THE HONG KONG POLYTECHNIC UNIVERSITY

CAPSTONE PROJECT

Comparison of the Performance of GAN and Diffusion Models in the Vocoder Field

Author:
XUE, Bingxin

Supervisor:
Dr. Jiang, Binyan

A thesis submitted in fulfillment of the requirements
for the degree of Bachelor of Science
in the
Data Science and Analytics

May 12, 2024

C Data Shape Transformation

C.1 From Audio Signal to Mel-Spectrogram

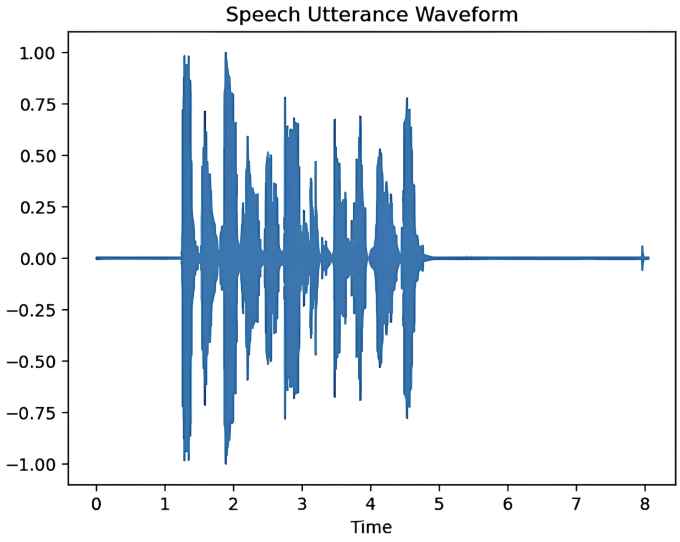


Figure 24: (Roberts, 2020)

As discussed above in the main text, the data is preprocessed before it goes to the vocoder to convert the signal into a mel-spectrogram.

The sound we can hear is a perception created by the vibration of an object propagating through a medium. When an object vibrates, it causes periodic compression and rarefaction of the molecules in the surrounding medium, creating sound waves. These sound waves travel through the air or other media and are eventually received by our ears and interpreted as sound by the auditory system.

However, if we want to process sound with a computer, we need to convert the sound into a digital signal. Thus, a signal can be said to be a representation of sound. We capture the waveform of the sound by sampling it and, using the Fourier Transform, convert the signal from the time domain to the frequency domain, producing a spectrogram.

The mel-spectrogram is one of the spectrograms more suitable for human perception. Taking Hi-Fi GAN as an example, the sample data of the LJSpeech Dataset is speech, as shown below:

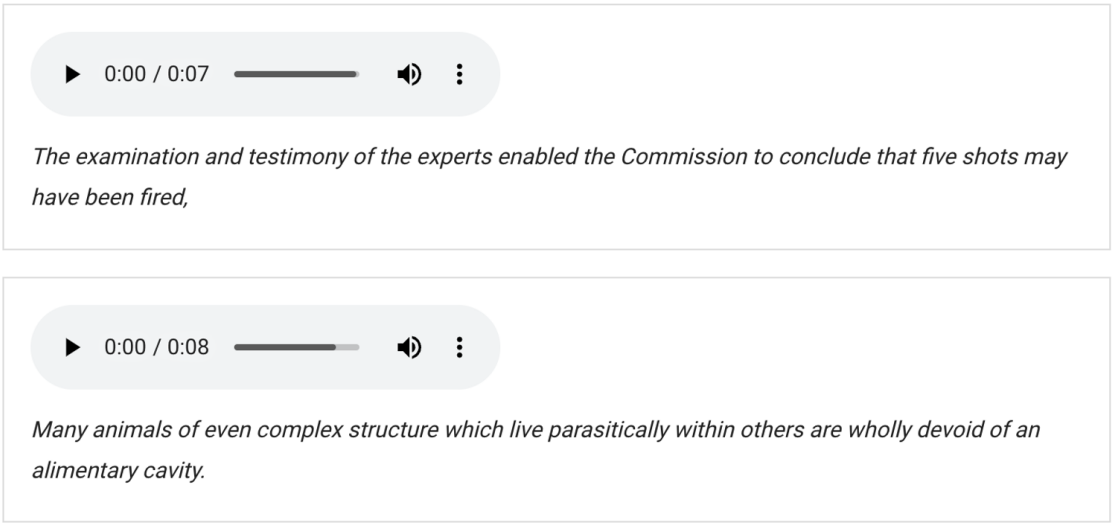


Figure 25: Screenshot of Database (Ito and Johnson, n.d.)

During training, the audio will be loaded through MelDataset. The content of the MelDataset class includes normalising the data and converting them into tensors:

```
def __getitem__(self, index):
    filename = self.audio_files[index]
    if self._cache_ref_count == 0:
        audio, sampling_rate = load_wav(filename)       # Load WAV file
        audio = audio / MAX_WAV_VALUE                   # Normalize audio data
        if not self.fine_tuning:
            audio = normalize(audio) * 0.95             # Normalize audio data further
        self.cached_wav = audio                         # Cache the normalized audio data
        if sampling_rate != self.sampling_rate:
            raise ValueError("{} SR doesn't match target {} SR".format(
                sampling_rate, self.sampling_rate))     # Raise value error
        self._cache_ref_count = self.n_cache_reuse      # Reset cache reference count
    else:
        audio = self.cached_wav                         # Use cached audio data
        self._cache_ref_count -= 1                      # Decrement cache reference count

    audio = torch.FloatTensor(audio)                    # Convert audio to PyTorch float tensor
    audio = audio.unsqueeze(0)                          # Add singleton dimension to audio tensor
```

