensured that the voice detection is smooth for this they have used moving averages. Next step is to convert the raw data into mel-spectrograms so that it can serve as an input to the encoder. This is done by using librosa.feature.melspectrogram. The output is a mel-spectrogram not a log-mel-spectrogram. This can be converted into a log-mel-spectrogram by converting the function amplitude\_to\_db giving the input as the mel-spectrogram obtained above.

For the case of synthesizer, dataset which have text are only used to train the synthesizer ***(confirm by opening the datasets).***  Firstly all the files with all the possible extensions are loaded onto the model and then we rescale the file by dividing it by the maximum possible value of the audio file and then rescale it by using the hyperparameters which contains the max\_rescale value. Then we check for the corresponding text. This is done by checking for the files with extension .txt if these files are not available then we check for normalized.txt files. If the text files are not available then we use LibriSpeech which contains an alignment file. This file contains the words which are coma separated.

The proper training cycle that has been used in the model is that the all the different models are trained separately. This cycle therefore requires two datasets which are used by the different networks in the model. These models are trained separately but they still follow a certain order. First we train the encoder network of the model using the dataset that contains recorded utterances from various speakers. These utterances need to be converted into log-mel spectrograms by using various preprocessing techniques. The could then serve as input for our model to make predictions. Then we follow the common training cycle – making predictions, computing loss, calculating gradients and then updating the model parameters. Once the training of the encoder is done then we can start the training of the other networks in the model namely – the synthesizer and the vocoder. Next we turn our attention to the synthesizer network. The synthesizer network which is basically a TTS (text-to-speech) network, takes as input the text from the second dataset (this is the dataset that needs to have the audios as well as the corresponding text files or captions for the speech) and along with we will have to run the trained encoder in parallel. For that the audio files of the second dataset are converted into log-mel spectrograms. These converted spectrograms are then fed as an input to the encoder that generates the speaker embedding for the inserted log-mel spectrograms. The speaker embedding along with the text is then fed into the synthesizer network. This synthesizer network is basically a tacotron network which then computes the predicted log-mel spectrograms for the data that is inputted. Now a simple loss function is used to calculate the loss. This loss function its input as the predicted lo-mel spectrograms as well as the original spectrograms which were calculated by converting the audio files. Based on this loss the gradients are computed and the then these are used to train the synthesizer network. Now finally we have to train the remaining network of the vocoder. For this all the networks will be used simultaneously. Now for this training process we will first follow all the processes that were done earlier. So the audio files which were converted to log-mel spectrograms are first fed into the encoder which generates the corresponding speaker embedding. In the next step these speaker embedding are passed into the synthesizer network along with the text. This generates the predicted log-mel spectrograms. These predicted log-mel spectrograms are then passed into the vocoder which produces the corresponding waveforms. These waveforms along with the real audio samples are used to compute the loss which is used to compute the gradients that are used to train the model.

Symbols for the terms used

U\_ij – This denotes jth utterance of the ith speaker

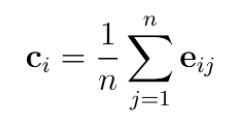
X\_ij – denotes the log-mel spectrogram (function to extract features from the waveform) of the U\_th utterance

