Annotated WaveNet

Noah Fleischmann

December 2020

Abstract

The purpose of this paper is to annotate and explain DeepMind's WaveNet architecture. To do this, we divide the problem into submodules that build upon each other, first understanding and preparing our data, then constructing the architecture's key parts before building specialty classes that train- and generate from the model. After creation of these parts, we will train our stripped down version using a portion of CMU Arctic bdl, a dataset created by Carnegie Mellon University, and generating using `helloworld.wav` from Jaehun Ryu's implementation on GitHub, from which this paper is based. We will find that this version of the model can identify and imitate broad features of the given seed audio, in this case, the rhythm of speech, but is not able to fully imitate the data it is trained on.

1 Introduction

At the time of its release, WaveNet was revolutionary. Particularly with its pre-trained model, an architecture that allowed for audio-to-audio sequencing was available for specialization with relatively small amounts of time and computing power. Still nearly four years later, information on how WaveNet works in detail is thin. The aim of this paper is to fill the gap between the abstract description of the original paper and the undocumented versions one might find on GitHub.

This paper is a deep dive into a PyTorch implementation by Jaehun Ryu on github, which can be found here. A full list of the modules we will use can be found in `requirements.txt`. In particular, torchaudio, librosa, and soundfile are key workhorses that do not come standard with most environments. If following along on any Linux operating system, ffmpeg is also required as an audio backend.

2 Data

First we will start by creating our data sub-module, which will convert Wav files on disk to torch tensors, prepare and trim them, and then put them in a dataloader.

One of the key problems in dissecting this implementation is in understanding the structure of the data it takes in and returns. To put simply, WaveNet must take in sections of audio files that are longer than its receptive field, which is the length of audio it collapses to predict the next sample. The prediction targets and the output of the model are the section of audio starting at the first sample after the receptive field to the end of the file. This is because the model acts broadly as a convolutional network with a kernel size of the receptive field, passing over the audio and giving an output of size $file\_length - receptive\_field$.

This snippet below imports all of the modules we will need to create our data sub-module and defines a load function, which defers to librosa's load function to import the raw audio as a Numpy array. It throws away the extra dimension and optionally trims the file into sections of the given length.

```
import os
import librosa
import numpy as np
import torch
import torch.utils.data as d

def load(filename, sample_rate=16000, trim=2048):
    '''
    filename (string): Path to audio file
    sample_rate (int): Sample rate to import file with
    Trim (int): number of samples in each clip, 0 returns whole file

    returns:
        audio     (NP array) : shape (trim,)
          OR
        raw_audio (NP array) : shape (n,)
    '''
    # Load in the raw audio using librosa and reshape to remove extra dimension
    # Underscore is the sample rate, which we already know
    raw_audio, _ = librosa.load(filename, sr=sample_rate, mono=True)
    raw_audio = raw_audio.reshape(-1, 1)

    # If requested, trim the clips of the specified number of samples
    if trim > 0:
        audio, _ = librosa.effects.trim(raw_audio, frame_length=trim)
        return audio

    return raw_audio
```

What does this returned data look like? If we run the following snippet, we can get an idea:

```
>>> x = load('./data/helloworld.wav')
>>> x
array([[ 3.0305667e-09],
       [ 2.7823888e-10],
       [-4.6928004e-09],
       ...,
       [ 0.0000000e+00],
       [ 0.0000000e+00],
       [ 0.0000000e+00]], dtype=float32)
>>> x.shape
(1668550, 1)
```

Just as we wanted, our function returns a one dimensional NumPy array of the raw audio. However, we're not done yet. We still need to decide what our targets are, and before we can do that, we need to adjust the values of the array to something our model can work with more accurately. At the end of our process we use a softmax distribution to predict the next sample, which will perform poorly if trying to estimate for the 65536 possible values of a 16 bit audio file. To remedy will will use Mu-law companding, as recommended by the original paper, to compress the range down to 256 possible values. Unfortunately, this is a lossy compression that will reduce our bit depth down to 8. The following shows the formula to achieve this:

$$f(x_t) = \text{sign}(x_t)\frac{\ln(1+\mu|x_t|)}{\ln(1+\mu)}$$

```
def mu_law_encode(waveform, channels=256):
    '''
    Quantize waveform amplitudes
    '''
    # create an array of equally spaced points between -1 and 1
    lin_space = np.linspace(-1, 1, channels)
    # Apply the formula
    channels = float(channels)
    quantized = np.sign(waveform) * np.log(1 + (channels - 1) * np.abs(waveform)) / np.log(channels)
    # Discretize and return
    return np.digitize(quantized, lin_space) - 1
```

```
>>> x = load('./data/helloworld.wav')
>>> mu_law_encode(x)
array([[127],
       [127],
       [127],
       ...,
       [127],
       [127],
       [127]])
```