Figure 26: Screenshot of code (meldataset.py) (Jungil Kong, 2020)

Spliting the audio data into frames and converting them into mel-spectrogram through the mel_spectrogram function, and also calculate the mel-spectrogram of the loss:

```
        if not self.fine_tuning:
            if self.split:
                if audio.size(1) >= self.segment_size:
                    max_audio_start = audio.size(1) - self.segment_size  # Compute maximum audio start index
                    audio_start = random.randint(0, max_audio_start)  # Generate random audio start index
                    audio = audio[:, audio_start:audio_start+self.segment_size]  # Extract audio segment
                else:
                    audio = torch.nn.functional.pad(audio, (0, self.segment_size - audio.size(1)), 'constant')  # Pad audio
            mel = mel_spectrogram(audio, self.n_fft, self.num_mels,
                                  self.sampling_rate, self.hop_size, self.win_size, self.fmin, self.fmax,
                                  center=False)             # Compute Mel spectrogram
        else:
            mel = np.load(                                  # Load precomputed Mel spectrogram
                os.path.join(self.base_mels_path, os.path.splitext(os.path.split(filename)[-1])[0] + '.npy'))
            mel = torch.from_numpy(mel)                     # Convert Mel spectrogram to tensor

            if len(mel.shape) < 3:
                mel = mel.unsqueeze(0)                      # Add singleton dimension to Mel spectrogram

            if self.split:
                frames_per_seg = math.ceil(self.segment_size / self.hop_size)  # Compute frames per segment
                if audio.size(1) >= self.segment_size:
                    mel_start = random.randint(0, mel.size(2) - frames_per_seg - 1)  # Generate random Mel start index
                    mel = mel[:, :, mel_start:mel_start + frames_per_seg]  # Extract Mel segment
                    audio = audio[:, mel_start * self.hop_size:(mel_start + frames_per_seg) * self.hop_size]  # Extract audio segment
                else:
                    mel = torch.nn.functional.pad(mel, (0, frames_per_seg - mel.size(2)), 'constant')  # Pad Mel spectrogram
                    audio = torch.nn.functional.pad(audio, (0, self.segment_size - audio.size(1)), 'constant')  # Pad audio

        mel_loss = mel_spectrogram(audio, self.n_fft, self.num_mels,
                                   self.sampling_rate, self.hop_size, self.win_size, self.fmin, self.fmax_loss,
                                   center=False)            # Compute Mel spectrogram for loss

        # Return Mel spectrogram, audio, filename, and Mel spectrogram for loss
        return (mel.squeeze(), audio.squeeze(0), filename, mel_loss.squeeze())
```

Figure 27: Extension of the code above. (meldataset.py) (Jungil Kong, 2020)

```
def mel_spectrogram(y, n_fft, num_mels, sampling_rate, hop_size, win_size, fmin, fmax, center=False):
    if torch.min(y) < -1.:
        print('min value is ', torch.min(y))
    if torch.max(y) > 1.:
        print('max value is ', torch.max(y))

    global mel_basis, hann_window
    if fmax not in mel_basis:
        mel = librosa_mel_fn(sampling_rate, n_fft, num_mels, fmin, fmax)
        mel_basis[str(fmax)+'_'+str(y.device)] = torch.from_numpy(mel).float().to(y.device)
        hann_window[str(y.device)] = torch.hann_window(win_size).to(y.device)

    # # Apply padding to input tensor
    y = torch.nn.functional.pad(y.unsqueeze(1), (int((n_fft-hop_size)/2), int((n_fft-hop_size)/2)), mode='reflect')
    y = y.squeeze(1)

    # Compute Short-Time Fourier Transform
    spec = torch.stft(y, n_fft, hop_length=hop_size, win_length=win_size, window=hann_window[str(y.device)],
                      center=center, pad_mode='reflect', normalized=False, onesided=True)

    # Compute magnitude spectrogram
    spec = torch.sqrt(spec.pow(2).sum(-1)+(1e-9))

    # Generate Mel Spectrogram
    spec = torch.matmul(mel_basis[str(fmax)+'_'+str(y.device)], spec)

    # # Normalize the spectrogram
    spec = spectral_normalize_torch(spec)

    return spec
```

Figure 28: Details of the mel_sepctrogram function (meldataset.py) (Jungil Kong, 2020)

The mel-spectrogram method is described in detail below:

```
with torch.no_grad():
    for i, filname in enumerate(filelist):
        wav, sr = load_wav(os.path.join(a.input_wavs_dir, filname))
        pdb.set_trace() # 1

        wav = wav / MAX_WAV_VALUE
        wav = torch.FloatTensor(wav).to(device)
        pdb.set_trace() # 2

        x = get_mel(wav.unsqueeze(0))
        pdb.set_trace() # 13

        start_1 = perf_counter()
        y_g_hat = generator(x)
        pdb.set_trace() # 14

        end_1 = perf_counter()
        rtf_1 = (wav.shape[-1] / 22050) / (end_1 - start_1)
        rtf_list.append(rtf_1)
        audio = y_g_hat.squeeze()
        audio = audio * MAX_WAV_VALUE
        audio = audio.cpu().numpy().astype('int16')

        output_file = '/data2/xbx/hifi-gan-1/output/' + filname.split("/")[-1]
        write(output_file, h.sampling_rate, audio)

        print(output_file)

    sum = 0
    for i in rtf_list:
        sum += i
    pdb.set_trace()
    print(sum / len(rtf_list))
```

Figure 29: inference.py (Jungil Kong, 2020)

pdb.set_trace is a breakpoint. Taking an audio of size 114077 as an example.