E\_ij – Embedding corresponding to the u\_th utterance

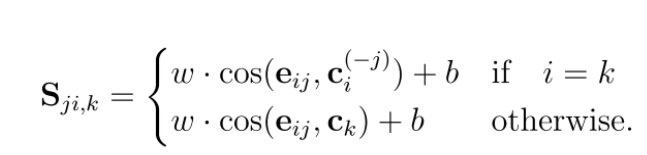
C\_i– Speaker embedding

T\_ij – transcription of the utterance u\_ij

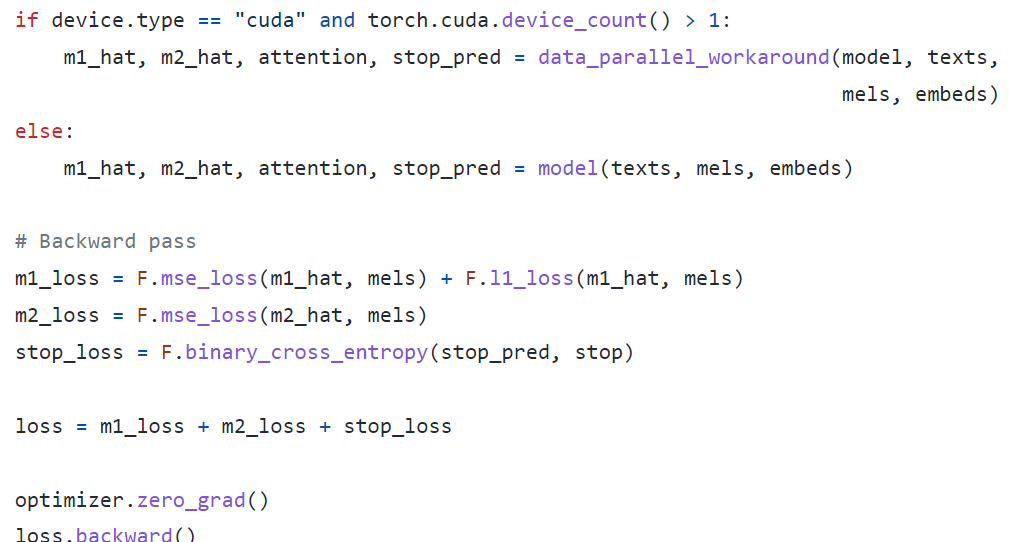
Now the function of the encoder is to compute the embedding, e\_ij = E(x\_ij, wE) here wE are the set of parameters of the encoder model and E is a general function that computes these embedding. The speaker embedding is calculated by finding the centroid of the embedding of the speaker utterances. The speaker embedding can be calculated by



The synthesizer S is then fed c\_i and t\_ij as input and this is used to model x^ij. Therefore we have x^ij = S(c\_i, t\_ij, wS), where wS are the parameters of the synthesizer model. But in this case the we use a different model x^ij = S(u\_ij, t\_ij, wS). The vocoder is used to estimate u\_ij for a given x\_ij. Therefore we have u^ij=V(x^ij, wV). The loss function that is used to train the encoder model is the GE2E loss function. The encoder takes M utterances of fixed duration from N speaker and then creates embedding (e\_ij) using them. The dimensions of these embedding are NxM. Then the above calculated speaker embedding (c\_i) (Dimensions are Nx1) are compared to the embedding of all the speaker embedding by using a similarity matrix, which is expected to output high similarity values in case of an optimal model. To optimize, the loss will be calculated by row-wise Softmax losses. When embedding, the utterances are included in the centroid of the same speaker. To avoid the bias that is created towards the speaker independently of the model accuracy, an utterance that is compared against its own speaker's embedding will be removed from the speaker embedding. The similarity matrix will look like this:

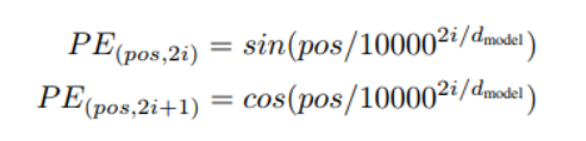


The samples are then preprocessed and the silent parts of the utterance are removed with the help of a python package (librosa). He used the dilation technique (s+1) where S is the maximum silent duration tolerated.



**Transformers**

The encoder is basically a network that can be thought of as taking the words in the sentences and forming a compact representation of the sentence in general. The word embedding layer can be thought of as a look up table which contains the learned representations of the words in the form of vectors. Now in the case of transformers we do not have any way to specify the positional information as it was lost since we are feeding the network with the whole sentence at a single time. This was not the case with the traditional RNNs, GRUs and LSTMs. Unlike the previous stated examples, transformers have infinite lookback capacity for the context formation as they take in input the whole of the sentence. This was not the case with the normal RNNs, Grus and LSTMs which on the other hand have a limited capacity for context formation and consequently a limited context vector. The earlier stated networks take a single word at a time and ence have the positional information which they can leverage later. That is not the case with the transformers. Therefore in the transformer network there is a need to explicitly input the positional information for the sentence formation. This has been done in the paper with the help of the sine and the cosine function using the following equations:



For every even time step we use the sine function and for every odd time step we use the cosine function. Then this positional encoding information is added to the word embedding that are generated at the corresponding time steps.

