Attempting Audio Synthesis Using Generative Adversarial Networks

Asia Brooks, Mama Sallah

CSCE A485

Dr. Masoumeh Heidari

December 8th, 2024

Introduction

For our project, we elected to focus on audio synthesis through the lens of machine vision. Audio synthesis is the process of generating sound electronically or through the process of manipulating or modulating existing samples. To this end, we attempt to use spectrograms, which are visual representations of audio data, to synthesize audio. A spectrogram's x-axis represents time, the y-axis represents frequency, and the intensity, brightness, or just the color of a signal depicted in a spectrogram reflects the amplitude of the frequency at any given point in time. Technically, because of the time-frequency relationship between pixels which hold semantic meaning, spectrograms are not your traditional images in a machine vision context. Nevertheless, there is still an established framework for image-generation and audio synthesis using spectrograms, which revolves around Generative Adversarial Networks (GANs).

GANs are machine learning models commonly used for image generation tasks. Their basic structure consists of two neural networks: the generator and the discriminator, which are trained together in a competitive framework. The generator, using a starting random seed and upsampling layers, creates fake data (in our case, fake spectrograms), while the discriminator, which essentially has the real data on hand, attempts to distinguish between real and generated data. The discriminator can ascribe generated images a true/false score or confidence rating, indicating the likelihood of that generated image passing for the real thing, and this evaluation is kicked back to the generator as feedback during the training process.

The trick with GANs is that they are a very careful balancing act, where we do not want either the generator or the discriminator to overly dominate the other, and instead give the generator the necessary room or time to explore possible features. Through the adversarial

process, the generator should learn to produce data that closely resembles real data. In our project, we attempted to apply this approach for generating spectrograms using an existing dataset of real spectrograms, and then converting these generated spectrograms back into audio waves and comparing the audio samples in quality. In an abstract way, we wanted the model to be able to “read” sheet music in the form of a spectrogram, reproduce the sheet music, and then be able to “play” it afterward.

Dataset

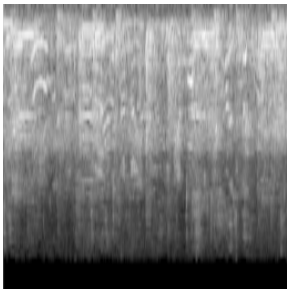
For this project we use the UrbanSound8K dataset, which consists of 8,732 labeled sound excerpts from 10 urban sound classes. These classes cover a wide range of noise types, including street air conditioners, car horns, children playing, dog barks, drilling, engine idling, gunshots, jackhammers, sirens, and street music. We arbitrarily elected to use the street music class of .wav files because it features live music being performed in many different settings, while still containing some ambient noise like traffic or human chatter and singing that could inject some added challenge in the audio synthesis process. These specific street music audio samples are a 1,000 total, and range from 126 KB to 2251 KB in size, and are cut to a standard 4 secs maximum in length.

The data pre-processing phase mostly consisted of just scraping/compiling the desired files from the UrbanSound8K dataset, and then converting them into mel spectrograms, which would be used as our ‘real’ images in the GAN. The conversion process was done referencing Ketan Doshi’s 2021 article “Audio Deep Learning Made Simple: Sound Classification, Step-by-Step”, albeit using Tensorflow rather than Pytorch. This process included defining a set of parameters such as sample rate, mel channels, hop length, but also implementing a time shift,

and rechanneling and truncating the audio. Pre-processing resulted in a dataset of 1000 (256 x 256) grayscale spectrograms, each matched with a corresponding audio file from the dataset. Below we can see some excerpts from the pre-processing script, as well as a typical example of what kind of spectrograms it produced.

```
for file in os.listdir(spectrogram_dir):
    file_path = os.path.join(spectrogram_dir, file)
    if os.path.isfile(file_path):
        os.remove(file_path)

print(f"Cleared old spectrograms in {spectrogram_dir}")


# Spectrogram settings
target_sample_rate = 22050
n_mels = 64
hop_length = 512
n_fft = 1024
max_duration_ms = 4000  # 4 seconds
time_shift_limit = 0.4  # Percentage of sample length, totals to 5.6
resize_shape = (256, 256)  # Target resize shape (height, width)

# Function to rechannel audio
def rechannel_audio(y, sr, target_channels=2):  1 usage
    if len(y.shape) == 1 and target_channels == 2:
        # Mono to stereo by duplicating the single channel
        y = np.stack([y, y])
    elif len(y.shape) == 2 and y.shape[0] == 2 and target_channels == 1:
        # Stereo to mono by selecting the first channel
        y = y[0]
    return y

# Function to resample audio
def resample_audio(y, orig_sr, target_sr):  1 usage
    if orig_sr != target_sr:
        y = librosa.resample(y, orig_sr=orig_sr, target_sr=target_sr)
    return y

# Function to pad or truncate audio
def pad_truncate_audio(y, sr, max_duration_ms):  1 usage
    max_len = int(sr * (max_duration_ms / 1000))  # Maximum length in samples
    sig_len = y.shape[0]
```

```
# Save spectrogram
fig, ax = plt.subplots()
librosa.display.specshow(S_dB_resized, sr=target_sample_rate, hop_length=hop_length, x_axis='time', y_axis='mel', ax=ax)
plt.axis('off')
spectrogram_path = os.path.join(spectrogram_dir, f"{file_name.replace( _old: '.wav',  _new: '')}_spectrogram.png")
# Save resized spectrograms as grayscale images
S_dB_resized = tf.image.resize(S_dB[..., np.newaxis], resize_shape, method='bilinear').numpy()
tf.keras.utils.save_img(spectrogram_path, S_dB_resized, scale=True)

plt.close(fig)

print(f"Generated and resized spectrogram for {file_name} at {spectrogram_path}")
```



Past Implementations

This project was inspired by a core set of GAN implementations that I researched before diving into implementation myself and still continuously consulted with during development. These works consist of:

- GANSynth: Adversarial Neural Audio Synthesis by Engel et al, 2019
    - GANSynth focuses on the synthesis of musical notes from various instruments. Its creators leveraged the NSynth dataset, which includes an extensive library of tones, chords, and timbres across a wide range of acoustic and synthesized instruments, and it demonstrated the ability to generate high-quality audio with rich harmonic detail.
- MelGAN: Generative Adversarial Networks for Conditional Waveform Synthesis by Kumar et al, 2019
    - Focused on natural datasets such as speech synthesis and birdsong. Being relatively lightweight, I was interested in attempting to mimic their approach.
- WaveGAN/SpecGAN: Adversarial Audio Synthesis by Donahue et al, 2019
    - WaveGAN and SpecGAN are two dual models developed concurrently for two different inputs, one using raw waveforms and the other spectrograms. These models are tailored for synthesizing audio from spectrograms. SpecGAN's reliance on spectrogram input and easily accessible documentation meant it became something of a primary reference for me.

