DD2465 Advanced, Individual Course in Computer Science

Speech Synthesis

Replicating MelNet: A Generative Model for Audio in the Frequency Domain

Jorge García Pueyo

KUNGLIGA TEKNISKA HÖGSKOLAN
Stockholm, Sweden 2020

Contents

List of Figures

List of Tables

1 Introduction

Speech synthesis consists on artificially creating human speech. It can be categorised into unconditional speech synthesis, which can be defined as generating random babbling similar to the sounds that a baby would produce, and conditional speech, also known as text-to-speech where the sound produced is conditioned on text.

There exist different approaches to speech synthesis: concatenative speech synthesis based on the concatenation of sounds of small duration, and parametric speech synthesis based on models where the information is stored in the parameters (i.e. using a Hidden Markov Models (HMM) or deep learning model).

Parametric models work on various audio representations. Some of them use waveforms, which are one-dimensional time representations, like WaveNet[1]. Other models use spectrograms, which are two-dimensional time-frequency representations, like MelNet[2].

MelNet, a generative model for audio, takes advantage of the time-frequency representation to capture global structure dependencies using a highly expressive probabilistic model that factorizes the distribution of the spectrograms over time and frequency axis. To produce high-quality audio, MelNet produces high-resolution spectrograms using a multiscale generation approach.

The objective of this project is to implement MelNet (S. Vasquez and M. Lewis, "Melnet: A generative model for audio in the frequency domain," *arXiv preprint arXiv:1906.01083*, 2019), for which there is no de facto implementation, and train it with a dataset containing audio from a dialogue-based podcast to generate audio which is closer to a conversation.

This report has a similar structure to the MelNet paper. In this report, chapter 2, chapter 3 and chapter 4 serve as an extension to the explanations of the paper, adding insights into the structure and formulae of the model gained during the course of this project as well as information about the implementation that is not specified in the paper. For this reason, and to give credit to the authors of MelNet, the explanations that are in bold are the same in these chapters and in the paper. In chapter 5 we explain how we have implemented MelNet MelNet in PyTorch and the tricks to train it. In chapter 6 we present the results of the project.

The source code of the project can be found in:

`https://github.com/jgarciapueyo/MelNet-SpeechGeneration` .

2
Audio Representation

2.1 Theory behind audio representation

Sound is transmitted through the air as pressure oscillations which can be represented by a pressure-time plot, also known as waveform. The plot shows the deviation of air pressure from normal state. A waveform is represented digitally as a discrete-time signal $y = (y_1, ..., y_n)$. When the waveform is periodic, we can identify several characteristics like the period, which is the time to complete a cycle, or the frequency, which is the inverse of the period. Other characteristics are the amplitude, which is the maximum oscillation of the waveform from its mean, and the phase, which is where in the cycle the waveform is at zero time.

Real world waveforms are far from being periodic but any of these more complex waveforms can be constructed by adding simple sine waves of different frequency, amplitude and phase together. The process that allows to know what the simple sine waves are is the Fourier Transform.

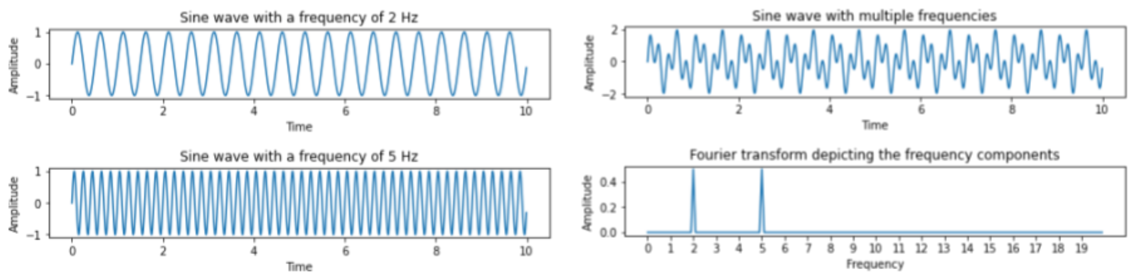


Figure 2.1: On the left, there are the periodic waves of 2 Hz and 5Hz. On the top right, there is the periodic wave resulting from adding the two periodic waves. on the bottom right, there is the result of applying the Fourier Transform.

The Fourier transform gives information that is averaged over the entire domain, but it does not give us information about when the frequencies occur in the entire waveform. We use the Short-Time Fourier Transform that consists on applying the Fourier Transform to a small section, a window, of the waveform. This window is shifted across time and by repeatedly applying the Fourier Transform, we obtain a representation known as a spectrogram $x = \|STFT(y)\|^2$. To be more specific, this is an amplitude spectrogram, which is computed by taking the square of the magnitude and discarding the phase information, both values computed by the Fourier

Transform. This two dimension plot represents time and frequency in the horizontal and vertical axis respectively and the value x_{ij} represents the amplitude of the j^{th} frequency at the i^{th} timestep.

The Fourier Transform returns a complex number, representing the amplitude and the phase, but we discard the phase information and only plot the amplitude in the spectrogram. This is why it is said that the amplitude spectrogram (or magnitude spectrogram) is a lossy representation of the corresponding time-domain signal. Also, the formula computes the squared magnitude and this depends on the type of spectrogram to generate.

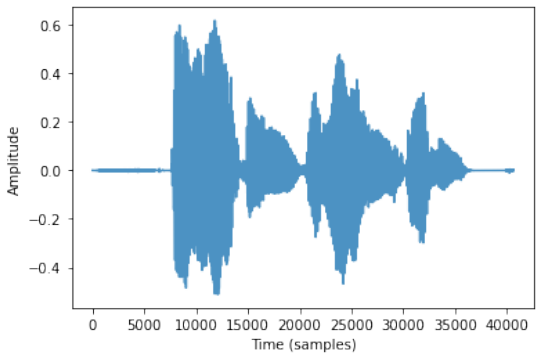


Figure 2.2: Audio waveform of real speech.

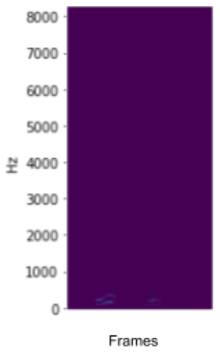


Figure 2.3: Spectrogram corresponding to fig. 2.2.

Basic spectrograms are not aligned with how humans perceive audio and this can be seen in fig. 2.3, where we can not extract visually any information. To better align the spectrogram with how humans perceive sound, we transform the frequency axis to the Mel scale fig. 2.4 and the amplitude values to the decibel scale fig. 2.5.

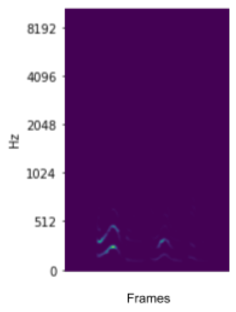


Figure 2.4: Melspectrogram corresponding to fig. 2.2.

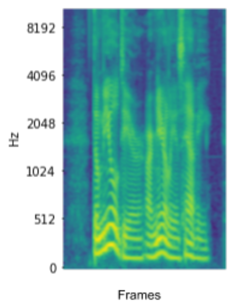


Figure 2.5: Melspectrogram in decibel scale corresponding to fig. 2.2.