1 (first breakpoint): Check the signal shape when the audio is loaded.

```
(Pdb) wav
array([-82, -36,   3, ...,  39,  34,  28], dtype=int16)
(Pdb) wav.shape
(114077,)
```

The 114077 represents the number of samples in the audio waveform, i.e. the length of the audio signal.

2 (second breakpoint): When the audio signal is normalised and converted into a tensor the signal form and its shape:

```
(Pdb) wav
tensor([-2.5024e-03, -1.0986e-03,  9.1553e-05,  ...,  1.1902e-03,
         1.0376e-03,  8.5449e-04], device='cuda:0')
(Pdb) wav.shape
torch.Size([114077])
```

3 (same as above): The audio data is fed into the get_mel method in the form of wav.unsqueeze(0). get_mel is a call to the mel_spectrogram method, where we start the mel-spectrogram generation phase.

```
def get_mel(x):
    return mel_spectrogram(x, h.n_fft, h.num_mels, h.sampling_rate, h.hop_size, h.win_size, h.fmin, h.fmax)
```

Figure 30: inference.py (Jungil Kong, 2020)

C.1.1 Mel-Spectrogram Method

```
def mel_spectrogram(y, n_fft, num_mels, sampling_rate, hop_size, win_size, fmin, fmax, center=False):
    if torch.min(y) < -1.:      # Check if the minimum value of the input tensor y is less than -1
        print('min value is ', torch.min(y))
    if torch.max(y) > 1.:       # Check if the maximum value of the input tensor y is greater than 1
        print('max value is ', torch.max(y))

    pdb.set_trace() # 3
    global mel_basis, hann_window
    if fmax not in mel_basis:   # Check if the maximum frequency fmax is not already in the precomputed mel basis
        # Compute mel filterbank
        mel = librosa_mel_fn(sampling_rate, n_fft, num_mels, fmin, fmax)
        pdb.set_trace() # 4
        # Store mel basis in a dictionary for future use
        mel_basis[str(fmax)+'_'+str(y.device)] = torch.from_numpy(mel).float().to(y.device)
        pdb.set_trace() # 5
        # Compute Hann window and store it in a dictionary
        hann_window[str(y.device)] = torch.hann_window(win_size).to(y.device)
        pdb.set_trace() # 6

    # Pad the input tensor y with reflection padding
    y = torch.nn.functional.pad(y.unsqueeze(1), (int((n_fft-hop_size)/2), int((n_fft-hop_size)/2)), mode='reflect')
    pdb.set_trace() # 7
    # Squeeze the tensor y along the second dimension
    y = y.squeeze(1)
    pdb.set_trace() # 8
    # Compute short-time Fourier transform (STFT) of the input tensor y
    spec = torch.stft(y, n_fft, hop_length=hop_size, win_length=win_size, window=hann_window[str(y.device)],
                      center=center, pad_mode='reflect', normalized=False, onesided=True)
    pdb.set_trace() # 9
    # Compute magnitude spectrogram
    spec = torch.sqrt(spec.pow(2).sum(-1)+(1e-9))
    pdb.set_trace() # 10
    # Apply mel filterbank to the magnitude spectrogram
    spec = torch.matmul(mel_basis[str(fmax)+'_'+str(y.device)], spec)
    pdb.set_trace() # 11
    # Normalize the spectrogram
    spec = spectral_normalize_torch(spec)
    pdb.set_trace() # 12
    # Return the spectrogram
    return spec
```

Figure 31: meldataset.py (Jungil Kong, 2020)

3: The purpose of wav.unsqueeze(0) is to insert a dimension into the first dimension, and y is input audio signal data.

```
(Pdb) y.shape
torch.Size([1, 114077])
(Pdb) n_fft, num_mels, sampling_rate, hop_size, win_size, fmin, fmax
(1024, 80, 22050, 300, 1024, 80, 8000)
```

4:

```
(Pdb) mel.shape
(80, 513)
(Pdb) mel
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

5:

```
(Pdb) mel_basis[str(fmax)+'_'+str(y.device)]
tensor([[0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.]], device='cuda:0')
(Pdb) mel_basis[str(fmax)+'_'+str(y.device)].shape
torch.Size([80, 513])
```

6:

```
(Pdb) hann_window[str(y.device)]
tensor([0.0000e+00, 9.4175e-06, 3.7640e-05,  ..., 8.4698e-05, 3.7640e-05,
        9.4175e-06], device='cuda:0')
(Pdb) hann_window[str(y.device)].shape
torch.Size([1024])
```

7:

```
(Pdb) y
tensor([[[0.0144, 0.0171, 0.0157,  ..., 0.0007, 0.0009, 0.0011]]],
       device='cuda:0')
(Pdb) y.shape
torch.Size([1, 1, 114801])
```

8: The purpose of y.squeeze(1) is to remove the dimension with dimension 1.