The encoder network then consists of a multi-headed attention and the feed forward network. For the multi-headed we feed the embedding with the positional embedding to three fully connected layers which produce as output the query, key and the value. These are similar to like when we perform a search on YouTube. We type out a query in the search box and then this query gets mapped to the headings of the databases and then the values are returned to us which are in accordance with the queries. Here the values can be thought of as the context vector that is formed and which is then multiplied by a set of values (a vector) which contain the information of how much of each thing from the context is required. The queries and the keys undergo a dot product matrix multiplication, to produce the attention scores. For arriving at the attention scores, first step is to create the score matrix. The score matrix contains values which tell how much a single word is related to the other words. Then the score matrix is divided by the dimensionality of the key matrix i.e. the number of words in the sentence. This is done to avoid the problem of exploding gradients. This then produces the scaled scores matrix. Then we take the softmax function of the values in the scaled score matrix. This then outputs the attention scores. These contain the values which are the information of how much word has to used from the values matrix. Then we put this in the linear layer. The process that is described above is for the single-headed attention. For using multi-headed attention we produce the query, key and values separately and then these follow the procedure described above and one unit is called the head. All the outputs from the various layers are then concatenated and then passed into a linear layer for producing the output. This output is in the form of vectors which contains the information of input vector in the compressed form. The multi-headed attention output vector is added to the original input via a residual connection and the output of the residual connection goes through a layer normalization. The normalized output is fed to point wise feed forward network which also contains a ReLU activation layer. This is the transformer encoder. The information that is received from the encoder is a representation of the input data which is the mixture of the all the important words that the decoder needs to pay close attention to while performing the reconstruction from the input data that has been encoded by the encoder. We can stack up many layers of the encoder network to boost up the power with which the encoder produces the context vector and this helps in learning deeper and richer representations of the input data which can then be used by the decoder. The decoder network takes in the target sequence and then in a similar fashion as in the encoder network, the embedding for the words as well as the positional embedding are generated which are then concatenated and fed to the first multi-headed attention layer which computes the attention scores for the input target sequence. Before converting the input target sentence into embeddings, the input sentence gets a start token in the beginning of the sentence and an end token at the end. This sequence needs to be prevented from looking in the forward direction because it is a regressive model. This is done by producing a mask for the words come later in the sequence. This prevents the network from looking at those words for building on the output sequence. Look-ahead mask is used for this purpose. This mask is basically a matrix which contains values of 0 and –inf and each value is basically a relation of the word in the column with the word in the row. The –inf values are attributed to the words that come later in the sequence. This masking matrix is added with the scaled scores that are produced from the multi-headed attention layer and then this layer is passed through a softmax which leaves the probability of these words to be zeros and therefore are not considered in the process of word generation by the decoder network. The values that are produced from the softmax function basically contain the information of the words that the decoder needs to pay attention to during the process of sequence generation. The second multi-headed attention layer takes as input from the encoder and the values from the first multi-headed layer. The output of the second multi-headed layer goes through a point wise feed forward layer. The output of this layer goes through a final linear layer which acts as a classifier which outputs the probabilities of the words that are to come next in the sequence. This output sequence is then input into the decoder network. This process is continued until the end token is produced. Similar to the encoder the decoder can be stacked many layers high so that the output of the decoder is also fine-tuned. The transformer is a regressive network which takes in output as it is produced and builds upon it word by word. During the time of training the network we input the word sequence and input the target sequence as well. All the sequences that are input in the decoder network are initialized with a start token and an end token. This is used for the training network. Then the output of the decoder network is compared to the target sequence to compute the loss which is then used for training the network. Now in the time of inference, we do not have the target sequence and therefore it uses the a start token and then the output of the decoder network gets fed the output as it is generated regressively and then it is used along with the encoder network to build on the output sequence.