The transformations presented here allows us to go from the audio waveform to a melspectrogram in the decibel scale. This last representation is the one used by

MelNet to generate audio. To recover the waveform audio, first we reverse the decibel transformation. Because it is a logarithmic rescaling there is no loss. Later, we recover the spectrogram from the melspectrogram, although this is a lossy transformation. Finally, we need to transform a spectrogram into the audio waveform. For this, we use the classical spectrogram inversion algorithm Griffin-Lim [3].

2.2 Implementation of the transformations between the audio representations

This project uses Python and the framework PyTorch for the deep learning model. PyTorch contains packages for different domains like vision or audio. However, torchaudio was a package released recently, less than a year before at the start of the project. In order to guarantee that torchaudio was mature and the implementation of the audio transforms were correct, it was compared against librosa, the reference package in Python to work with audio.

Every transformation mentioned before was implemented in both libraries and compared against each other to see how much information was lost in each transformation. This is useful also in the project in case the results are different from what it is expected, because it allows to identify the source of the error by isolating the different parts of the data pipeline.

The results indicated that torchaudio has similar information loss compared against librosa in any of the transformations. Considering that information loss is similar, we will use torchaudio because of the better interoperability with PyTorch. In addition, when performing the whole transformation pipeline, from waveform to melspectrogram in decibel scale and back to recover the audio waveform, we compared the quality of the original and the reconstructed waveform and both sounded extremely similar. We can conclude that these transformations do not affect the quality of the audio in an audible way.

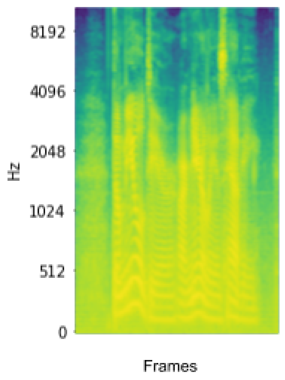


Figure 2.6: Melspectrogram corresponding to fig. 2.2 with decibel scale applied before mel scale transformation.

In the paper, it is specified that MelNet works on melspectrograms in decibel scale, however it is not specified which transformation is applied first, if melscale or decibel scale. Looking at other sources, it does not appear to be any information related to this, so we compared both approaches and found that applying melscale first is better. If the decibel scale is applied before, the resulting melspectrogram looks blurrier fig. 2.6 and there is a greater loss when recovering the audio waveform.

3
MelNet: Single-Tier

3.1 Probabilistic Model

3.1.1 Definition of the Probabilistic Model

MelNet is an autoregressive model which factorizes the joint distribution over a spectrogram x as a product of conditional distributions. The joint density is factorized as

$$p(x;\theta) = \prod_i \prod_j p(x_{ij}|x_{<ij};\theta_{ij}) \tag{3.1}$$

where θ_{ij} parameterizes a univariate density over x_{ij}, and $x_{<ij}$ are the elements of the spectrogram previous to x_{ij}. We will consider the spectrogram as a sequence of frames, $x_{i,*}$, and inside each frame we go from low to high frequencies, so $x_{<ij}$ consists of all the previous frames $x_{<i,*}$ and of the lower frequencies in the same frame $x_{i,<j}$. **Each factor is modelled as a Gaussian Mixture Model with K components $\theta_{ij} = \{\mu_{ijk}, \sigma_{ijk}, \pi_{ijk}\}_{k=1}^{K}$. The factor can be described as**

$$p(x_{ij}|x_{<ij};\theta_{ij}) = \sum_{k=1}^{K} \pi_{ijk} N(x_{ij};\mu_{ijk},\sigma_{ijk}). \tag{3.2}$$

Knowing the values of θ_{ij} for all i and j, MelNet is a simple probabilistic model. The complexity resides in how the parameters are estimated. Similar to Mixture Density Networks [4], MelNet makes the parameters θ_{ij} of the Gaussian Mixture Models to be governed by the output of a neural network f with weigths ψ as a function of the context $x_{<ij}$, i.e. $\theta_{ij} = f(x_{ij},\psi)$. **To ensure that the network output parameterizes a valid Gaussian Mixture Model, the paper assumes that the output of the network are unconstrained parameters $\{\hat{\mu}_{ijk}, \hat{\sigma}_{ijk}, \hat{\pi}_{ijk}\}_{k=1}^{K} = \hat{\theta}_{ij} = f(x_{ij},\psi)$ and applies the following constraints**:

$$\mu_{ijk} = \hat{\mu}_{ijk} \tag{3.3}$$

$$\sigma_{ijk} = exp(\hat{\sigma}_{ijk}) \tag{3.4}$$

$$\pi_{ijk} = \frac{exp(\hat{\pi}_{ijk})}{\sum_{k=1}^{K} exp(\hat{\pi}_{ijk})} \tag{3.5}$$

The architecture of the neural network used as the function f is explained in section 3.2.

3.1.2 Loss of the Probabilistic Model

MelNet uses maximum-likelihood estimation for the parameters $\theta = \{\theta_{00}, ..., \theta_{ij}, ...\theta_{frames,freq}\}$ by minimizing the negative log-likelihood via gradient descent. This means that it uses the negative log-likelihood of a spectrogram x with respect to the parameters θ as the error function of the neural network and modify the weights of the neural network ψ accordingly using gradient descent.

$$
\begin{aligned}
loss(x, \theta) &= -log\ \mathcal{L}(\theta|x) && (3.6)\\
&= -log\ p(x;\theta) = -log \prod_i \prod_j p(x_{ij}|x_{<ij};\theta_{ij}) && (3.7)\\
&= \sum_i \sum_j -log\ p(x_{ij}|x_{<ij};\theta_{ij}) && (3.8)
\end{aligned}
$$

The negative log-likelihood of the spectrogram x with respect to the parameters θ is the sum of the individual negative log-likelihood for every value x_{ij}. The individual negative log-likelihood can be expressed as

$$
-\ log\ p(x_{ij}|x_{<ij};\theta_{ij}) = -log \sum_{k=1}^{K} \pi_{ijk} N(x_{ij}; \mu_{ijk}, \sigma_{ijk}). \qquad (3.9)
$$

The implementation of this formula can have underflow problems because of the product of probabilities, i.e. $\pi_{ijk} N(x_{ij}; \mu_{ijk}, \sigma_{ijk})$. We transform the probabilities into the log domain making use of the identity $exp(log(a)) = a$:

$$
\begin{aligned}
-log \sum_{k=1}^{K} \pi_{ijk} N(x_{ij}; \mu_{ijk}, \sigma_{ijk}) &= -log \sum_{k=1}^{K} exp(log\ \pi_{ijk} N(x_{ij}; \mu_{ijk}, \sigma_{ijk})) && (3.10)\\
&= -log \sum_{k=1}^{K} exp(log\ \pi_{ijk} + log\ N(x_{ij}; \mu_{ijk}, \sigma_{ijk})) && (3.11)
\end{aligned}
$$

This formula is the one used to calculate the loss function, using the log probabilities. The implementation makes use of the log-sum-exp trick for better numerical stability [5] [6].

3.1.3 Generating a Spectrogram from the Probabilistic Model

We have shown the definition of the model and what is used as loss to train the neural network f. Assuming the neural network has been trained and can generate the maximum-likelihood estimate for the parameters θ_{ij}, to calculate the value x_{ij} it needs to sample from the Gaussian Mixture Model with parameters $\theta_{ij} = \{\hat{\mu}_{ijk}, \hat{\sigma}_{ijk}, \hat{\pi}_{ijk}\}_{k=1}^{K}$. This is done by

- sampling L from a Categorical Distribution parameterized by the mixing coefficients $\pi_{ij} = \{\pi_{ijk}|k \in [1, K]\}$ and
- sampling x_{ij} from a Gaussian Distribution parameterized by μ_{ijL} and σ_{ijL}.