```
(Pdb) y.shape
torch.Size([1, 114801])
```

9: Spectrogram

```
spec:tensor( [ [ [ [-5.3430e-02,   0.0000e+00],
                   [-2.5064e-02,   0.0000e+00],
                   [ 7.5338e-03,   0.0000e+00],
                   ...,
                   [-2.7422e-04,   0.0000e+00],
                   [ 5.8605e-06,   0.0000e+00],
                   [ 4.1137e-05,   0.0000e+00] ] ] ], device='cuda:0')
```

spec.shape: torch.Size([1, 513, 380, 2])

10: Magnitude Spectrogram

spec.shape: torch.Size([1, 513, 380])

11: Magnitude Spectrogram

```
(Pdb) spec.shape
torch.Size([1, 80, 380])
(Pdb) spec
tensor([[[2.4392e-03, 3.0793e-03, 4.2780e-03,  ..., 1.8907e-03,
          1.7693e-03, 2.4423e-03],
         [1.9216e-02, 2.4079e-02, 5.7693e-03,  ..., 1.6415e-03,
          1.3375e-03, 1.8210e-03],
         [6.3031e-02, 2.4042e-01, 3.2363e-01,  ..., 1.7247e-03,
          1.4511e-03, 2.5266e-03],
         ...,
         [2.1550e-03, 5.9826e-03, 7.2751e-03,  ..., 6.1339e-05,
          6.9058e-05, 7.1356e-05],
         [1.1718e-03, 3.1190e-03, 4.3347e-03,  ..., 6.1210e-05,
          7.1279e-05, 7.1111e-05],
         [4.8726e-04, 1.0784e-03, 1.9367e-03,  ..., 7.1329e-05,
          5.8054e-05, 6.5357e-05]]], device='cuda:0')
```

spec.shape: torch.Size([1, 80, 380])

12: Normalize the spectrogram.

```
(Pdb) spec
tensor([[[-6.0161, -5.7830, -5.4543,  ..., -6.2708, -6.3372, -6.0148],
         [-3.9520, -3.7264, -5.1552,  ..., -6.4121, -6.6169, -6.3084],
         [-2.7641, -1.4254, -1.1281,  ..., -6.3627, -6.5355, -5.9809],
         ...,
         [-6.1400, -5.1189, -4.9233,  ..., -9.6991, -9.5806, -9.5478],
         [-6.7492, -5.7703, -5.4411,  ..., -9.7012, -9.5489, -9.5513],
         [-7.6267, -6.8323, -6.2468,  ..., -9.5482, -9.7541, -9.6356]]],
       device='cuda:0')
```

The mel_spectrogram inputs audio data and returns the mel-spectrogram, so the x in x = get_mel(wav.unsqueeze(0)) is the returned mel-spectrogram. Here, we are done with the data preprocessing part. The first dimension, 1 in [1,80,380], means batch size, meaning that only one piece of audio has been processed.

The second dimension, 80, means the number of channels of the mel-spectrogram, that is, the dimension of the mel-spectrogram.

The third dimension, 380, represents the length of the audio data, which can also be described as the number of time steps, i.e. the number of frames of the audio signal in time.

C.2 From Mel-Spectrogram to Audio

The next step is to enter the processed data into the generator:

```
with torch.no_grad():
    for i, filname in enumerate(filelist):
        wav, sr = load_wav(os.path.join(a.input_wavs_dir, filname))
        wav = wav / MAX_WAV_VALUE
        wav = torch.FloatTensor(wav).to(device)
        # pdb.set_trace()
        x = get_mel(wav.unsqueeze(0))
        start_1 = perf_counter()
        y_g_hat = generator(x)
        end_1 = perf_counter()
        rtf_1 = (wav.shape[-1] / 22050) / (end_1 - start_1)
        rtf_list.append(rtf_1)
        audio = y_g_hat.squeeze()
        audio = audio * MAX_WAV_VALUE
        audio = audio.cpu().numpy().astype('int16')

        output_file = '/data2/xbx/hifi-gan-1/output/' + filname.split("/")[-1]
        # pdb.set_trace()
        write(output_file, h.sampling_rate, audio)

        print(output_file)

    sum = 0
    for i in rtf_list:
        sum += i
    pdb.set_trace()
    print(sum / len(rtf_list))
```

Figure 32: train.py (Jungil Kong, 2020)

```
def inference(a):
    generator = Generator(h).to(device)
```

Figure 33: train.py (Jungil Kong, 2020)

```python
class Generator(torch.nn.Module):
    def __init__(self, h):
        super(Generator, self).__init__()
        self.h = h
        self.num_kernels = len(h.resblock_kernel_sizes)
        self.num_upsamples = len(h.upsample_rates)
        self.conv_pre = weight_norm(Conv1d(80, h.upsample_initial_channel, 7, 1, padding=3))
        resblock = ResBlock1 if h.resblock == '1' else ResBlock2

        self.ups = nn.ModuleList()
        for i, (u, k) in enumerate(zip(h.upsample_rates, h.upsample_kernel_sizes)):
            self.ups.append(weight_norm(
                ConvTranspose1d(h.upsample_initial_channel//(2**i), h.upsample_initial_channel//(2**(i+1)),
                                k, u, padding=(k-u)//2)))

        self.resblocks = nn.ModuleList()
        for i in range(len(self.ups)):
            ch = h.upsample_initial_channel//(2**(i+1))
            for j, (k, d) in enumerate(zip(h.resblock_kernel_sizes, h.resblock_dilation_sizes)):
                self.resblocks.append(resblock(h, ch, k, d))

        self.conv_post = weight_norm(Conv1d(ch, 1, 7, 1, padding=3))
        self.ups.apply(init_weights)
        self.conv_post.apply(init_weights)

    def forward(self, x):
        x = self.conv_pre(x)
        for i in range(self.num_upsamples):
            x = F.leaky_relu(x, LRELU_SLOPE)
            x = self.ups[i](x)
            xs = None
            for j in range(self.num_kernels):
                if xs is None:
                    xs = self.resblocks[i*self.num_kernels+j](x)
                else:
                    xs += self.resblocks[i*self.num_kernels+j](x)
            x = xs / self.num_kernels
        x = F.leaky_relu(x)
        x = self.conv_post(x)
        x = torch.tanh(x)