After quantizing, we create a one-hot encoding to convert a complicated regression task into a simpler classification task.

```
def one_hot_encode(data, channels=256):
    '''
    Creates a one-hot encoding of the 1D input data
    returns np array of size (data.size, channels)
    '''
    one_hot = np.zeros((data.size, channels), dtype=float)
    # Make the value one at the the column corresponding to the row value
    one_hot[np.arange(data.size), data.ravel()] = 1

    return one_hot
```

Finally, we need functions to convert our encoding and companding back to raw audio. The formula to convert our Mu law companding back to floats is the following:

$$f^{-1}(y_t) = \text{sign}(y_t)(1/\mu)((1+\mu)^{|y_t|} - 1$$

```
def one_hot_decode(data, axis=1):
    '''
    Decodes a one-hot encoding
    returns a 1D array
    '''
    return np.argmax(data, axis=axis)

def mu_law_decode(data, channels=256):
    '''
    Recovers the waveform from discretized data
    '''
    channels = float(channels)
    # re-centers the data around 0
    exp = -1 + (data / channels) *2.0
    return np.sign(exp) * (np.exp(np.abs(exp) * np.log(channels)) - 1) / (channels - 1)
```

Now that we've got all of our importing and translation functions, we can create a custom dataset and dataloader. To create a dataset, we need to define two methods: one that gets the item at a given index, and one that returns the length of the dataset. Ours will take in the path to a directory of audio files and for each index return an encoded version of the whole file, which our dataloader will divide up into pieces for analysis.

Our model won't use this directly, but our dataloader will use it behind the scenes[1]. To create a dataloader, we need to define a function that collates a given file into a generator of trimmed sections, shown below. This function is called while looping through the dataloader, which automatically prepends a batch dimension. We have two hidden helper methods not shown: one that calculates the sample size and one that converts our NumPy array to a torch tensor using the GPU if it is available.

```
def _collate_fn(self, audio):
    # First pad the audio along the timestep dimension to make sure its longer
    # than the required sample size, it gets cut down later
    audio = np.pad(audio, [[0, 0], [self.receptive_field, 0], [0, 0]], 'constant')

    if self.sample_size:
        # If a sample size greater than 0 is given, break the file into
        # sections that are that long
        sample_size = self._sample_size(audio)

        # This while loop breaks the waveform up into chunks and returns them
        # one at a time
        while sample_size > self.receptive_field:
            inputs = audio[:, :sample_size, :]
            targets = audio[:, self.receptive_field:sample_size, :]

            yield self._variable(inputs), self._variable(one_hot_decode(targets, 2))

            audio = audio[:, sample_size - self.receptive_field:, :]
            sample_size = self._sample_size(audio)
    else:
        # if the sample size is zero or None or False, return the whole file
        targets = audio[:, self.receiptive_field:, :]
        return self._variable(audio), self._variable(one_hot_decode(targets, 2))
```

Now that we have our dataloader, we can package all of this into a sub-module that we can import in our next adventure.

3 Model Parts

Our second sub-module will define the different layer types we see in our model. Figure 1, taken from the original WaveNet paper, shows the model relies on three key pieces: an input convolution, a stack of residual blocks, and an output network. Each block in the residual stack can be further broken down into a dilated convolution, a non linearity convolution, and a residual connection. The dilated convolution is the key ingredient in this model, so we will define that first. Thankfully, in the true spirit of PyTorch, the built-in module will do the complicated part for us using the dilation keyword argument. All we have to do is feed it the distance between convoluted samples and the number of residual channels at input and output. To define the forward pass, we defer to the convolution layer.