The process to generate a spectrogram x is to generate it value by value x_{ij}, starting from an empty spectrogram and adding values to the spectrogram by repeatedly estimating the parameters θ_{ij} from the context $x_{<ij}$ as it grows. See fig. 3.1.

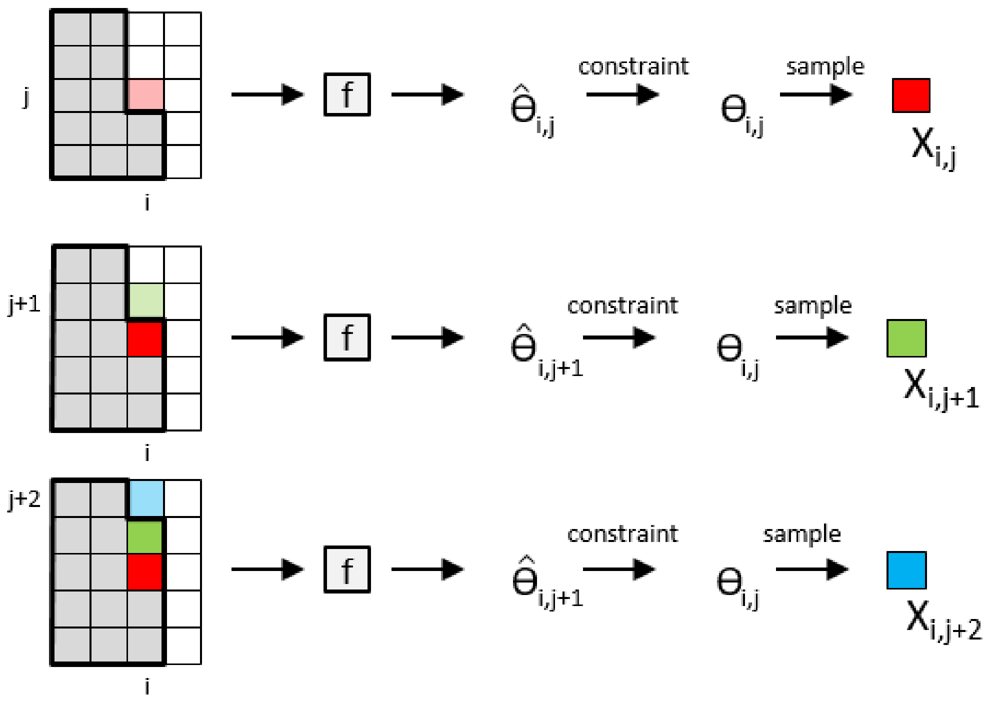


Figure 3.1: Process to generate a spectrogram unconditionally value by value. The values inside the bold line are the context $x_{<ij}$

3.2 Network Architecture

This section describes the architecture of the neural network f with weights ψ used to ouput the parameters of the Gaussian Mixture Model θ_{ij} from the context $x_{<ij}$. This network is composed of tiers, in this simpler case a single tier, and every tier is composed of layers.

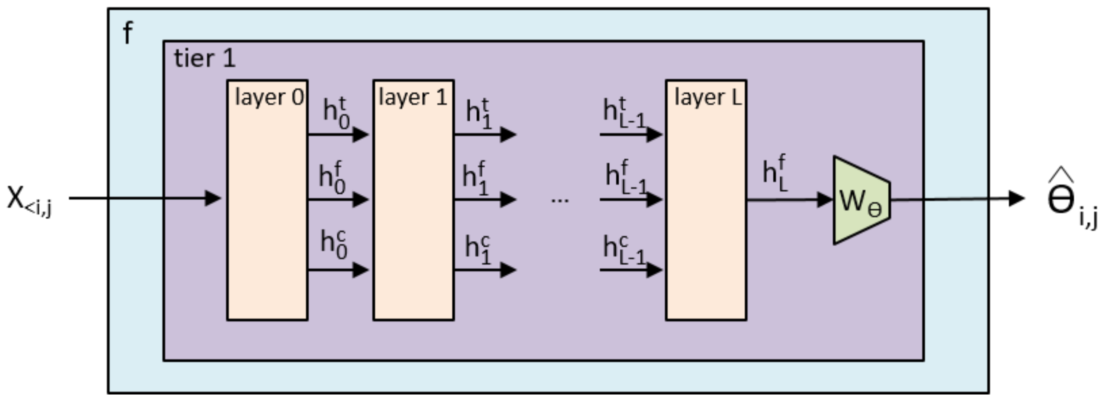


Figure 3.2: Architecture of a single-tier model and how it generates the unconstrained parameters $\hat{\theta}_{i,j}$ from the context $x_{<i,j}$.

Every layer is composed of stacks whose objective is to extract features from different segments of the input to summarize the full context x_{ij}.

There are three stacks:

- **The time-delayed stack which computes features from the previous frames** $x_{<i,*}$.
- **The frequency-delayed stack which computes features from the elements within a frame** $x_{i,<j}$ **and the features from the time-delayed stack**.
- **The (optional) centralized stack which computes information from the previous frames** $x_{<i,*}$ **but taking an entire frame as input**.