        return x
```

Figure 34: models.py (Jungil Kong, 2020)

generator: self.conv_pre(x),
self.conv_pre = weight_norm(Conv1d(80, h.upsample_initial_channel, 7, 1, padding=3))

1. 80: Number of input channels, indicating that the dimension of the input feature is 80.
2. h.upsample_initial_channel: Number of output channels; this value is defined as 512 in the code and indicates the number of channels in the feature map output by the convolutional layer.
3. 7: Convolution kernel size
4. 1: Step size
5. padding=3: Padding size, which means three elements at each end of the input.

The weight_norm is a function that applies weight normalisation to this convolutional layer, which is done to improve the training stability and convergence performance of the model. By weight normalisation, the parameters in the network can be better controlled, and the problem of exploding and vanishing gradients can be lessened, helping to speed up the model training process. Therefore x.shape changed from [1, 80, 380] to [1, 512, 380].

After that, the data is upsampled: x = self.ups[i](x). the number of channels = h.upsample_initial_channel//(2**(i+1)), and upsample_initial_channel= 512,

Besides these, input_length = 380, u = upsample_rates = [10,5,3,2], k = upsample_kernel_sizes = [16,16,4,4], padding = padding =(k-u)//2) = [3, 5, 0, 1].

i	**Number of Channel** $(\frac{512}{2^{2(i+1)}})$	Upsample Rate (u)	Upsample Kernel Size (k)	Padding $(\frac{k-u}{2})$
0	**256**	10	16	3
1	**128**	5	16	5
2	**64**	3	4	0
3	**32**	2	4	1

$$\text{Output Length} = (\text{Input Length} - 1) \times \text{Upsampling Rate} - 2 \times \text{Padding} + \text{Kernel Size} \quad (15)$$

Here are the detailed calculation steps:

i	Input Length	Upsampling Rate (u)	Padding $(\frac{k-u}{2})$	Kernel Size (k)	**Output Length**
0	380	10	3	16	**3800**
1	3800	5	5	16	**19001**
2	19001	3	0	4	**57004**
3	57004	2	1	4	**114008**

So after four upsamples, the data becomes the following:
$[1, 512, 380] \rightarrow [1, 256, 3800] \rightarrow [1, 128, 19001] \rightarrow [1, 64, 57004] \rightarrow [1, 32, 114008]$

The data is sampled and then convolved: self.conv_post(x)
self.conv_post = weight_norm(Conv1d(ch, 1, 7, 1, padding=3)) Output layer convolution maps the feature vector to the final audio waveform and returns. Data Shape : $[1, 32, 114008] \rightarrow [1, 1, 114008]$ print(x.shape): torch.Size([1, 1, 114008])

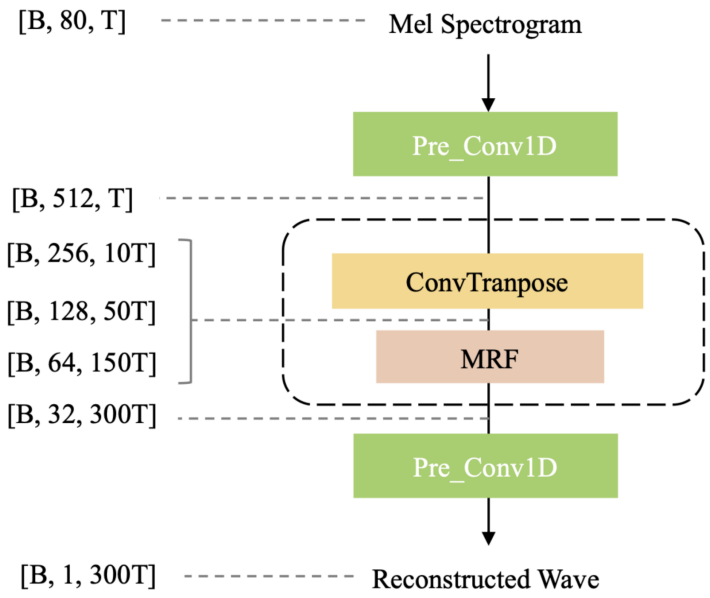
print(y_g_hat.shape): torch.Size([1, 1, 114008])

```
with torch.no_grad():
    for i, filname in enumerate(filelist):
        wav, sr = load_wav(os.path.join(a.input_wavs_dir, filname))
        wav = wav / MAX_WAV_VALUE
        wav = torch.FloatTensor(wav).to(device)
        # pdb.set_trace()
        x = get_mel(wav.unsqueeze(0))
        start_1 = perf_counter()
        y_g_hat = generator(x)
        end_1 = perf_counter()
        rtf_1 = (wav.shape[-1] / 22050) / (end_1 - start_1)
        rtf_list.append(rtf_1)
        audio = y_g_hat.squeeze()
        audio = audio * MAX_WAV_VALUE
        audio = audio.cpu().numpy().astype('int16')

        output_file = '/data2/xbx/hifi-gan-1/output/' + filname.split("/")[-1]
        # pdb.set_trace()
        write(output_file, h.sampling_rate, audio)

        print(output_file)

    sum = 0
    for i in rtf_list:
        sum += i
    pdb.set_trace()
    print(sum / len(rtf_list))
```

Figure 35: train.py (Jungil Kong, 2020)

print(audio.shape): torch.Size([114008])

After the squeeze() process, the 3D tensor is converted to a 1D vector. The shape is consistent with the audio data loaded at the beginning. The following is the detailed procedure for generating audio from mel-spectrogram in Hi-Fi GAN's generator:



C.3 WaveGrad Preprocessing Steps