[1]The code for the `CustomDataset` class is available in Appendix A

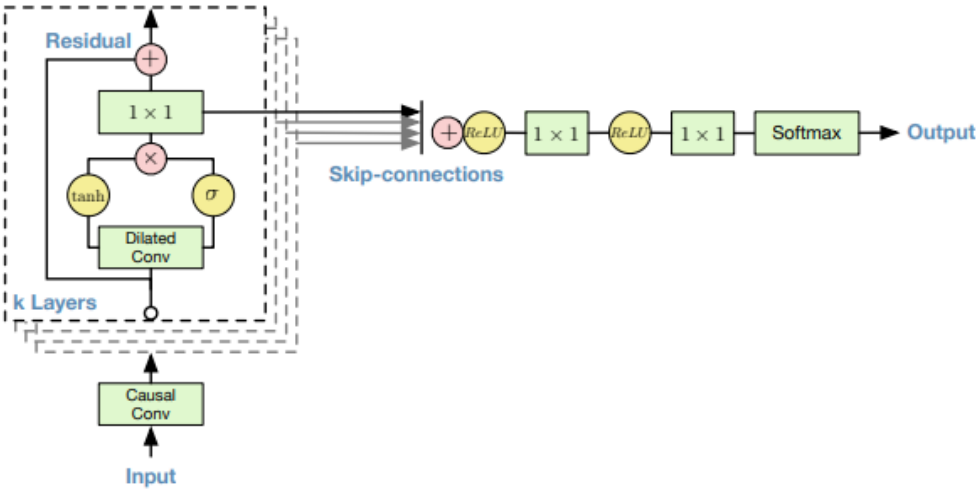

Figure 1: The structure of WaveNet's architecture.

```
import torch
from torch import nn

class DilatedCausalConv(nn.Module):
    def __init__(self, channels, dilation=1):
        super(DilatedCausalConv, self).__init__()

        self.conv = nn.Conv1d(channels, channels,
                              kernel_size = 2, stride=1,
                              dilation=dilation,
                              padding=0,
                              bias=False)

    def forward(self, x):
        output = self.conv(x)
        return output
```

Next we will take on the more complicated task of defining our residual blocks. At this point, our implementation will diverge from the original paper. Rather than having one convolutional layer that outputs to both the residual and skip connections, each will get their own independently parameterized convolutional layer to allow for greater control in training. With this in mind, our residual block has five parts: A dilated convolution layer, tanh filter and sigmoid gate nonlinearities, an two 1×1 convolutional layers for the residual and skip connections.

```
class ResidualBlock(nn.Module):
    def __init__(self, residual_channels, skip_channels, dilation):
        super(ResidualBlock, self).__init__()

        self.dilated = DilatedCausalConv(residual_channels, dilation=dilation)
        self.residual_conv = nn.Conv1d(residual_channels, residual_channels, 1)
        self.skip_conv = nn.Conv1d(residual_channels, skip_channels, 1)

        self.filter_tanh = nn.Tanh()
        self.gate_sig = nn.Sigmoid()
```

```
    def forward(self, x, skip_size):
        output = self.dilated(x)
        # Activated hidden state:
        filter = self.filter_tanh(output)
        gate = self.gate_sig(output)
        activated = gate * filter
        # Calculate the residual connection
        output = self.residual_conv(activated)
        cut_input = x[:, :, -output.shape[2]:]
        output += cut_input
        #Calculate and cut the skip connection to size
        skip = self.skip_conv(activated)
        skip = skip[:, :, -skip_size:]

        return output, skip
```

At the forward pass after running our activated hidden state through the residual convolution, we cut the input to the length of the output and add. After running our hidden state through the skip convolution, we cut to the desired output size. Now that we have a definition for a single block, we can create a class that stacks them for modularity. At initialization, our residual stack class takes in the number of blocks and the number of layers per block, and at the forward pass, it recursively defines the input to the next model and keeps track of the skip connections before stacking them at the return.

```
class BlockStack(nn.Module):
    def __init__(self, layer_size, stack_size, residual_channels, skip_channels):
        super(BlockStack, self).__init__()

        self.layer_size = layer_size
        self.stack_size = stack_size

        self.blocks = self.stack_blocks(residual_channels, skip_channels)

    def _dilations(self):
        dilations = []
        for stack in range(0, self.stack_size):
            for layer in range(0, self.layer_size):
                dilations.append(2 ** layer)
        return dilations

    def stack_blocks(self, residual_channels, skip_channels):
        return [ResidualBlock(residual_channels, skip_channels, d) for d in self._dilations()]

    def forward(self, x, skip_size):
        output = x
        skips = []
        for i in range(len(self.blocks)):
            block = self.blocks[i]
            output, skip = block(output, skip_size)
            skips.append(skip)
        return torch.stack(skips)
```