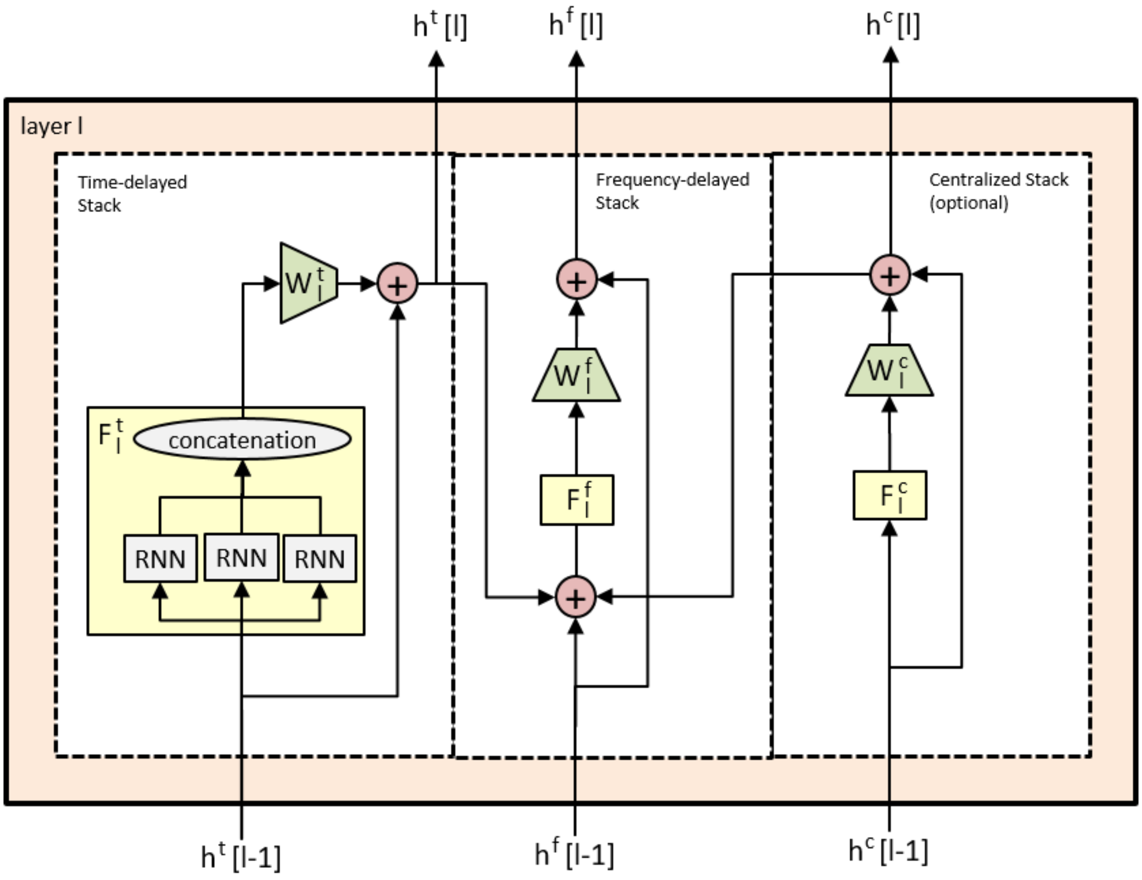


Figure 3.3: Computation graph for a single layer of the network with the three stacks of computation.

3.2.1 Time-Delayed Stack

The output of the time-delayed stack of the first layer $h^t[0]$ is computed as a linear transformation of the context $x_{<ij}$. **However, to ensure that the output** $h^t_{ij}[0]$ **is only a function of the previous frames, the input is shifted backwards one step in time**:

$$h^t_{ij}[0] = W^t_0 x_{i-1,j}. \tag{3.12}$$

The output of the first frame $h^t_{0j}[0]$ should use the information of previous frames which is impossible. The paper does not specify what to do in this situation so we use a frame filled with zeros. Other option can be to use the average of the initial frame of all the items in the dataset.

The time-delayed stack of any other layer l uses a multidimensional RNN $\mathcal{F}_l^t$, which is composed of three one-dimension RNN: one that runs forwards along time axis, one that runs forwards along the frequency axis and one that runs backwards along the frequency axis. The output of the multidimensional RNN is the concatenation of the hidden states of the three one-dimension RNN. The output of a layer l of the time-delayed stack, $h^t[l]$, is a linear transformation of the output of the multidimensional RNN plus a residual connection from the output of the time-delayed stack of the previous layer, $h^t[l-1]$. The input to the multidimensional RNN is also the output of the time delayed stack of the previous layer.

$$h_{ij}^t[l] = W_l^t \mathcal{F}_l^t(h^t[l-1])_{ij} + h_{ij}^t[l-1] \tag{3.13}$$

3.2.2 Frequency-Delayed Stack

The output of the frequency-delayed stack of the first layer $h^f[0]$ is computed as a linear transformation of the context $x_{<ij}$. **However, to ensure that the output $h_{ij}^f[0]$ is only a function of the elements in the context $x_{<ij}$, the input is shifted backwards one step in the frequency axis**:

$$h_{ij}^f[0] = W_0^f x_{i,j-1}. \tag{3.14}$$

The output of the lowest frequency slice $h_{i0}^f[0]$ should use information of the previous frequency slice which is impossible. Similar to the time-delayed stack, we use a frequency slice of zeros.

The frequency-delayed stack of any other layer l uses a one-dimensional RNN $\mathcal{F}_l^f$ that runs forward along the frequency axis, but it is also conditioned on the outputs of the time-delayed stack of the same layer. This allows to use the full two-dimensional context $x_{<ij}$. The output of a layer l of the frequency delayed stack, $h^f[l]$, is a linear transformation of the output of the one-dimensional RNN plus a residual connection from the output of the frequency-delayed stack of the previous layer $h^f[l-1]$. The input to the one-dimensional RNN is the sum of the output of the frequency-delayed stack of the previous layer $h^f[l-1]$ and the output of the time-delayed stack of the current layer $h^f[l]$.

$$h_{ij}^f[l] = W_l^f \mathcal{F}_l^f(h^f[l-1], h^t[l])_{ij} + h_{ij}^f[l-1] \tag{3.15}$$

At the final layer L a linear transformation is applied to the output of the frequency-delayed stack to produce the unconstrained parameters:

$$\hat{\theta}_{ij} = W_\theta h_{ij}^f[L]. \tag{3.16}$$

3.2.3 Centralized Stack

The recurrent state of the time-delayed stack (output of the time-delayed stack) is distributed across an array of RNN cells which tile the frequency axis. To allow for a more centralized representation, the centralized stack takes an entire frame as input $x_{i,*}$.

The output of the centralized stack of the first layer $h^f[0]$ is computed as a linear transformation of the previous frame. **However, to ensure that the output** $h_i^c[0]$ **is only a function of the frames which lie in the context** $x_{<ij}$**, the input is shifted backwards on step in time**:

$$h_i^c[0] = W_0^c x_{i-1,*}. \tag{3.17}$$

The output of the first frame $h_0^c[0]$ should use the information of the previous frame which is impossible. Similar to the other two stacks, we use a frame filled with zeros.

The centralized stack of any other layer l uses a one-dimensional RNN $\mathcal{F}_l^c$ that runs along the time axis, taking an entire frame as input. The output of a layer l of the centralized stack is a linear transformation of the output of the one-dimensional RNN plus a residual connection from the output of the centralized stack of the previous layer $h^c[l-1]$:

$$h_i^c[l] = W_l^c \mathcal{F}_l^c(h^c[l-1])_i + h_i^c[l-1] \tag{3.18}$$

This is an optional stack, if it is used, the output of the centralized stack is input to the frequency delayed stack at each layer, modifying eq. (3.15) as

$$h^f[l] = W_l^f \mathcal{F}_l^f(h^f[l-1], h^t[l], h^c[l]) + h^f[l-1] \tag{3.19}$$

where the three inputs to the one-dimensional RNN $\mathcal{F}_l^f$ are simply summed.

4
MelNet: Multi-Tier

4.1 Multiscale Modelling

4.1.1 Definition of Multiscale Modelling

In the single tier approach described before, the autoregressive ordering is a simple time-major ordering which proceeds frame by frame from low to high frequencies. However, a spectrogram has a high number of dimensions. This hinders learning the global structure of spectrograms, because autoregressive models tend to learn the local structure. **To solve this, MelNet uses a multiscale approach and generates the spectrograms in a coarse-to-fine order**.

The elements of a spectrogram x are partitioned into G tiers $x = (x^1, ..., x^G)$. We define $x^{<g}$ as the union of all tiers which preceed x^g, i.e $x^{<g} = (x^1, ..., x^{g-1})$. The joint distribution of a spectrogram x is then factorized over the tiers:

$$p(x; \psi) = \prod_g p(x^g | x^{<g}; \theta^g = f(\psi^g)) \tag{4.1}$$

where each tier is conditioned on the previous tiers and the distribution of each tier is factorized element-by-element as described in eq. (3.2). We have modified eq. (4.1) with respect to the equation from MelNet, to make more explicit that the distribution of each tier $p(x^g | x^{<g}; \theta^g = f(\psi^g))$ is parametrized by θ^g which is the output of a network f with weights ψ, and also make it more coherent with eq. (3.2). This also helps to indicate that each tier is modelled by a separate network.