```
mel_fn = MelSpectrogramFixed(
    sample_rate=config.data_config.sample_r
    n_fft=config.data_config.n_fft,
    win_length=config.data_config.win_lengt
    hop_length=config.data_config.hop_lengt
    f_min=config.data_config.f_min,
    f_max=config.data_config.f_max,
    n_mels=config.data_config.n_mels,
    window_fn=torch.hann_window
).cuda()
```

Figure 36: train.py (Vovk, 2020)

```
class MelSpectrogramFixed(torch.nn.Module):
    """In order to remove padding of torchaudio package + add log10 scale."""
    def __init__(self, **kwargs):
        super(MelSpectrogramFixed, self).__init__()
        self.torchaudio_backend = MelSpectrogram(**kwargs)

    def forward(self, x):
        outputs = self.torchaudio_backend(x).log10()
        mask = torch.isinf(outputs)
        outputs[mask] = 0
        return outputs[..., :-1]
```

Figure 37: data.py (Vovk, 2020)

WaveGrad directly references the MelSpectrogramFixed module to calculate the mel-spectrogram when training the model. The 'MelSpectrogramFixed' module calls the 'MelSpectrogram' function from the Torchaudio package.

The 'forward' method first calls the 'MelSpectrogram' function to compute the mel-spectrogram of the original audio and then applies a logarithmic transformation (log10) to scale the result. Then, it checks if any infinity values appear in the computed result and replaces them with zeros. Finally, the last element of the mel-spectrogram is deleted, and the processed mel-spectrogram is output.

The following is the data transformation process in the data preprocessing stage:

```
for audio_path in (tqdm(audio_filelist, leave=False) if args.verbose else audio_filelist):  # Iterate over the audio file list
    with torch.no_grad():

        audio, sample_rate = torchaudio.load(audio_path)                    # Load the audio data
        pdb.set_trace() # 1  audio.shape: torch.Size([1, 154508])           # Pause execution for debugging

        audio = audio.to(device)                                            # Move the audio data to the specified device

        # Calculate padding
        p = (audio.shape[-1] // config.data_config.hop_length + 1) * config.data_config.hop_length - audio.shape[-1]
        audio = torch.nn.functional.pad(audio, (0, p), mode='constant').data  # Pad the audio data

        audio = audio.squeeze()                                             # Remove redundant dimensions from the audio data
        pdb.set_trace() # 2  audio.shape: torch.Size([154800])

        mel = mel_fn(audio)[None]                                           # Compute the Mel spectrogram for the audio
        pdb.set_trace() # 3  mel.shape: torch.Size([1, 80, 604])

        start = datetime.now()                                              # Record start time
        outputs = model.forward(mel, store_intermediate_states=False)       # Perform model inference
        pdb.set_trace() # 19  outputs.shape: torch.Size([1, 181200])

        end = datetime.now()                                                # Record end time

        outputs = outputs.cpu().squeeze()  # Move the output back to CPU and remove redundant dimensions
        baseidx = os.path.basename(os.path.abspath(audio_path)).split('_')[-1].replace('.pt', '')  # Extract base index
        save_path = f'/data2/xbx/WaveGrad/output/{baseidx}'                 # Set the save path
        torchaudio.save(                                                    # Save the output audio
            save_path, outputs, sample_rate=config.data_config.sample_rate
        )

        inference_time = (end - start).total_seconds()                      # Calculate inference time
        rtf = compute_rtf(outputs, inference_time, sample_rate=config.data_config.sample_rate)  # Calculate RTF
        rtfs.append(rtf)                                                    # Store RTF
```

Figure 38: inference.py (Vovk, 2020)

C.4 Reconstructing Noise

Reconstruction noise is an integral part of the WaveGrad Model and is required for both forward and reverse processes.

The following is the specific data conversion process (the data has been preprocessed into mel-spectrogram):(The 'mels' is the 'mel' in step 3)

```
def forward(self, mels, store_intermediate_states=False):
    pdb.set_trace() # 4 mels.shape: torch.Size([1, 80, 604])
    """
    Generates speech from given mel-spectrogram.
    :param mels (torch.Tensor): mel-spectrogram tensor of shape [1, n_mels, T//hop_length]
    :param store_intermediate_states (bool, optional):
        flag to set return tensor to be a set of all states of denoising process
    """
    self._verify_noise_schedule_existence()

    return self.sample(
        mels, store_intermediate_states
    )
```

Figure 39: diffusion_process.py (Vovk, 2020)

```
def sample(self, mels, store_intermediate_states=False):
    """
    Samples speech waveform via progressive denoising of white noise with guidance of mels-epctrogram.
    :param mels (torch.Tensor): mel-spectrograms acoustic features of shape [B, n_mels, T//hop_length]
    :param store_intermediate_states (bool, optional): whether to store dynamics trajectory or not
    :return ys (list of torch.Tensor) (if store_intermediate_states=True)
        or y_0 (torch.Tensor): predicted signals on every dynamics iteration of shape [B, T]
    """
    with torch.no_grad():
        device = next(self.parameters()).device
        batch_size, T = mels.shape[0], mels.shape[-1]
        ys = [torch.randn(batch_size, T*self.total_factor, dtype=torch.float32).to(device)]
        t = self.n_iter - 1
        while t >= 0:
            pdb.set_trace() # 5 mels.shape: torch.Size([1, 80, 604])

            y_t = self.compute_inverse_dynamics(mels, y=ys[-1], t=t)
            pdb.set_trace() # 18 y_t.shape: torch.Size([1, 181200])

            ys.append(y_t)
            t -= 1
        return ys if store_intermediate_states else ys[-1]
```

Figure 40: diffusion_process.py (Vovk, 2020)

In this function, 'ys' is a list to store the audio signal in each dynamic iteration. 'y' is the stored current state, i.e. the audio signals for the current time step. These audio signals are generated by progressive denoising.