Finally, we define the input and output networks. The input network is a simple 2×1 convolution with one sample worth of padding on either end. At forward, we clip off the last value to assure the input and output of this network are the same size. The purpose of this network is to get the input data up to the required number of channels to be put into the residual block.

```
class CausalConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(CausalConv, self).__init__()
        self.conv = nn.Conv1d(in_channels, out_channels,
                              kernel_size=2, stride=1, padding=1,
                              bias=False)

    def forward(self, x):
        output = self.conv(x)
        return output[:, :, :-1]



class OutNet(nn.Module):
    def __init__(self, channels):
        super(OutNet, self).__init__()
        self.relu = nn.ReLU()
        self.layer1 = nn.Conv1d(channels, channels, 1)
        self.layer2 = nn.Conv1d(channels, channels, 1)
        self.sm = nn.Softmax(dim=1)

    def forward(self, x):
        output = self.relu(x)
        output = self.layer1(x)
        output = self.relu(x)
        output = self.layer2(x)
        return output
```

Symmetrically, the purpose of the output network is to condense the stack of skip connections back down to the number of input channels. The output network uses two independent 1×1 convolutional layers each proceeded by a ReLU activation. The paper specifies that after the second convolution, the output should go through a softmax. However, we will be training using PyTorch's Cross Entropy Loss, which starts by calculating a softmax for us, so we can skip that step here.

4 WaveNet

Now that we have our model parts submodule, we can assemble WaveNet. To acomplish this, we need to implement four things: initialization, the forward pass, and then training and generating from one instance.

4.1 Initialization

To initialize our model, we need to take in the number of blocks, the number of layers in each block, the number of input channels, and the number of residual channels, as well as the learning rate for which we will set a default.

```
import torch
from torch import nn
import time
import os

from WaveNet.model_parts import *
from WaveNet.exceptions import InputSizeError
```

```
class WaveNet(nn.Module):
    def __init__(self, layer_size, stack_size, in_channels, residual_channels, lr=0.002):
        super(WaveNet, self).__init__()

        self.in_channels = in_channels
        self.receptive_field = sum([2**i for i in range(0, layer_size)] * stack_size)

        self.start_conv = CausalConv(in_channels, residual_channels)
        self.stack = BlockStack(layer_size, stack_size, residual_channels, in_channels)
        self.outnet = OutNet(in_channels)

        self.lr = lr
        self.loss = self._loss()
        self.optimizer = self._optimizer()

    @staticmethod
    def _loss():
        loss = torch.nn.CrossEntropyLoss()
        if torch.cuda.is_available():
            loss = loss.cuda()
        return loss

    def _optimizer(self):
        return torch.optim.Adam(self.parameters(), lr=self.lr)
```

At this point we can exactly calculate the receptive field using the following formula: $R = B\sum_{i=0}^{n} 2^i$, where R is the receptive field, B is the number of blocks, and n is the number of layers in each block. Additionally, we will define two helper functions that define the loss and the optimizer, optionally putting them on the GPU if it is available.

4.2 Forward Pass

Similarly, defining the forward pass is straight forward. Our model will start by transposing the second and third dimensions of our input to put the number of channels in the middle. After that, we simply pass forward.

```
class WaveNet(nn.module):
    ...
    def _output_size(self, x):
        output_size = int(x.shape[1]) - self.receptive_field
        if output_size < 1:
            raise InputSizeError(int(x.shape[1]), self.receptive_field, output_size)
        return output_size

    def forward(self, x, verbose=False):
        output = x.transpose(1, 2)                                  # Transpose
        output = self.start_conv(output)                            # In network
        if verbose: print('Dilating:')
        skips = self.stack(output, self._output_size(x))            # Dilation
        output = torch.sum(skips, dim=0)
        output = self.outnet(output)                                # Out network
        return output.transpose(1, 2).contiguous()                  # Retranspose
```

To pass to our residual stack, we need a third helper function that calculates our desired output size and raises an error[2] if the sample is shorter than our residual field. After getting our skip connections, we sum along the dimension we created when we stacked our skip connections in the residual block. Finally, we pass to our out network and re-transpose the dimensions back.