The distribution of each tier is factorized as

$$p(x^g | x^{<g}; \theta^g = f(\psi^g)) = \prod_i \prod_j p(x^g_{ij} | x^g_{<ij}, x^{<g}; \theta^g_{ij} = f(x^g_{ij}, x^{<g}; \psi^g)) \tag{4.2}$$

where the parameters θ^g_{ij} are governed by the output of a neural network f with weights ψ^g as a function of the context $x^g_{<ij}$ but also of the spectrogram generated at previous tiers $x^{<g}$. See fig. 4.1.

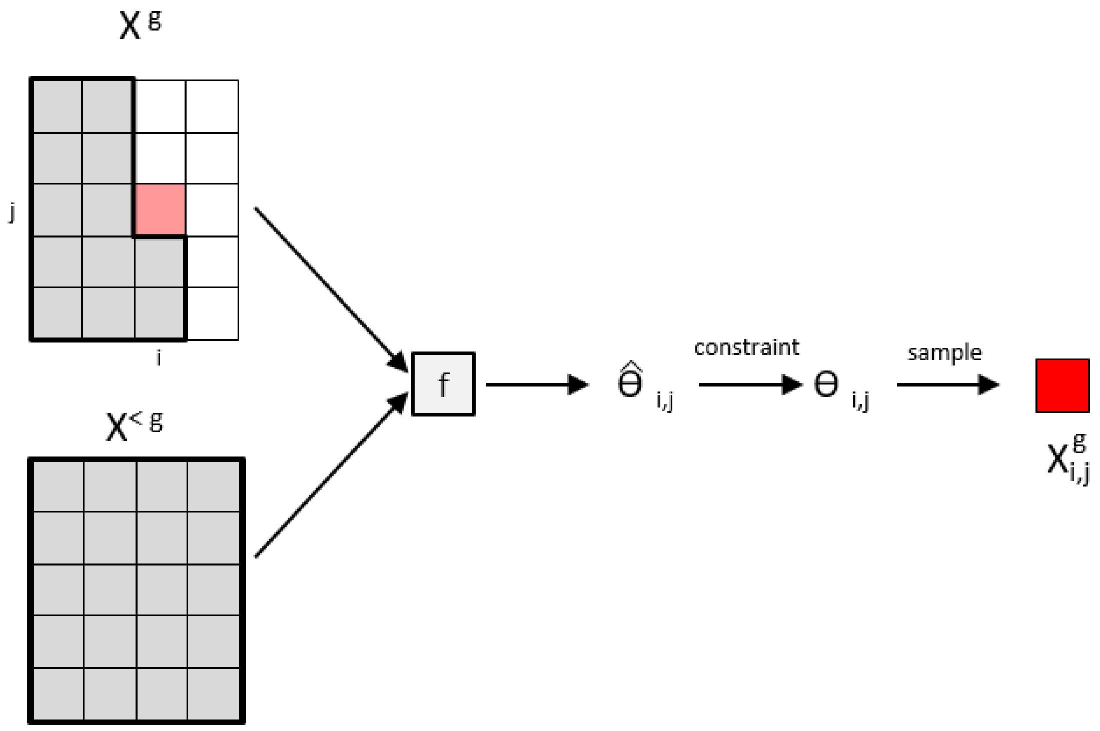


Figure 4.1: Process to generate the spectrogram corresponding to x^g value by value. The values inside the bold line in x^g correspond to the context $x^g_{<i,j}$.

4.1.2 Training and sampling

The spectrogram is divided into tiers by partitioning it into alternating rows along either the time or the frequency axis. **The paper defines the function *split* which partitions an input into even and odd rows along a given axis, depending on the tier. In the first step, the spectrogram x, or equivalently x^{G+1} is split into even rows, assigned to x^G, and odd rows, assigned to $x^{<G}$. Subsequent tiers are defined recursively:**

$$x^g, x^{<g} = split(x^{<g+1}) \tag{4.3}$$

At each step, MelNet models the distribution $p(x^g|x^{<g}; \theta^g = f(\psi^g))$. Each distribution is conditioned on $x^{<g}$, but the parameters of the distribution of each tier θ^g are modelled by a separate and independent network $f(\psi^g)$. This allows to train each tier independently.

To generate a spectrogram x we start generating the initial tier, x^1, which is unconditionally sampled from $p(x^1; \theta^1 = f(\psi^1))$ as shown in section 3.1.3. After that, we iteratively generate generate x^g conditioned on $x^{<g}$, sampling from $p(x^g|x^{<g}; \theta^g = f(\psi^g))$. At each tier, x^g is interleaved with $x^{<g}$ to generate $x^{<g+1}$:

$$x^{<g+1} = interleave(x^g, x^{<g}) \tag{4.4}$$

split and *interleave* are inverse functions, and x^g and $x^{<g}$ have the same dimensions.

4.2 Network Architecture

This section describes the architecture of the different neural network f used to output the parameters of the Gaussian Mixture Model of every tier θ^g.

The first tier network has the same structure as the network of the single tier model explained in chapter 3. The inputs and outputs are the same, which allows to generate x^1 unconditionally from the modelled distribution $p(x^1; \theta^1 = f(\psi^1))$.

The other tiers have a similar architecture to the network of the single tier model, but they need a mechanism to add the information from preceding tiers. This feature extraction network computes features from $x^{<g}$ that are used to condition the generation of x^g.

The feature extraction extraction uses a multidimensional RNN $\mathcal{F}^{fext}$, which is composed of two one-dimensional RNN which run bidirectionally along slices of both axes in x^g. Effectively there is four one-dimensional unidirectional RNN, one for each direction of the spectrogram. The output of the multidimensional RNN, $z^g = \mathcal{F}^{fext}(x^{<g})$, is the concatenation of hidden states of the one-dimensional RNN.

These features are used to condition the generation of x^g by using them as input to the first layer of the tier modifying eq. (3.12) and eq. (3.14) to:

$$h^t_{ij}[0] = W^t_0 x^g_{i-1,j} + W^t_z z^g_{ij} \tag{4.5}$$

$$h^f_{ij}[0] = W^f_0 x^g_{i,j-1} + W^f_z z^g_{ij} \tag{4.6}$$

Because x^g and $x^{<g}$ have the same dimensions, the implementation of the conditioning mechanism can be implemented straightforwardly.