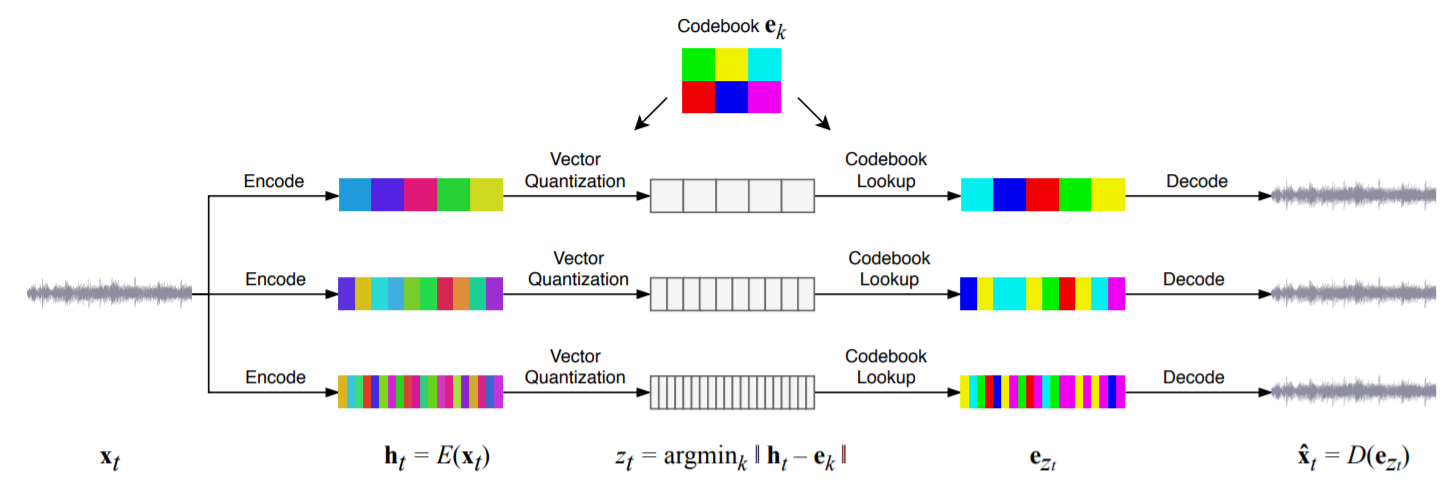
**VQ-VAE (Vector-quantized variational autoencoder)**

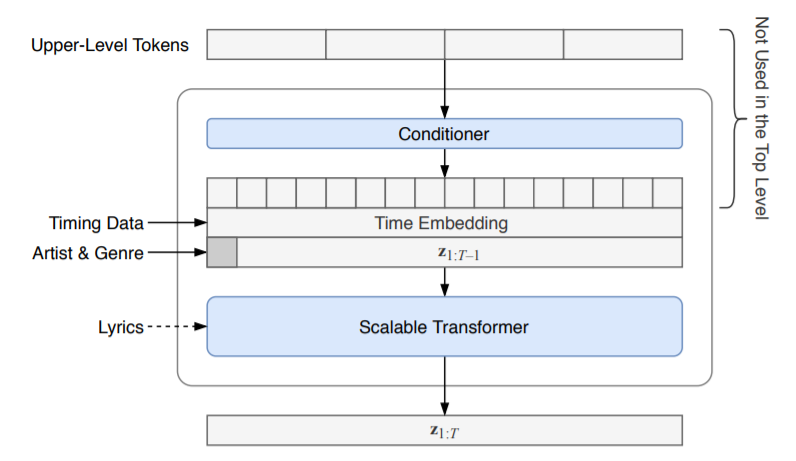
VQ-VAEs are different from the autoencoders and variational autoencoders. The autoencoders learn a latent space by compressing the input data and try to reconstruct the input data by using the vectors from the latent space. This latent space comprises of vectors which are basically the features that are all over the latent space. Even the features that have a close resemblance or relation are far apart or randomly distributed and not related thus making learning and sampling from the latent space very difficult and inefficient. This difficulty is overcome by the variational autoencoders. These networks take in a prior which is basically a distribution that we expect the data to follow beforehand. The networks then along with sampling from the latent space return values of the mean and the variance for the distribution (in case the distribution is a Gaussian) which makes it easy to sample and moreover the vectors which are represented in these spaces are grouped on the basis of feature similarity which makes sampling a lot easier. The spaces that are learned by the variational autoencoders are continuous spaces. Therefore if we sample from any point in this latent vector space we will get a vector from that space which has the features of the nearby prominent features. This job of grouping the vectors on the basis of features is accomplished by enforcing a probabilistic prior (this is a distribution which the data is assumed to be in beforehand) and therefore we train the model in accordance with this prior and the force the model to learn a distribution which is similar to that of the prior and this distribution is also modified on the way as the model is trained. In the variational autoencoders the data that is produced by the encoders is not directly fed to the decoder instead it is used to parameterize the priors (for example: compute mean and variance in the case of gaussians) that were already input to the model. Then from these gaussians (say) you sample points which are then clubbed together to form a vector which is then input into the decoder. These are the variational autoencoders. The vector-quantized variational auto encoders are different from the variational auto encoders in the way that the variational autoencoders learn a continuous space distribution and the vector quantized variational autoencoders on the other hand learns a discrete space. The possible reason why they learn a discrete representation is because all the things are discrete in nature and moreover there is no specific need to learn a continuous space when the same thing can be represented with limited vectors. The input of the decoder is not a vector containing a single value; they are generally in the form of a matrix (in case of an image) and therefore the possible number of combinations that can be filled in these matrices grow exponentially as the number of vectors from the codebook increase. Also another reason for discrete representation of the latent space is that a large number of algorithms are designed to work on the discrete data. The variational autoencoder is made discrete by the addition of a codebook. The codebook contains some specific vectors which contain information about the features of the input data. The vectors that are present in the codebook aslo get learned on the way the training is done. So the output of the encoder network of the autoencoder is compared with all the vectors present in the codebook this is done by comparing them using the mse (mean squared error) loss and that is minimized to assign the vector closest to the vector in the codebook vector. The loss function that is used for the training purpose is:

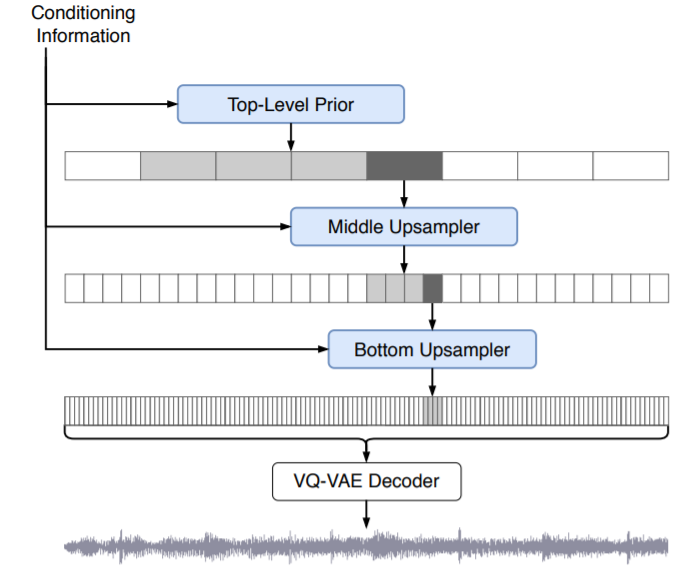


The first term is used for the reconstruction loss i.e. reconstruction of the input data using the hidden space which gets learnt on the way. The second term is used for mapping the each of the vectors which are output from the encoder to the vectors present in the codebook therefore there is a stop gradient in the encoders output. The third term is used to train the encoder to get the output vectors as close as possible to the codebook vectors therefore there is a stop gradient (sg) on the codebook vectors (e).

Jukebox



The above figure showcases how the jukebox works. First we send an audio signal and then the same audio signal passes through three vector quantized variational autoencoders. All the three VQ-VAEs have different sampling length through which they sample the given audio. The first one takes samples that are coarse thus capturing the long range dependencies of the music sample. The middle one captures the mid-range dependencies and the shortest one is used for improving the reconstruction loss and basically learns the short term details. Only the short ones aren’t used because they might cause overfitting. 

Now each of the three layers has transformers which are used for predicting the next words by using the embeddings that are learnt and taken from the latent space of the autoencoder. First the layer that takes coarse samples from the input music is taken and the VQ VAE produces a encoding for the input music. Subsequently, this encoding is then passed on to the next VQ VAE which learns more finer details by upsampling the data that it gets as input from the previous VQ VAE. Similarly the output of this layer gets input to the next layer which upsamples the data. For performing this upsampling they have used the upper-level tokens. These tokens are taken from the upper layer and are then passed in the conditioner network then their embedding vectors go through non-causal WaveNet-like layers with increasing dilated convolutions. They are using the transposed 1-D convolutions for upsampling the data. The data from the conditioner network is passed in the next layers where the positional and the timing information are imparted in the form of vectors. Additional information like the artist and the genre of the music are also imparted by producing vectors for each one of them and then adding them up and adding them to the data. This prepares the embedding vectors that are then passed onto the scalable transformer. This transformer in the first level takes lyrics as its input and then embedding vectors for these lyrics are generated. 

These embedding go through a series of layers which contain the attention layers which are used for producing finer embedding are then send to the module which acts like the decoder in the transformer. This layer also takes as input then VQ codes of the music samples and then they go through various layers which contain the attention mechanism and in the encoder-decoder attention layer the lyrics embeddings are given a input. After going through a series of attention layers VQ code embeddings are generated and these are then passed onto the next layer for further processing. The next layer which contains similar attention networks and then short term details of the music sample are added and then these are then passed through the decoder network of the VQ VAE which tries to reconstruct the music.