4.3 Training & Generating

Now that we have our forward method, we can define a method that trains the model on one instance. We simply have to pass some inputs through the `forward` method we just wrote, evaluate our loss, and step our optimizer. We'll also add a print statement for the time and for the loss, because watching the loss tick down is an integral part of the user experience and gives us an excuse to say we are working while we drink our coffee. Later, we will put our full training loop in a class that cycles through our dataset calling this method.

```
class WaveNet(nn.Module):
    ...
    def train(self, inputs, targets, verbose=False, timer=False):
        if timer: start_time = time.time()

        # Train one time
        outputs = self.forward(inputs, verbose=verbose)

        loss = self.loss(outputs.view(-1, self.in_channels),
                         targets.long().view(-1))

        if timer:
            print('\t\tLoss: {:.6f}'.format(loss.item()) \
                    + '\tTime: {:.2f} seconds'.format(time.time() - start_time))
        else:
            print('\t\tLoss: {}'.format(loss.item()))

        self.optimizer.zero_grad()

        if verbose: print('Backpropagating...')
        loss.backward()
        if verbose: print('Optimizing...')
        self.optimizer.step()

        return loss.item()

    def generate(self, inputs):
        # Generate 1 time
        return self.forward(inputs)
```

To generate on one instance, we just have to pass an input – in this case the previous receptive-field's worth of samples – through forward and return. Similarly we will create a Generator class that will do the heavy lifting of creating a seed and saving our output.

5 Trainer

Now that we have our model all defined, we can train it. To do this, we will create a `Trainer` class that takes in a parameterized model and the path of a directory to save its parameters after each file it trains over. The class will have one method, `train`, that takes in a directory of training files, the sample size, and

[2]The code for `InputSizeError` is available in Appendix A

the number of epochs. Because of the complexity of the model, we also want the option to train forever until interrupted. To accomplish this, we will set the default number of epochs to `None` and set an infinite loop with a counter and a break.

```
import WaveNet.data as data

class Trainer:
    def __init__(self, model, save_dir):
        self.model = model
        self.save_dir = save_dir

    def train(self, dir, sample_size, epochs=None, verbose=False, timer=False):
        '''
        Trains the model from files in the given directory

        Args:
            dir (str)         : Path to training files
            sample_size (int) : Number of samples in each file section, None for whole file
            epochs (int)      : Number of epochs to train, None to run until stopped
            verbose (bool)    : Should this print extra info about dilating and optimizing?
            timer (bool)      : Should this print extra info about the time taken to calculate?

        Returns: List of losses
        '''
        loader = data.WNLoader(dir, self.model.receptive_field, sample_size=sample_size)

        counter = 1

        while True:
            if epochs is not None and counter > epochs:
                break

            print('='*10, 'Epoch {}'.format(counter), '='*10)
            file_counter, epoch_losses = 1, []

            # Loop through the files
            for file in loader:
                print('\tFile {}'.format(file_counter))

                # Loop through the sections of each file
                for inputs, targets in file:
                    loss = self.model.train(inputs, targets, verbose=verbose, timer=timer)
                    epoch_losses.append(loss)
                file_counter += 1
                self.model.save(self.save_dir, step=file_counter)
            counter += 1

        return epoch_losses
```