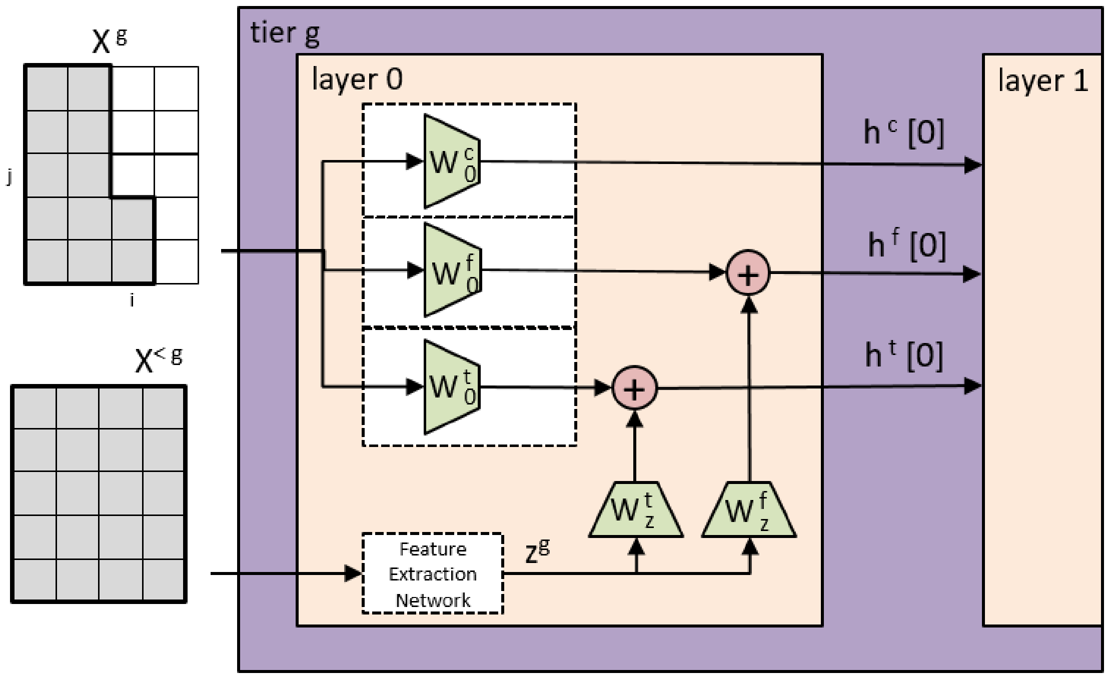


Figure 4.2: Computation graph for the layer 0 of a tier g ($g > 1$) where the hidden state of the layer 0 of the time-delayed stack $h^t[0]$ and frequency-delayed stack $h^f[0]$ are conditioned on the features z^g from x^g.

5 Implementation of MelNet

5.1 Implementation of the Model

This project has been implemented using the PyTorch framework for Python. MelNet is a modular model that contains tiers composed of layers composed of stacks of computation. PyTorch offers the class *torch.nn.Module* to create neural network modules, which can also contain other modules. This eases implementation because there is a one-to-one mapping between the modules explained in the paper [2] and the modules in PyTorch. The loss function of MelNet is also implemented as a PyTorch module.

5.2 Implementation of other utilities

Apart from the model, there are other utilities needed to train MelNet. For instance, to consume data in an efficient way the implementation follows PyTorch recommendations and wraps the data with *torch.utils.data.Dataset* to be consumed by *torch.utils.data.DataLoader*. There are some difficulties when working with a batchsize greater than one, because PyTorch expects all the items of a batch to have the same dimensions. In our case items have different dimensions, depending on the length of the waveform, so we opt to fill with zeros the waveforms so they have the same dimensions, although the paper does not specify how they do it. It is important not to score the model on this fake data, to make sure it does not affect optimisation.

Other utilities include loading the parameters from YAML files, which helps to maintain the parameters of the models organised and correctly labeled. This implementation runs in a conda environment inside a Docker container for a maximum control of the environment, also improving the portability and reproducibility.

5.3 Training

We train each tier separately, instead of training all the tiers of MelNet at the same time. For a single iteration of the training loop of the tier g, we recursively apply the *split* function to a real spectrogram y to get $y^{<g}$ and y^{g}. We use $y^{<g}$ as the condition to generate the parameters $\theta^{g} = f(y^{<g}, \psi^{g})$. Finally, we compute the error, $error(\theta^{g}, y^{g})$, calculate the gradients of the model and update the weights of the tier.

The initial idea was to train models on all the architectures of MelNet described in the work of Vasquez and Lewis [2] (Table 1) for unconditional speech and compare against each other on how they perform on the new podcast dataset. The architecture is defined by the number of tiers, the number of layers of each tier, and the hidden size, which controls the RNN hidden states size.

However, using the same parameters was impossible because of the limitations of the VRAM of the GPU machine. Tweaking the parameters showed that hidden size was the parameter that most affected model size. Using the architecture for unconditional speech for Blizzard (Table 1 [2]) we had to reduce the hidden size from 512 to 16 to fit the model in memory. This means that the model being trained is roughly 80 times smaller that the one described in the paper. In addition, batch size in the work of Vasquez and Lewis [2] varies between 16 and 128, but the maximum batchsize that allowed the model to fit into memory was 2.

To mitigate the model size problem, PyTorch offers checkpointing, a feature that helps to reduce the memory footprint of the model by trading compute for memory. In general, PyTorch stores all the intermediate activations in the forward pass, to later calculate the gradients in the backward pass. By checkpointing a *torch.nn.Module*, the activations of that module are not saved, and instead recomputed in the backward pass. The modules that implement checkpointing are the layers, and is the activations between layers that are stored. Because of how checkpointing is implemented in PyTorch, adding it to the implementation requires to add a wrapper around the modules that are checkpointed. Using this trick, hidden size was increased from 16 to 200. Other approaches that could help increase the size of the model in training is to use distributed training, but this was outside the scope of the project.

To increase batch size, gradient accumulation was used [7]. The idea of gradient accumulation is to perform multiple forward and backward passes with the same model weights, accumulating the gradients, to update the weights at once. In theory and as an example, using a real batch size of 2 and with 16 gradient accumulation steps, makes the effective batch size 32. Using this trick, the effective batch size for training can be set to any arbitrary size.

After adding checkpointing and gradient accumulation to the model, training would still fail because the parameters θ^g being calculated as the output of the tier were too big. The solution to this was normalizing the data using the same method as the Tacotron implementation [8]. This normalization includes mean and variance standatdisation and clamping the values between 0 and 1, so there is a loss which is audible when a normalized spectrogram is denormalized and transformed back to an audio waveform.

Other solution would be to use batch normalization, as it is done in Tacotron2 [9], but this can not be used together with gradient accumulation. Even adding normalization, the gradients of the model were exploding, possibly because of the use of RNN in the model, so we added gradient clipping [10], similar to the reference implementation of Tacotron2 [11].

6 Experiments and Results

	MelNet	fig. 6.1	fig. 6.2	fig. 6.3	fig. 6.4a	fig. 6.4b
Dataset	Blizzard	Podcast	LJSpeech	LJSpeech	LJSpeech	LJSpeech
Tiers	6	6	6	5	6	6
Layers (Initial Tier)	12	12	0	0	14	8,10,12,14,16
Layers (Upsampling Tiers)	5-4-3-2-2	5-4-3-2-2	7-6-5-4-4	6-5-4-4	7-6-5-4-4	7-6-5-4-4
Hidden Size	512	200	200	200	16,32,64,128,200	64
GMM Mixture Components	10	10	10	10	10	10
Effective Batch Size	32	8	16	16	16	8
Real Batch Size	32	1	1	1	1	1
Accumulation Steps	1	8	16	16	16	8
Sample Rate (Hz)	22050	22050	22050	22050	22050	22050
STFT Hop Size	256	256	256	256	256	256
SFTT Window Size	6*256	6*256	6*256	6*256	6*256	6*256
Learning Rate	10^{-4}	10^{-5}	$4 * 10^{-6}$	$4 * 10^{-6}$	$4 * 10^{-6}$	$4 * 10^{-6}$
Momentum	0.9	0.9	0.9	0.9	0.9	0.9

Table 6.1: Hyperparameters used in the experiments. If the number of layers is zero, it means that the tier was not used for training or synthesis. If the numbers are in a sequence separated by commas, it means that is the same architecture changing that hyperparameter. All the models of the experiments were trained on a machine with 16GB of RAM and a GPU GeForce RTX 2080 with 8GB of VRAM.