Each of these works make extended use of unsupervised learning GAN architecture to generate audio with the ultimate goal of generating samples that are indistinguishable from the real. They have varying model architectures and datasets, but still have similar goals and offer

some insights. Broadly speaking, GANSynth worked as early inspiration for the first half of development because of its explicit focus on music, with MelGAN and then especially SpecGAN acting as the primary inspiration for me during the latter half of development.

Attempted Methods and Breakdown of Current Methodology

As mentioned earlier, GANSynth was a large inspiration for my early efforts, but I don't actually have much in the way of code for this method. GANsynth was built using the Magenta library for Python, and despite Magenta's amount of accolades and talent, it's no longer maintained or updated. Even many of the Google Colab notebooks the team made for testing and familiarity purposes were broken or unavailable. Early efforts to try and use GANSynth mostly consisted of just trying to hassle with configuration both in Google Colab and with Pycharm on my own device, but due to hardcode incompatibilities, I was unable to fully utilize Magenta despite attempting and utilizing numerous workarounds, and was forced to migrate to using the easier and more current Tensorflow. This learning process took up an unfortunate amount of time, and I think had I been less stubborn in wanting to use Magenta (or more clever in trying to set it up), I could have saved a substantial amount of time.

After migrating to Tensorflow, I created what would partially become a template for future iterations. This early Tensorflow model was my attempt at creating a conditional generator rather than a classic GAN, because to me conditional GAN more naturally fit how I had originally envisioned the pipeline/process of a GAN. Below is my conditional generator:

```
# Directory paths for spectrograms and generated audio
real_spectrograms_dir = r'C:\Users\khafa\Downloads\UrbanSound8K\UrbanSound8K\spectrograms'
output_audio_dir = r'C:\Users\khafa\Downloads\UrbanSound8K\UrbanSound8K\generated_audio'
os.makedirs(output_audio_dir, exist_ok=True)

# Model parameters
latent_dim = 64  # Latent dimension for generator input
spectrogram_shape = (256, 256, 4)
sample_rate = 22050

def load_spectrograms(target_shape):
    spectrograms = []
    for filename in os.listdir(real_spectrograms_dir):
        if filename.endswith('.png'):
            img = plt.imread(os.path.join(real_spectrograms_dir, filename))
            img_resized = tf.image.resize(img[..., :1], target_shape[:2])
            spectrograms.append(img_resized)
    return np.array(spectrograms) / 255.0

real_spectrograms = load_spectrograms(spectrogram_shape)

# Conditional Generator Model
def build_conditional_generator(latent_dim, output_shape):
    noise_input = layers.Input(shape=(latent_dim,))
    spectrogram_input = layers.Input(shape=output_shape)

    x = layers.Concatenate()([noise_input, layers.Flatten()(spectrogram_input)])
    x = layers.Dense(512, activation='relu')(x)
    x = layers.Reshape((8, 8, 8))(x)
    x = layers.Conv2DTranspose(256, kernel_size=4, strides=2, padding='same', activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2DTranspose(128, kernel_size=4, strides=2, padding='same', activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2DTranspose(64, kernel_size=4, strides=2, padding='same', activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv2DTranspose(1, kernel_size=4, strides=2, padding='same', activation='sigmoid')(x)
    output = layers.Reshape(output_shape)(x)

    return Model([noise_input, spectrogram_input], output)
```

For the conditional generator I had to use a manual concatenation approach, which took some getting used to and I was not a fan of it. My model was very basic and I had issues handling the shape of the spectrogram during this implementation, and while I do not have any surviving or saved output from this model, I recall that it produced either stark white or stark black images, reflecting model collapse, and the audio it produced was unpleasant. My sense of experimentation at this point was limited to just adjusting the number of layers in the generator and discriminator, batch size, as well as altering the learning rate for both networks, so early

efforts were stalled by my own inexperience and also because I had yet to properly configure my GPU optimize my training speeds.

```
# Discriminator Model
def build_discriminator(input_shape):
    model = Sequential([
        layers.Input(shape=input_shape),
        layers.Conv2D(64, kernel_size=4, strides=2, padding='same'),
        layers.LeakyReLU(alpha=0.2),
        layers.Dropout(0.3),
        layers.Conv2D(128, kernel_size=4, strides=2, padding='same'),
        layers.LeakyReLU(alpha=0.2),
        layers.Dropout(0.3),
        layers.Flatten(),
        layers.Dense(1, activation='sigmoid')
    ])
    return model

generator = build_conditional_generator(latent_dim, spectrogram_shape)
discriminator = build_discriminator(spectrogram_shape)

generator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0004)
discriminator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0004)
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

In the weeks leading up to and after the project proposal I did much more experimentation and began to implement more changes and experiment more. I migrated away from the conditional generator format because I wanted to follow more established and conventional methods. Concurrent to this, I became aware of the checker-board pattern and similar artifacts that could affect my generated images' quality, and tried to adapt the phase-shuffle method that WaveGAN authors mentioned in their paper (Page 4) and utilized for their model. After not getting much better results, I then swapped this out for the more apt 'CustomConv2DTranspose' function they utilized specifically in SpecGAN:

```python
#borrowed from SpecGAN
class CustomConv2DTranspose(Layer):
    def __init__(self, filters, kernel_size, strides=2, padding='same', upsample='zeros', **kwargs):
        super(CustomConv2DTranspose, self).__init__(**kwargs)
        self.filters = filters
        self.kernel_size = kernel_size
        self.strides = strides
        self.padding = padding
        self.upsample = upsample

        if self.upsample == 'zeros':
            self.conv_transpose = Conv2DTranspose(filters, kernel_size, strides=strides, padding=padding)
        else:
            self.conv = Conv2D(filters, kernel_size, strides=1, padding=padding)

    def call(self, inputs):
        if self.upsample == 'zeros':
            return self.conv_transpose(inputs)
        else:
            if self.upsample == 'nn':
                method = ResizeMethod.NEAREST_NEIGHBOR
            elif self.upsample == 'linear':
                method = ResizeMethod.BILINEAR
            elif self.upsample == 'cubic':
                method = ResizeMethod.BICUBIC
            else:
                raise ValueError(f"Unsupported upsampling method: {self.upsample}")

            # Upsample the input
            upsampled = resize(inputs, [inputs.shape[1] * self.strides, inputs.shape[2] * self.strides], method=method)
            # Apply convolution
            return self.conv(upsampled)

    def get_config(self):
        config = super(CustomConv2DTranspose, self).get_config()
        config.update({
            'filters': self.filters,
            'kernel_size': self.kernel_size,
            'strides': self.strides,
            'padding': self.padding,
            'upsample': self.upsample
        })
        return config
```

This allowed for greater flexibility and for me to tailor what kind of upsampling techniques I could utilize to reduce artifacts. This, coupled with some refinement in the preprocessing phase (increasing their size to 256 x 256 from 128 x 128) and increasing my latent dim value to a standard 128 meant I could now reliably generate less pixelated and less starkly colored images. Below is current configuration for the generator and discriminator:

```python
def generator(latent_dim, output_shape):
    model = Sequential([
        # Input layer: Dense + Reshape
        layers.Dense(16 * 16 * latent_dim * 8, activation='relu', input_dim=latent_dim),
        layers.Reshape((16, 16, latent_dim * 8)),

        # Layer 1
        CustomConv2DTranspose(latent_dim * 4, kernel_size=7, strides=2, upsample='linear'),
        layers.BatchNormalization(),
        layers.ReLU(),

        # Layer 2
        CustomConv2DTranspose(latent_dim * 2, kernel_size=5, strides=2, upsample='linear'),
        layers.BatchNormalization(),
        layers.ReLU(),

        # Layer 3
        CustomConv2DTranspose(latent_dim, kernel_size=5, strides=2, upsample='cubic'),
        layers.BatchNormalization(),
        layers.ReLU(),

        # Layer 4 (ensure shape is 256 by 256)
        CustomConv2DTranspose(latent_dim // 2, kernel_size=3, strides=2, upsample='cubic'),
        layers.BatchNormalization(),
        layers.ReLU(),

        # Output Layer
        CustomConv2DTranspose(output_shape[-1], kernel_size=1, strides=1, upsample='nn'),
        layers.Activation('tanh'),
    ])
    return model

# Discriminator Model
def build_discriminator(input_shape):
    model = Sequential([
        layers.Input(shape=input_shape),
        layers.Conv2D(64, kernel_size=7, strides=2, padding='same'),
        layers.LeakyReLU(alpha=0.2),
        layers.Dropout(0.3),

        layers.Conv2D(128, kernel_size=5, strides=2, padding='same'),
        layers.BatchNormalization(),
        layers.LeakyReLU(alpha=0.2),
        layers.Dropout(0.3),

        layers.Conv2D(256, kernel_size=3, strides=2, padding='same'),
        layers.BatchNormalization(),
        layers.LeakyReLU(alpha=0.1),
        layers.Dropout(0.3),

        layers.Conv2D(512, kernel_size=1, strides=1, padding='same'),
        layers.BatchNormalization(),
        layers.LeakyReLU(alpha=0.1),
        layers.Flatten(),

        layers.Dense(1, activation='sigmoid')
    ])
    return model
```

I experimented frequently with balancing different upsampling methods, and the approach I settle on was to try and capture large patterns using linear upsampling and a larger kernel size and stride, and then drill down using cubic and nearest neighbor with smaller kernel sizes and strides, to try and find finer details. I implemented a somewhat similar method in the

discriminator (descending value of kernel size and strides), but also implemented LeakyReLU and Dropout to try and prevent overfitting and reduce discriminator dominance. My generator uses a standard tanh activation function, and the discriminator uses a standard and intuitive sigmoid activation function to make calls on a generated image's authenticity.

With more frequent experimentation and a more complex model, I learned that I had to reduce both batch size, when I had initially used values as high as 32 in prior iterations but would now only have to use eight, but also shrink the dataset itself, using only 50-100 samples. This was because of both the exorbitant amount of time that training 1000 epochs would take before any kind of structure might form, but also because of memory limitations with my GPU.