Now the only thing we need is data. To do this we're going to use torchaudio's datasets, specifically one called CMU Arctic bdl, which is more than a thousand clips of an American man reading short fragments of public domain books. To perform the training, we will write a short script to download the files — about 98 megabytes worth — and train over them:

```
import WaveNet.WaveNet as WaveNet
import WaveNet.trainer as trainer
import warnings
from torchaudio.datasets import CMUARCTIC

if __name__ == '__main__':
    # The sox backend is deprecated, but we can ignore that for now
    warnings.filterwarnings("ignore")

    # download data
    CMUARCTIC(root='./data/', url='bdl', download=True)

    # Initialize and train the model
    model = Wavenet.WaveNet(5, 12, 256, 512)
    print('Receptive Field:', model.receptive_field)
    trainer = trainer.Trainer(model, './model_saves/')
    trainer.train('./data/ARCTIC/cmu_us_bdl_arctic/wav/', 25000, epochs=1, timer=True)
```

```
>>> python train.py
Receptive Field: 20475
========== Epoch 1 ==========
        File 1
                Loss: 5.551476        Time: 6.76 seconds
                Loss: 5.470233        Time: 6.69 seconds
                Loss: 5.474391        Time: 6.49 seconds
                Loss: 5.473541        Time: 6.64 seconds
                Loss: 5.437691        Time: 6.93 seconds
                Loss: 5.322380        Time: 5.71 seconds
Saving model into ./model_saves/
        File 2
                Loss: 5.156125        Time: 6.84 seconds
                Loss: 5.210230        Time: 7.09 seconds
                Loss: 5.179309        Time: 6.64 seconds
                Loss: 5.210064        Time: 6.56 seconds
                Loss: 5.052970        Time: 6.53 seconds
                Loss: 5.000860        Time: 7.55 seconds
                Loss: 4.989880        Time: 7.05 seconds
    ...
```

Despite being considered fast as compared to other architectures of the past, it's still *state-of-the-art* fast which means it takes us plebeians days to finish a single epoch. This model will train overnight for 317 randomly selected files, or about a third of an epoch, totalling slightly more than 3000 samples. This turns out to be more than enough for the model to stabilize.

In Figure 2, we can see the loss took a significant drop in the first 500 samples and leveled out after. The limitation of this model is not the amount of data, but computing power. The rate at which this levels off is dependant on the receptive field, itself dependent on the size of the model. It is worth noting that this training could not be done on Google Colab, because the session would crash after eating through all of the available RAM if the receptive field was larger than around ten thousand. Instead, this was done using local runtime.

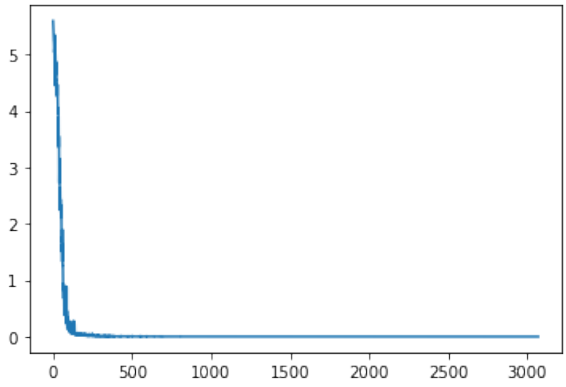

Figure 2: Cross Entropy Loss over time

6 Generator

Now that we have a model that has trained for some length of time, we can try getting it to generate something new of its own. To do this, we need to feed it a section of seed audio to act as the first input. After generating on the seed, it will look back at its previous generated material and keep constructing recursively. To accomplish this, we will create a generator class similar to our trainer class that loads a trained model, the path to a seed, the path to the output file and some other attributes.

```
import torch
import numpy as np
import os
from soundfile import write as write_wav

from WaveNet.Wavenet import *
import WaveNet.data as data

class Generator:
    def __init__(self, model, model_dir, step, sample_rate, seed_path, out_path, sample_size):
        """
        Args:
            model      (WaveNet) : Trained WaveNet model
            model_dir     (path) : Path to the model's saved parameters
            step           (int) : The number of the model to load
            sample_rate    (int) : Sample rate at which to generate
            seed_path     (path) : Path to the seed file
            out_path      (path) : File path of output file
            sample_size    (int) : Number of samples to generate
        """
        self.sample_rate = sample_rate
        self.sample_size = sample_size
        self.seed = seed_path
        self.out_path = out_path

        self.wavenet = model
        self.wavenet.load(model_dir, step)
```