6.1 Initial Experiment

The initial architecture trained in the podcast dataset was the one used by Vasquez and Lewis for unconditional speech for Blizzard ([2] Table 1) but with a hidden size of 200 instead of 512 due to memory constraints. Using this architecture, we obtained the results showed in fig. 6.1, but we were not able to reproduce the results presented by the paper.
To illustrate the architectures of the trained models in a compact way, from now on we will follow this pattern: *d(dataset)_t(number of tiers)_l[number of layers]_hd(hidden size)_gmm(GMM Mixture Components).*

With the initial architecture as reference, we trained other models varying the architecture parameters to see how this affected the spectrograms being generated. However, none of the models was generating realistic spectrograms. The upsampling tiers appeared to be able to add detail to the spectrogram generated by previous tiers, but the initial tier was not able to dictate a coherent high-level structure.

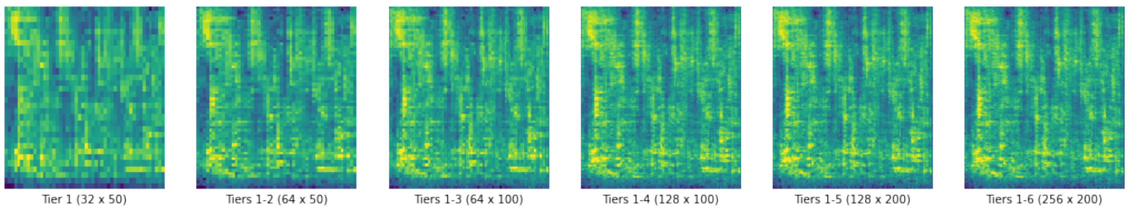


Figure 6.1: Spectrogram viewed at different stages generated by the initial architecture. Architecture: dpodcast_t6_l12.5.4.3.2.2_hd200_gmm10

6.2 Experiment with Upsampling Layers only

As a means to see how much impact the initial tier has in the final spectrogram, we modified the algorithm for synthesis. In the normal synthesis algorithm, the first tier generates unconditionally a low-resolution spectrogram and the upsampling tiers add detail. To separate the impact of the first tier, we used an item from the dataset as the output of the first tier (by applying the *split* function until the shape is the same as it would be if it was generated by the first tier) and made the upsampling tiers to add detail to this real low-resolution spectrogram. Also, we used the LJSpeech dataset [12], instead of the podcast dataset, because it contains more data and it is a widely-used dataset in the speech generation field, i.e. it is used to train the open-source implementation of Tacotron2 [11].

As it can be seen in fig. 6.2, using a real low-resolution spectrogram as the output of the first tier makes the final spectrogram closer to a realistic spectrogram, with better structure. The audio recovered from this spectrogram is still understandable, but it contains noise and the voice of the person sounds as if it had been filtered. This could be because the upsampling layers have not totally learn how to add detail, but the normalization to the data could affect too, since it is a lossy transformation.

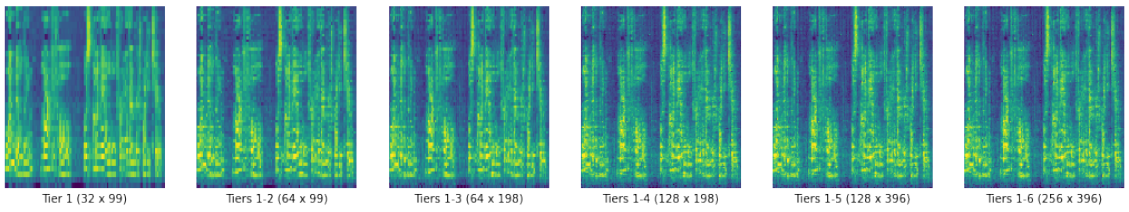


Figure 6.2: Spectrogram viewed at different stages generated using a real low-resolution spectrogram. Architecture: dljspeech_t6_l0.7.6.5.4.4_hd200_gmm10. The first tier does not have layers because it was not used.

To see if the noise was introduced by the upsample layers, we generated another spectrogram using the same real low-resolution spectrogram as the output of the first tier, but with a model containing 5 tiers instead of 6. Observing fig. 6.3, we see that the final spectrogram is slightly less blurry, which indicates that the upsampling layers add noise when adding more detail to the spectrograms.

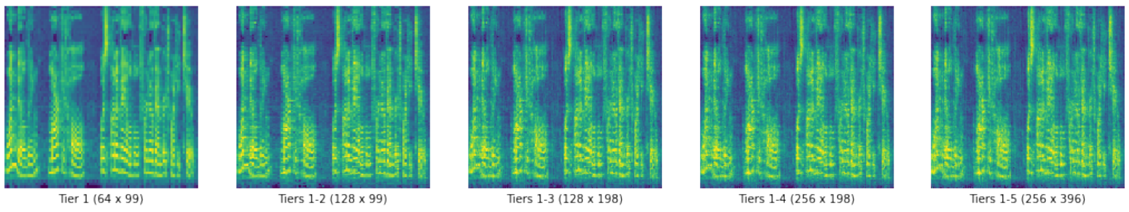
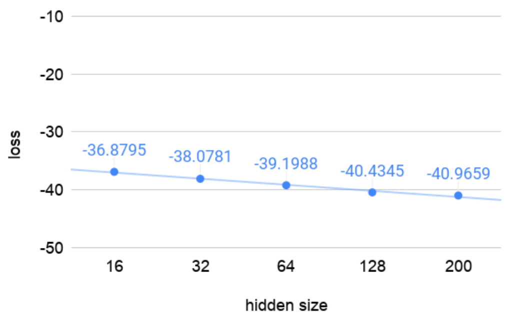


Figure 6.3: Spectrogram viewed at different stages generated using a real low-resolution spectrogram. Architecture: dljspeech_t6_l0.6.5.4.4_hd200_gmm10. The first tier does not have layers because it was not used.

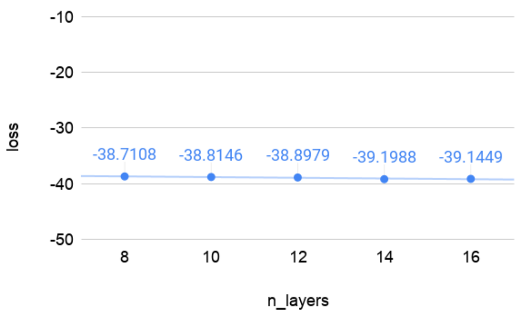
6.3 Experiments with First Tier

Knowing that the first tier is the the most important because it dictates the high-level structure, we trained different models varying the hidden size and the number of layers to see how this impacts loss, to try to find the best architecture.

Hidden size is the parameter that most affects the model size. In fig. 6.4a, we show how different hidden sizes impact the loss of the first tier. This shows us that it is an important parameter and not being able to train a model with a hidden size of 512, as Vasquez and Lewis [2], could be one of the reasons why our model is underperforming. The other parameter that affects model size is the number of layers of each tier. In this case, from fig. 6.4b we can conclude that increasing the layer size appears to have a lesser impact on the loss.