```
# Reduce the dataset size
target_size = 100  # Number of samples to keep, we can adjust as needed
np.random.seed(42)
selected_indices = np.random.choice(len(real_spectrograms), size=target_size, replace=False)
real_spectrograms = real_spectrograms[selected_indices]
```

Maybe because of these limitations in batch size and sample size, the model experienced a lot of infrequency and instability both confidence and loss wize for both the generator and the discriminator, with values oscillating frequently. To attempt to counter this, I attempted to crank the value of my lambda_gp to control the strength of the gradient penalty term, which could in turn restrain the discriminator. In addition, I attempted label smoothing for real samples to prevent overconfidence and promote generalization, dynamic weighing for the different real and fake losses, and gradient clipping to prevent overfitting.

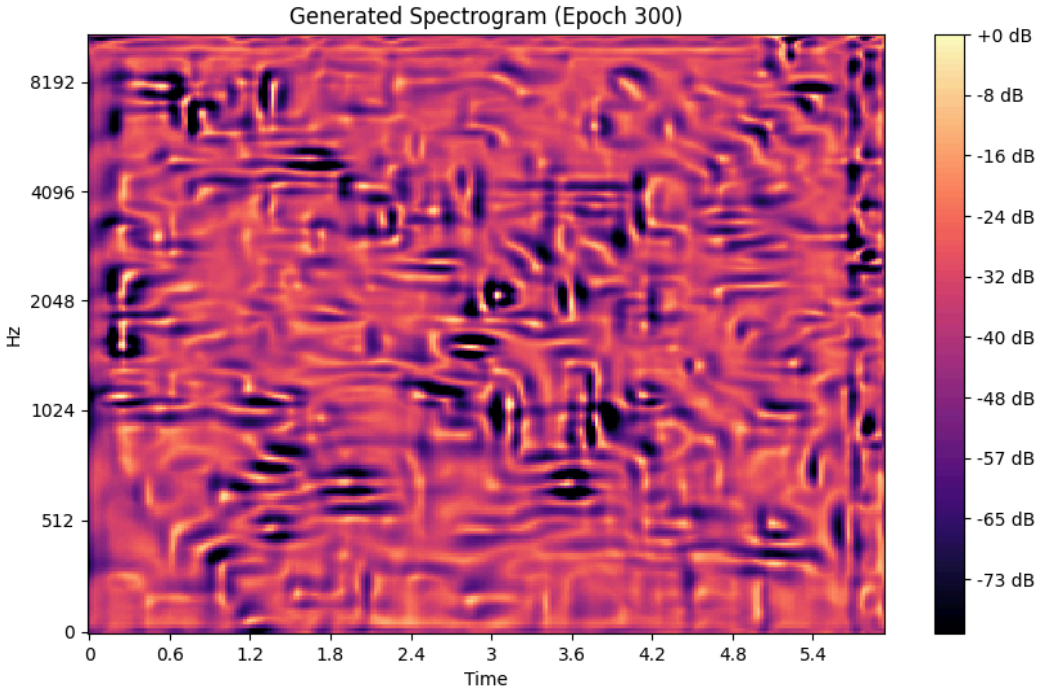
```python
# Custom training step with additional supervised loss for exact matching
@tf.function
def train_step(real_spectrograms, generator_updates=2, lambda_gp=25.0):
    batch_size = real_spectrograms.shape[0]
    noise = tf.random.normal([batch_size, latent_dim])

    # Train discriminator
    with tf.GradientTape() as disc_tape:
        real_spectrograms_noisy = real_spectrograms + tf.random.normal(real_spectrograms.shape, mean=0.0, stddev=0.02)
        generated_spectrograms = generator(noise, training=True)
        generated_spectrograms_noisy = generated_spectrograms #+ tf.random.normal(generated_spectrograms.shape, mean=0.0, stddev=0.01)
        #no additional noise need for the generated_spectrograms, might be doubling up
        #and harming the model
        real_output = discriminator(real_spectrograms_noisy, training=True)
        fake_output = discriminator(generated_spectrograms_noisy, training=True)

        real_labels = tf.ones_like(real_output) * 0.9  # Smoothing
        fake_labels = tf.zeros_like(fake_output)

        real_loss = cross_entropy(real_labels, real_output)
        fake_loss = cross_entropy(fake_labels, fake_output)

        # Dynamic weighting
        real_weight = 1.0 - tf.reduce_mean(real_output)
        fake_weight = tf.reduce_mean(fake_output)

        disc_loss = real_weight * real_loss + fake_weight * fake_loss

        # Gradient penalty computation
        alpha = tf.random.uniform([batch_size, 1, 1, 1], minval=0., maxval=1.)
        mixed_images = alpha * real_spectrograms + (1 - alpha) * generated_spectrograms
        with tf.GradientTape() as gp_tape:
            gp_tape.watch(mixed_images)
            mixed_output = discriminator(mixed_images, training=True)
        gradients = gp_tape.gradient(mixed_output, [mixed_images])[0]
        gradients_norm = tf.sqrt(tf.reduce_sum(tf.square(gradients), axis=[1, 2, 3]))
        gradient_penalty = tf.reduce_mean(tf.square(tf.maximum(gradients_norm - 1.0, 0)))

        # Add gradient penalty to discriminator loss
        disc_loss += lambda_gp * gradient_penalty

    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)
    clipped_gradients = [tf.clip_by_value(grad, -0.5, 0.5) for grad in gradients_of_discriminator]
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))
```

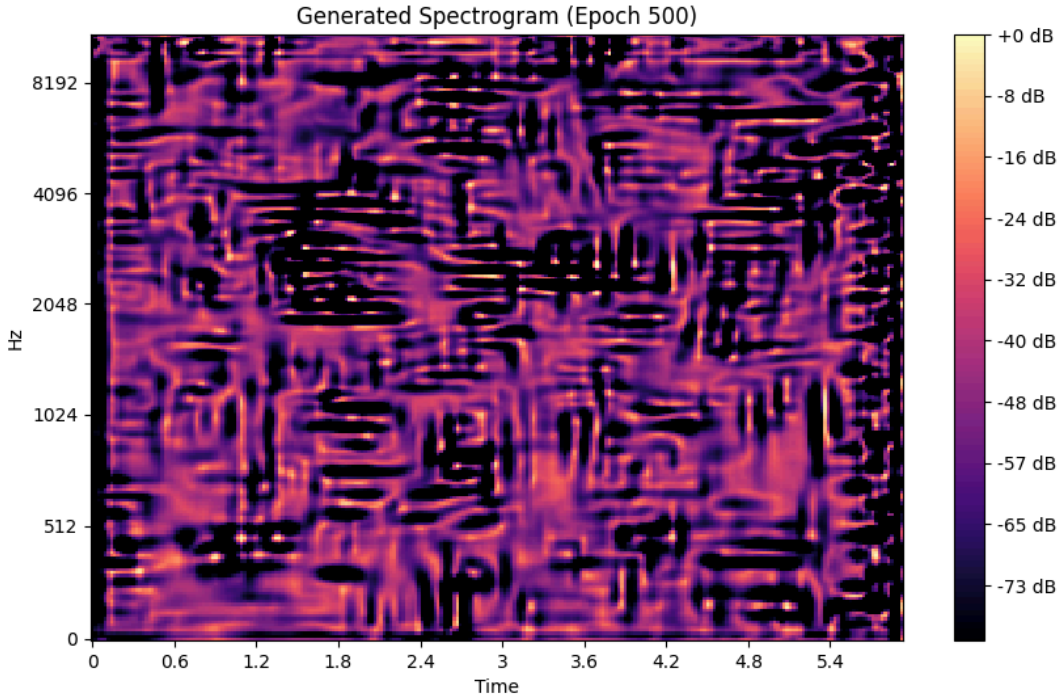
Last thing I want to point out is my visualization section, which was obviously handy and critical for me to evaluate how the model was performing. In most instances I only wanted the model to produce a generated spectrogram every ten epochs so I could closely follow its progress without being too inundated with samples.