Our generate method starts by companding and encoding the seed[3] and then feeding it through the generate method. After the first one, it reassigns the input to whatever it just generated and repeats until the output is the requested length or, if none is given, two receptive fields.

```
class Generator:
    ...
    def generate(self):
        """
        Generate from the given seed
        """
        outputs = []
        inputs, audio_length = self._get_seed_from_audio(self.seed)

        while True:
            new = self.wavenet.generate(inputs)

            # Not relevant but I love this syntax
            outputs = torch.cat((outputs, new), dim=1) if len(outputs) else new

            if len(outputs[0] >= audio_length):
                break

            # If not long enough, reassign input to whatever we just generated
            inputs = torch.cat((inputs[:, :-len(new[0]), :], new), dim=1)

        # Clip output to the desired length and save
        outputs = outputs[:, :audio_length, :]
        self._save_to_audio_file(outputs)
```

With our `Generator` class, we just need a quick script to actually run.

```
from WaveNet.Wavenet import WaveNet
from WaveNet.generator import Generator


if __name__ == '__main__':
        model = WaveNet(5, 10, 256, 512)

        for save_step in [2, 50, 100, 150, 200, 250, 300, 317]:
            outpath = './generator_output/gen_{}.wav'.format(save_step)
            generator = Generator(model, './model_saves',
                save_step, 16000, './data/helloworld.wav', outpath, 100000)
            generator.generate()
```

```
>>> python gen.py
Loading model from ./model_saves
94885/1668550 samples are generated.
Saved wav file at ./generator_output/gen_2.wav
Loading model from ./model_saves
94885/1668550 samples are generated.
...
```

Listening to the samples, we get what sounds like rhythmic static. A keen ear will notice that that `gen_2.wav` has more high end[4], and looking at the spectrograms in Figure 3, we see `gen_317.wav` is slightly

[3] The methods that do this in the `Generator` class are available in Appendix A

[4] I'd like to take this opportunity to thank Dr. Doug Bielmeier for forcing me to do ear training in Recording 1.

darker in the areas between the breaks, indicated a lesser presence of frequency content there relative to the darker portions. The rhythmic nature of these examples is the most interesting feature, likely having been picked up by the model from the rhythmic nature of the seed file. This would mean that at this level of complexity and this amount of training, the model can at least pick up on broad structural features of the seed. For comparison when seeded with `arctic_a0001.wav`, we do not see this rhythm, shown in `gen_counter.wav`.