```
def compute_inverse_dynamics(self, mels, y, t, clip_denoised=True):
    pdb.set_trace()# 6 mels.shape: torch.Size([1, 80, 604]) y.shape: torch.Size([1, 181200]), t=5
    """
    Computes reverse (denoising) process dynamics. Closely related to the idea of Langevin dynamics.
    :param mels (torch.Tensor): mel-spectrograms acoustic features of shape [B, n_mels, T//hop_length]
    :param y (torch.Tensor): previous state from dynamics trajectory
    :param clip_denoised (bool, optional): clip signal to [-1, 1]
    :return (torch.Tensor): next state
    """
    model_mean, model_log_variance = self.p_mean_variance(mels, y, t, clip_denoised)
    eps = torch.randn_like(y) if t > 0 else torch.zeros_like(y)
    return model_mean + eps * (0.5 * model_log_variance).exp()
```

Figure 41: diffusion_process.py (Vovk, 2020)

```
def p_mean_variance(self, mels, y, t, clip_denoised: bool):
    """
    Computes Gaussian transitions of Markov chain at step t
    for further computation of y_{t-1} given current state y_t and features.
    """
    batch_size = mels.shape[0]
    noise_level = torch.FloatTensor([self.sqrt_alphas_cumprod_prev[t+1]]).repeat(batch_size, 1).to(mels)
    pdb.set_trace()# 7 mels.shape: torch.Size([1, 80, 604]) #  y.shape: torch.Size([1, 181200]), t=5

    eps_recon = self.nn(mels, y, noise_level)# eps_recon is the reconstructed noise
    pdb.set_trace()# 15 eps_recon.shape: torch.Size([1, 181200])

    # y_recon is the predicted initial signal computed from the given current state and the reconstructed noise.
    y_recon = self.predict_start_from_noise(y, t, eps_recon)
    pdb.set_trace()# 16 y_recon.shape: torch.Size([1, 181200])

    if clip_denoised:
        y_recon.clamp_(-1.0, 1.0)

    model_mean, posterior_log_variance = self.q_posterior(y_start=y_recon, y=y, t=t)
    pdb.set_trace()# 17 model_mean.shape: torch.Size([1, 181200]; posterior_log_variance_clipped: tensor(-4.5795, device='cuda:0')

    return model_mean, posterior_log_variance
```

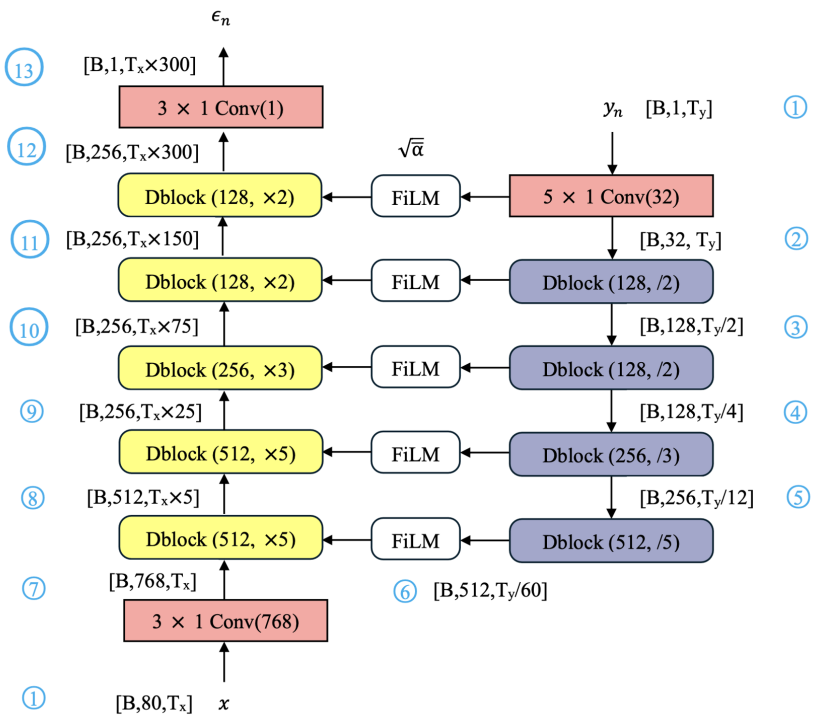
Figure 42: diffusion_process.py (Vovk, 2020)

C.4.1 The Process of Reconstructing Noise

```
def forward(self, mels, yn, noise_level): # x = mels yn = yn (1)
    """
    Computes forward pass of neural network.
    :param mels (torch.Tensor): mel-spectrogram acoustic features of shape [B, n_mels, T//hop_length]
    :param yn (torch.Tensor): noised signal `y_n` of shape [B, T]
    :param noise_level (float): level of noise added by diffusion
    :return (torch.Tensor): epsilon noise
    """
    # Prepare inputs
    assert len(mels.shape) == 3
    pdb.set_trace()# 8 mels.shape: torch.Size([1, 80, 604]); yn.shape: torch.Size([1, 181200])

    yn = yn.unsqueeze(1)
    pdb.set_trace()# 9 yn.shape: torch.Size([1, 1, 181200])
    assert len(yn.shape) == 3