```python
# Visualize and save generated spectrograms for specific epochs
if epoch == 0 or (epoch + 1) % 10 == 0:
    noise = tf.random.normal([batch_size, latent_dim])
    generated_spectrogram = generator.predict(noise)[0]  # Generate one spectrogram

    # Rescale from [-1, 1] to [0, 1]
    generated_spectrogram_rescaled = (generated_spectrogram + 1.0) / 2.0

    # Convert to power spectrogram (intensity approximation)
    power_spectrogram = generated_spectrogram_rescaled ** 2

    # Convert to dB scale for visualization
    spectrogram_db = librosa.power_to_db(power_spectrogram.squeeze(), ref=np.max)

    # Save the spectrogram as an image
    epoch_dir = os.path.join(genspec_dir, f"epoch_{epoch + 1}")
    os.makedirs(epoch_dir, exist_ok=True)
    spectrogram_path = os.path.join(epoch_dir, f"generated_spectrogram_{epoch + 1}.png")

    plt.figure(figsize=(10, 6))
    librosa.display.specshow(
        spectrogram_db,
        sr=sample_rate,
        hop_length=512,
        x_axis="time",
        y_axis="mel",
    )
    plt.colorbar(format="%+2.0f dB", boundaries=np.linspace(-80, 0, 100))
    plt.title(f"Generated Spectrogram (Epoch {epoch + 1})")
    plt.savefig(spectrogram_path)
    plt.close()