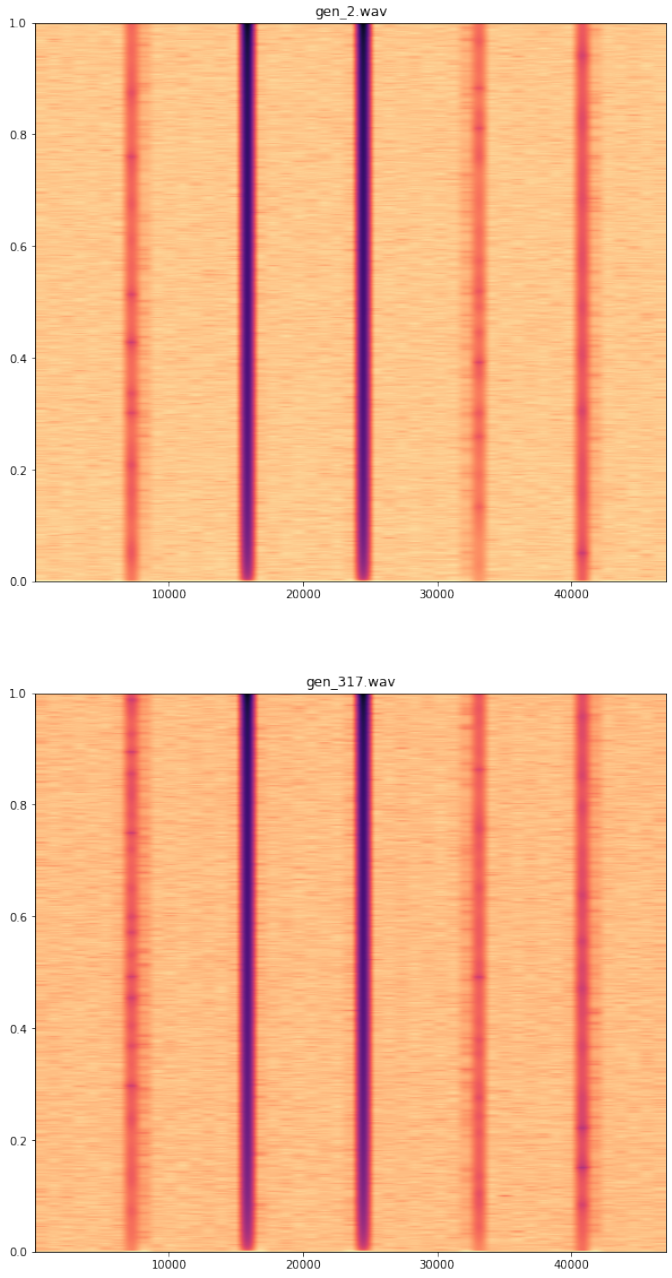

Figure 3: Mel Spectrograms for `gen_2.wav` and `gen_317.wav`

7 Conclusion & Next Steps

With the computing power available, we built and trained a model that can recognize and imitate broad features like repetitive rhythms in audio files. While the results may not appear impressive at first pass, the fact this is at all possible using 317 audio clips and a GTX 1660 Super is a feat. An obvious next step would be to train a more complicated model for longer, which might give results similar to the samples in the original WaveNet blog post. There are also a number of other features still left to implement, like combining this with a text-to-sequence model for a full text-to-speech application, or including interchangeable speaker embedding vectors. This paper has just scratched the surface getting the model up and running. The modular approach allows for the expansion of this architecture into whatever creativity and computational complexity allows.

A Extra Code Snippets

A.1 Backend dataset

```
class CustomDataset(d.Dataset):
    def __init__(self, dir, sr=16000, channels=256, trim=2048):
        '''
        dir : Path to directory of audio files
        sr  : Sample rate of audio files
        channels : Number of quantization channels
        trim : Number of samples in each segment, 0 is no trim
        '''
        super(CustomDataset, self).__init__()

        self.channels = channels
        self.sample_rate = sr
        self.trim = trim

        self.path = dir
        self.filenames = [filename for filename in sorted(os.listdir(dir))]

    def __getitem__(self, idx):
        file = os.path.join(self.path, self.filenames[idx])
        audio = load(file, self.sample_rate, self.trim)
        companded = mu_law_encode(audio, self.channels)
        return one_hot_encode(companded, self.channels)

    def __len__(self):
        return len(self.filenames)
```

A.2 InputSizeError

```
class InputSizeError(Exception):
    def __init__(self, input_size, receptive_fields, output_size):

        message = 'Input size has to be larger than receptive_fields\n'
        message += 'Input size: {0}, Receptive fields size: {1}, Output size: {2}'.format(
            input_size, receptive_fields, output_size)

        super(InputSizeError, self).__init__(message)
```

A.3 `Generator` methods

```
class Generator:
    ...
    @staticmethod
    def _variable(data):
        tensor = torch.from_numpy(data).float()

        if torch.cuda.is_available():
            return torch.autograd.Variable(tensor.cuda())
        else:
            return torch.autograd.Variable(tensor)

    def _make_seed(self, audio):
        audio = np.pad([audio], [[0, 0], [self.wavenet.receptive_field, 0], [0, 0]], 'constant')

        if self.sample_size:
            seed = audio[:, :self.sample_size, :]
        else:
            seed = audio[:, :self.wavenet_receptive_field * 2, :]

        return seed

    def _get_seed_from_audio(self, filepath):
        audio = data.load(filepath, self.sample_rate)
        audio_length = len(audio)

        audio = data.mu_law_encode(audio, self.wavenet.in_channels)
        audio = data.one_hot_encode(audio, self.wavenet.in_channels)

        seed = self._make_seed(audio)

        return self._variable(seed), audio_length

    def _save_to_audio_file(self, output):
        output = output[0].cpu().data.numpy()
        output = data.one_hot_decode(output, axis=1)
        waveform = data.mu_law_decode(output, self.wavenet.in_channels)

        write_wav(self.out_path, waveform, self.sample_rate)
        print('Saved wav file at {}'.format(self.out_path))
```