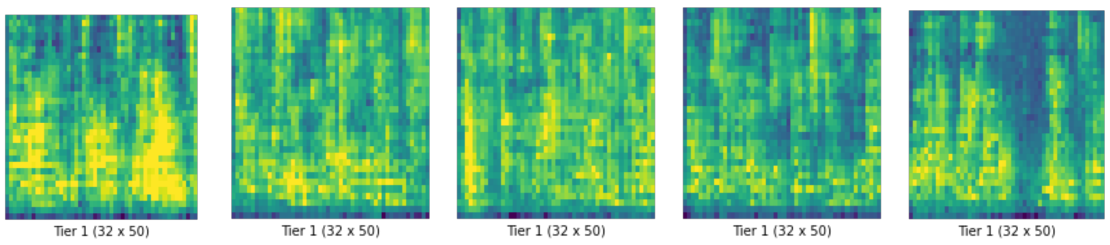
(a) First tier: hidden size vs. loss. Architecture: dljspeech_t6_l14.5.4.3.2.2_hdX_gmm10.



(b) First tier: number of layers vs. loss. Architecture: dljspeech_t6_lX.5.4.3.2.2_hd64_gmm10.

Figure 6.4: The loss is the average loss of a frame over the evaluation dataset.

To have a better understanding of the relation between the loss of a tier and the spectrogram it generates, we generated the first tier spectrograms from the models in fig. 6.4a. In fig. 6.5 we see that the bigger the hidden size is, and consequently the smaller the loss, the better structure of the spectrogram generated by the first tier.



(a) hd = 16 **(b)** hd = 32 **(c)** hd = 64 **(d)** hd = 128 **(e)** hd = 200

Figure 6.5: Spectrogram generated by the first tier with different hidden size. Architecture: dljspeech_t6_l14.5.4.3.2.2_hdX_gmm10.

6.4 Discussion

The previous experiments showed us that the first tier is the most important tier to generate realistic spectrograms for speech, because it dictates the high-level structure. After that, we saw that bigger models, especially with bigger hidden sizes, produce better spectrograms. We compared the effect that different hidden sizes had in spectrograms generated by the first tier, and how the models with bigger hidden sizes could learn to produce better initial structure.

From these results we can conclude that the memory constraints that prevent to train a model with a hidden size greater than 200 is one of the reasons for the underperformance of this implementation. We should also note that Vasquez and Lewis [2] present different architectures for different models, which could indicate that they carried out an extensive search to find the best parameters. Due to the limitations of the available machines to replicate MelNet, we could not afford to do such an extensive search which makes difficult to get comparable results. This is especially true when it is considered the big number of parameters this model has, not only to control the architecture, but also the audio transformation parameters. The duration of the training of each model could be another reason, since we had to do a trade off between the duration of the training of a single model and the number of different architectures we could try.

With all this information, we trained the biggest model we could, since we have seen that this has an impact on the quality of the spectrograms generated. The model was trained on LJSpeech dataset [12] for longer than the other models and the results were better. In fig. 6.6 we can see a spectrogram generated by this model. It is a promising result because the spectrogram contains structures closer to spectrograms of real speech, especially on the first part of the example.

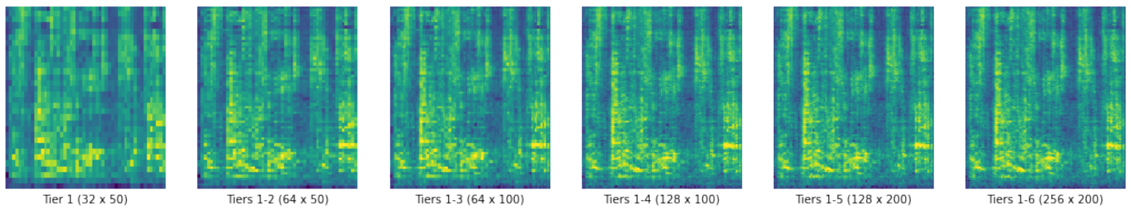


Figure 6.6: Spectrogram viewed at different stages.

Dataset	LJSpeech
Tiers	6
Layers (Initial Tier)	12
Layers (Upsampling Tiers)	7-6-5-4-4
Hidden Size	200
GMM Mixture Components	10
Effective Batch Size	8
Real Batch Size	1
Accumulation Steps	8
Sample Rate (Hz)	22050
STFT Hop Size	256
SFTT Window Size	6*256
Learning Rate	$4 * 10^{-6}$
Momentum	0.9

Table 6.2: Hyperparameters used in fig. 6.6.

The files for the spectrograms, audio waveforms and wav files for the examples of these experiments can be found in:

`https://github.com/jgarciapueyo/MelNet-SpeechGeneration/tree/master/results`.

7 Conclusion

We have implemented MelNet (S. Vasquez and M. Lewis, "Melnet: A generative model for audio in the frequency domain," *arXiv preprint arXiv:1906.01083*, 2019), a probabilistic model for speech synthesis that works on spectrograms, for which there was no de-facto standard implementation. We applied MelNet to the task of unconditional speech training it on a dataset containing audio from a dialogue-based podcast. In this report, we expanded the explanations of the paper with respect to the architecture of MelNet, adding information about techniques used to adapt this model to fit into a machine with memory constraints and modifications to the training process.

In our experiments, we found that the hidden size of a model plays an important role in generating spectrograms with better structure, and that the first tier is the most important one because it dictates the structure of the final spectrogram. Being limited to use models with small hidden size, comparing it to the architectures showed in the paper, has restricted the ability to produce human-like spech (or babbling in the case of unconditional speech generation). However, we showed promising results in spectrograms generated by a bigger model that contained structures close to the ones in spectrograms of real speech.

Bibliography

[1] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," 2016.

[2] S. Vasquez and M. Lewis, "Melnet: A generative model for audio in the frequency domain," *arXiv preprint arXiv:1906.01083*, 2019.

[3] D. Griffin and J. Lim, "Signal estimation from modified short-time Fourier transform," *IEEE T. Acoust. Speech*, vol. 32, no. 2, pp. 236–243, 1984.

[4] C. M. Bishop, "Mixture density networks," Tech. Rep. `http://publications.aston.ac.uk/id/eprint/373/`NCRG/94/004, Aston University, Birmingham, UK, 1994.

[5] T. P. Mann, "Numerically stable hidden markov model implementation," 2006.

[6] O. Freifeld, "Class probabilities and the log-sum-exp trick." `https://www.cs.bgu.ac.il/~cv201/wiki.files/logsumexp_trick.pdf`, 2017.

[7] T. Wolf, "Training neural nets on larger batches: Practical tips for 1-gpu, multi-gpu & distributed setups." `https://medium.com/huggingface/training-larger-batches-practical-tips-on-1-gpu-multi-gpu-distributed-setups-e`

[8] K. Ito, "Tacotron." `https://github.com/keithito/tacotron/blob/master/util/audio.py`.

[9] J. Shen, R. Pang, R. J. Weiss, M. Schuster, N. Jaitly, Z. Yang, Z. Chen, Y. Zhang, Y. Wang, R. Skerry-Ryan, R. A. Saurous, Y. Agiomyrgiannakis, and Y. Wu, "Natural tts synthesis by conditioning wavenet on mel spectrogram predictions," 2017.

[10] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," 2012.

[11] NVIDIA, "Tacotron2." `https://github.com/NVIDIA/tacotron2`.

[12] K. Ito and L. Johnson, "The lj speech dataset." `https://keithito.com/LJ-Speech-Dataset/`, 2017.