# Print epoch results
print(f"Epoch {epoch + 1}, Discriminator Loss: {d_loss:.6f}, Generator Loss: {g_loss:.6f}, "
      f"Real Confidence: {real_confidence:.6f}, Fake Confidence: {fake_confidence:.6f}, "
      f"RMSE: {epoch_rmse:.6f}")
```

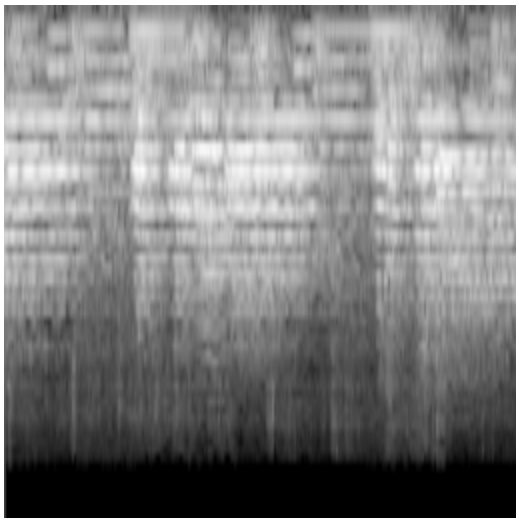
Results

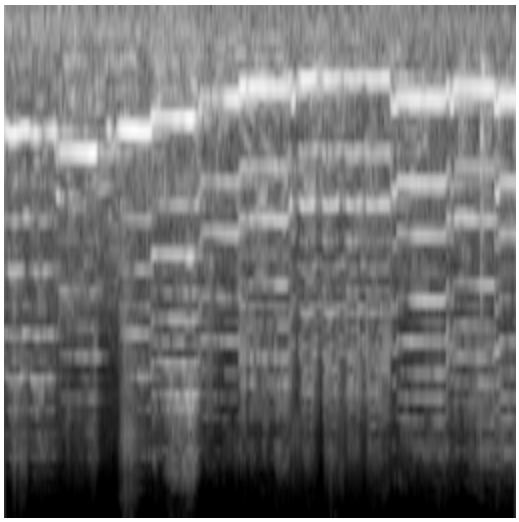
To be blunt, my results were not great both heuristically and objectively. For one, the generated spectrograms could not really be identifiably labeled spectrograms in most instances. Experimentation with upsampling methods and other parameters did yield a bevy of interesting images that were indicative of some kind of training and learning going on, but did not ever converge. For instance, we can look at a series of of images which I deemed the maze series during my experimentation:


Generated Spectrogram (Epoch 300)
Hz
8192
4096
2048
1024
512
0
0
0.6
1.2
1.8
2.4
3
3.6
4.2