    # Downsampling stream + Linear Modulation statistics calculation
    statistics = []
    dblock_outputs = self.dblock_preconv(yn) # (2)
    pdb.set_trace()# 10 dblock_outputs.shape: torch.Size([1, 32, 181200])

    scale, shift = self.films[0](x=dblock_outputs, noise_level=noise_level)
    statistics.append([scale, shift])
    for dblock, film in zip(self.dblocks, self.films[1:]):
        dblock_outputs = dblock(dblock_outputs) # (3) - (6)
        pdb.set_trace()# 11
        # dblock_outputs.shape:
        # torch.Size([1, 128, 90600]) --> torch.Size([1, 128, 45300]) -->
        # torch.Size([1, 256, 15100]) --> torch.Size([1, 512, 3020])

        scale, shift = film(x=dblock_outputs, noise_level=noise_level)
        statistics.append([scale, shift])
    statistics = statistics[::-1]

    # Upsampling stream
    ublock_outputs = self.ublock_preconv(mels) # (7)
    pdb.set_trace()# 12 ublock_outputs.shape: torch.Size([1, 768, 604])

    for i, ublock in enumerate(self.ublocks):
        scale, shift = statistics[i]
        ublock_outputs = ublock(x=ublock_outputs, scale=scale, shift=shift) # (8) - (12)
        pdb.set_trace()# 13
        # ublock_outputs.shape.shape: torch.Size([1, 512, 3020])]
        # --> torch.Size([1, 512, 15100]) --> torch.Size([1, 256, 45300])
        # --> torch.Size([1, 128, 90600]) --> torch.Size([1, 128, 181200])

    outputs = self.ublock_postconv(ublock_outputs) # (13)
    pdb.set_trace()# 14 outputs.shape: torch.Size([1, 1, 181200])
    # outputs.squeeze(1).shape: torch.Size([1, 181200])

    return outputs.squeeze(1)
```

Figure 43: nn.py (Vovk, 2020)

The numerical labels in parentheses in the above figure correspond to the sequence numbers in the following figure.



The last returned outputs. squeeze(1) is eps_recon.

```
def p_mean_variance(self, mels, y, t, clip_denoised: bool):
    """
    Computes Gaussian transitions of Markov chain at step t
    for further computation of y_{t-1} given current state y_t and features.
    """
    batch_size = mels.shape[0]
    noise_level = torch.FloatTensor([self.sqrt_alphas_cumprod_prev[t+1]]).repeat(batch_size, 1).to(mels)
    pdb.set_trace()# 7 mels.shape: torch.Size([1, 80, 604]) #  y.shape: torch.Size([1, 181200]), t=5

    eps_recon = self.nn(mels, y, noise_level) # eps_recon is the reconstructed noise
    pdb.set_trace()# 15 eps_recon.shape: torch.Size([1, 181200])

    # y_recon is the predicted initial signal computed from the given current state and the reconstructed noise.
    y_recon = self.predict_start_from_noise(y, t, eps_recon)
    pdb.set_trace()# 16 y_recon.shape: torch.Size([1, 181200])

    if clip_denoised:
        y_recon.clamp_(-1.0, 1.0)

    model_mean, posterior_log_variance = self.q_posterior(y_start=y_recon, y=y, t=t)
    pdb.set_trace()# 17 model_mean.shape: torch.Size([1, 181200]; posterior_log_variance.shape: ?

    return model_mean, posterior_log_variance
```

Figure 44: diffusion_process.py (Vovk, 2020)

```
def sample(self, mels, store_intermediate_states=False):
    """
    Samples speech waveform via progressive denoising of white noise with guidance of mels-epctrogram.
    :param mels (torch.Tensor): mel-spectrograms acoustic features of shape [B, n_mels, T//hop_length]
    :param store_intermediate_states (bool, optional): whether to store dynamics trajectory or not
    :return ys (list of torch.Tensor) (if store_intermediate_states=True)
        or y_0 (torch.Tensor): predicted signals on every dynamics iteration of shape [B, T]
    """
    with torch.no_grad():
        device = next(self.parameters()).device
        batch_size, T = mels.shape[0], mels.shape[-1]
        ys = [torch.randn(batch_size, T*self.total_factor, dtype=torch.float32).to(device)]
        t = self.n_iter - 1
        while t >= 0:
            pdb.set_trace() # 5 mels.shape: torch.Size([1, 80, 604])

            y_t = self.compute_inverse_dynamics(mels, y=ys[-1], t=t)
            pdb.set_trace() # 18 y_t.shape: torch.Size([1, 181200])

            ys.append(y_t)
            t -= 1
        return ys if store_intermediate_states else ys[-1]
```

Figure 45: diffusion_process.py (Vovk, 2020)

The returned value y_t is the audio signal at the next time step, and the model repeats this process until it generates a complete, perceptually similar audio waveform to the original.

The outputs are the synthesised speech. We can see that the shape of the final generated speech is the same as that of the input speech audio data at the beginning.

Synthesised Audio:

```
start = datetime.now()                                          # Record start time
outputs = model.forward(mel, store_intermediate_states=False)   # Perform model inference
pdb.set_trace() # 19 outputs.shape: torch.Size([1, 181200])
```

Figure 46: inference.py (Vovk, 2020)

Original Audio:

```
audio, sample_rate = torchaudio.load(audio_path)                # Load the audio data
pdb.set_trace() # 1 audio.shape: torch.Size([1, 154508])        # Pause execution for debugging
```

Figure 47: inference.py (Vovk, 2020)