4.8
5.4
Time
+0 dB
-8 dB
-16 dB
-24 dB
-32 dB
-40 dB
-48 dB
-57 dB
-65 dB
-73 dB

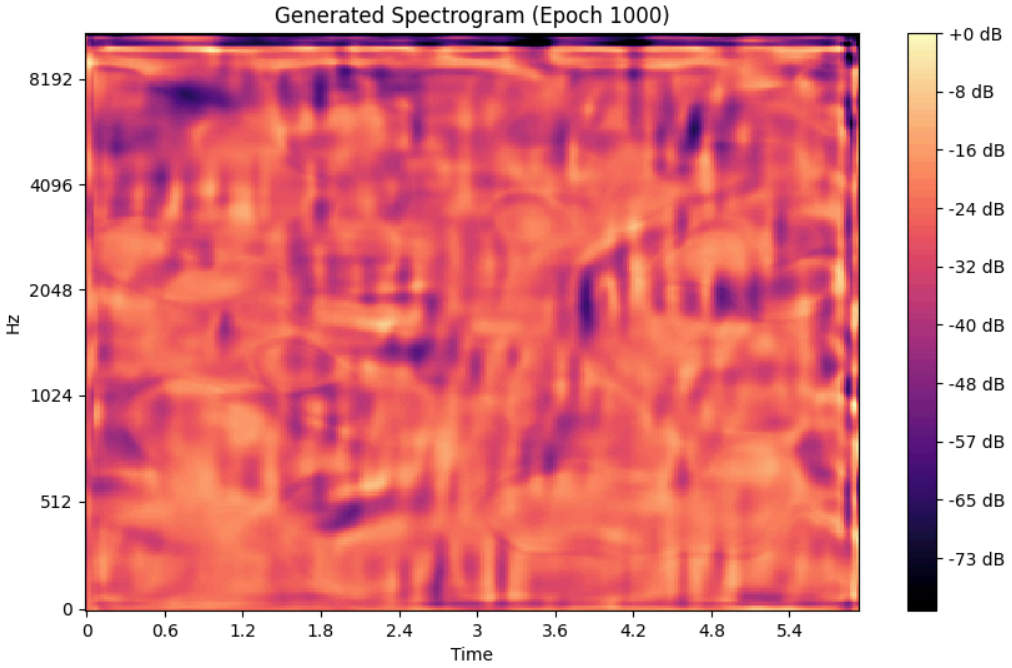

Generated Spectrogram (Epoch 500)
Hz
8192
4096
2048
1024
512
0
0
0.6
1.2
1.8
2.4
3
3.6
4.2
4.8
5.4
Time
+0 dB
-8 dB
-16 dB
-24 dB
-32 dB
-40 dB
-48 dB
-57 dB
-65 dB
-73 dB

Some things that we can observe is the clear lean towards taking on a sort of structured pattern, reminiscent of a maze, which I believe are patterned after the kinds of horizontal and vertical streaks we'd expect to see in a standard spectrogram, though taken to an obvious extreme.

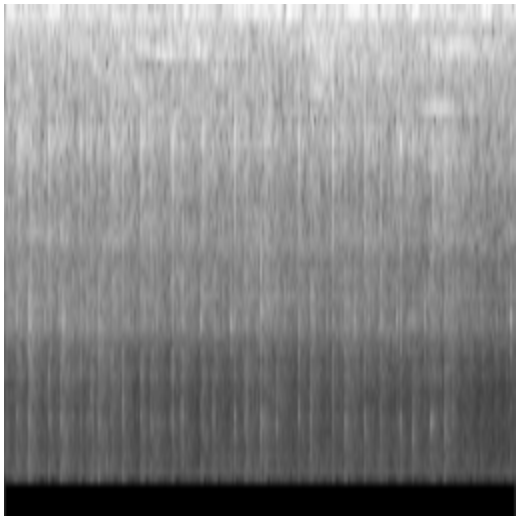


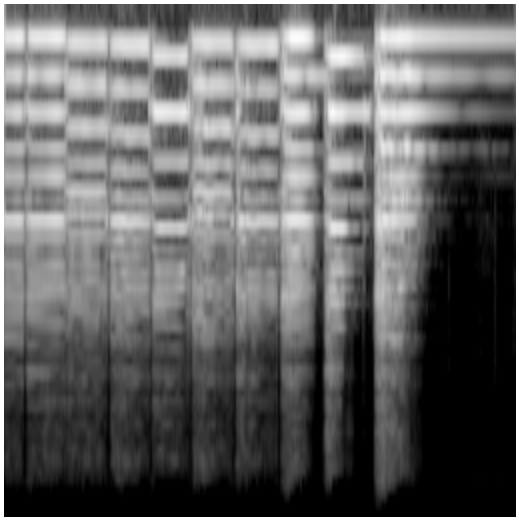


The blurred sections present in the generated spectrograms make recurring appearances throughout the epochs, and seem to be an error in the convolution and upsampling process, and when they're more pronounced, I've taken to calling it a sort of swiss cheese pattern or artifact.

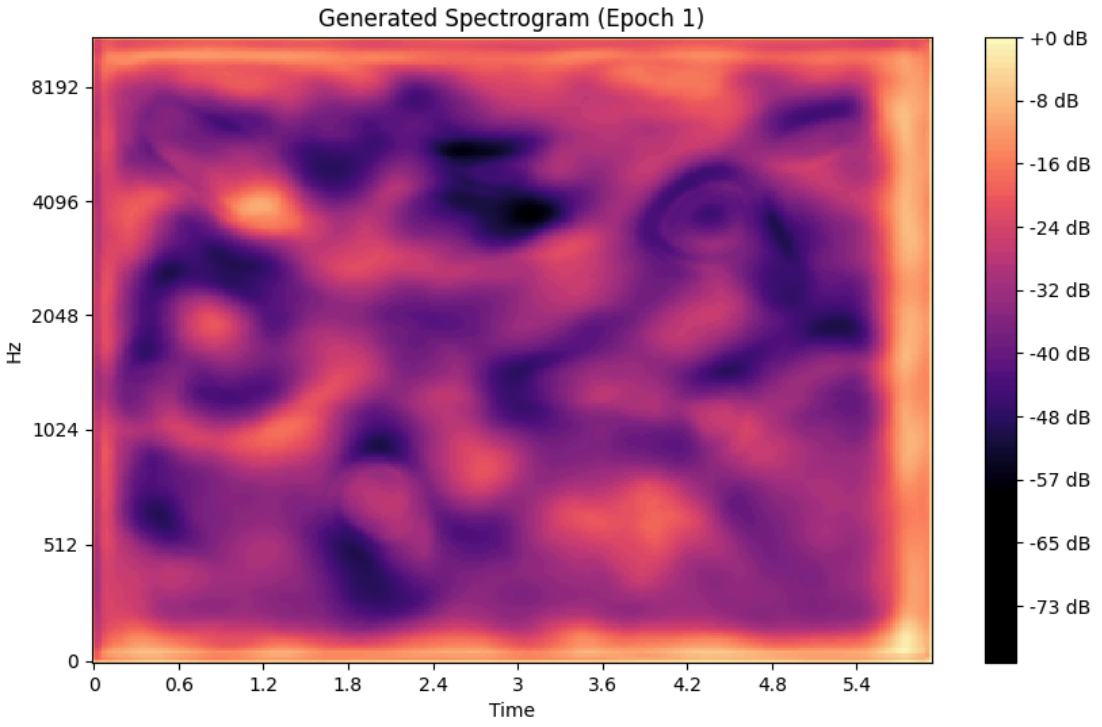


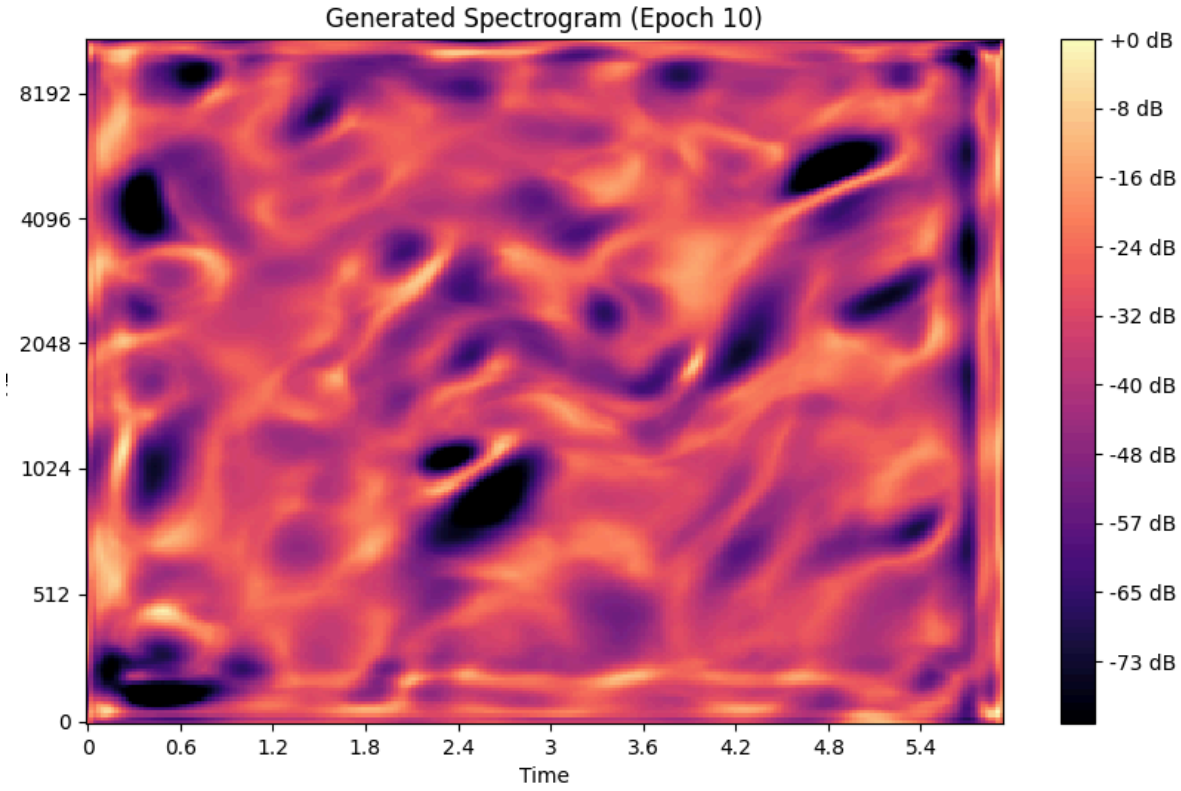
The above sample, generated from my current model, has some of the blurred sections as I mentioned, but also shows clear (as clear as it can be) striation, somewhat mimicking a real spectrogram's appearance. What's also of note is the rectangular border around the image, sometimes with a dark band. This feature pops frequently in the generated specs, and seems to reflect the tendency for the sides of a spectrogram to be pitch black, representing an absence of frequencies or silence.



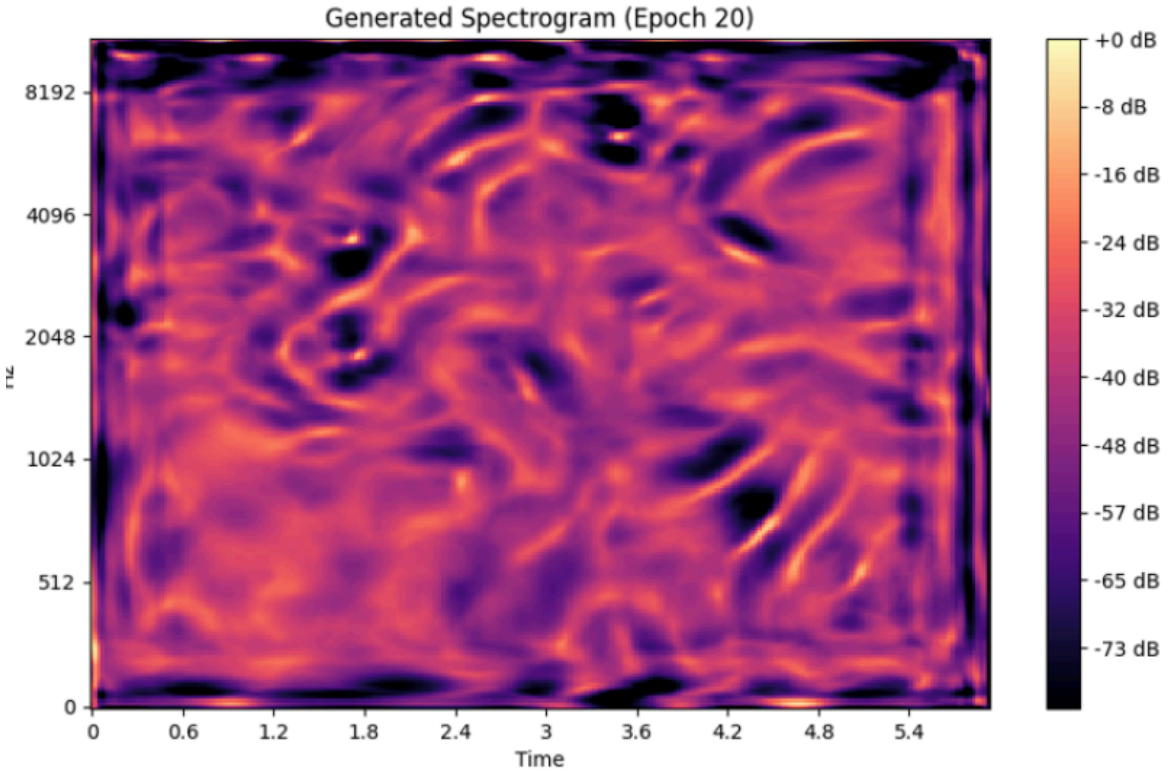


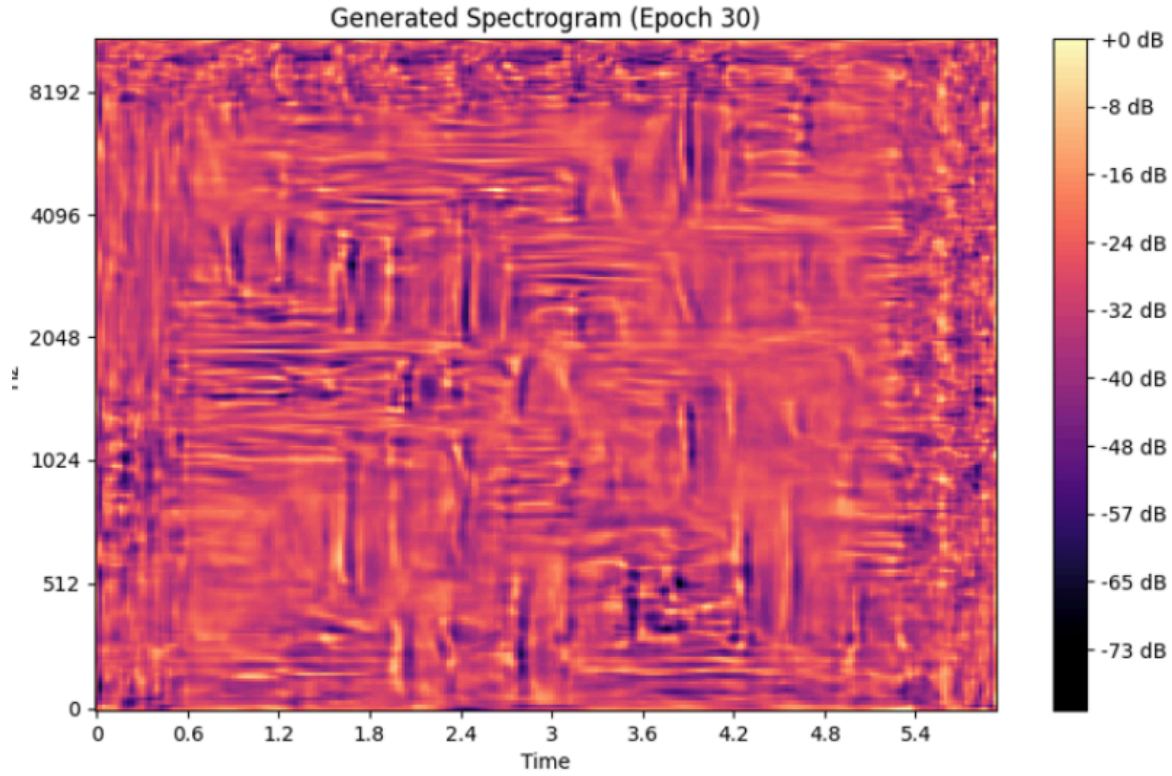
To better demonstrate what the training process looks like, here is a series of generated spectrograms from an implementation of my model that utilized much more bicubic upsampling.

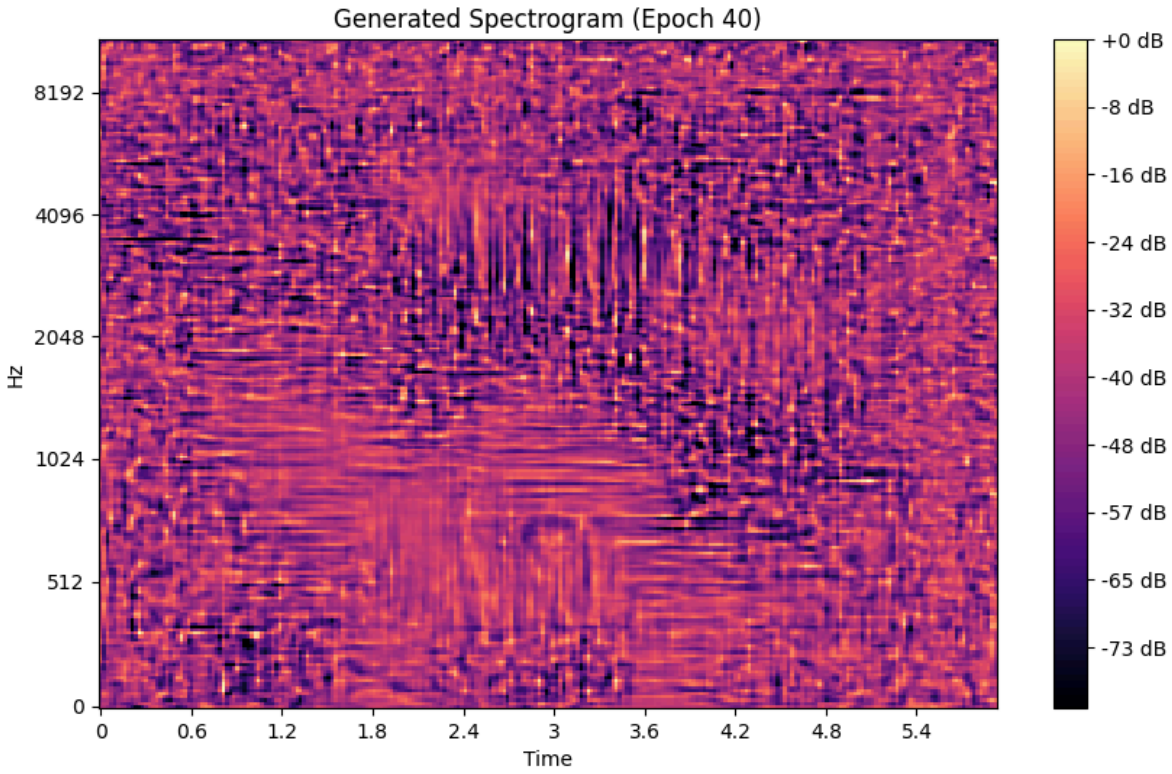


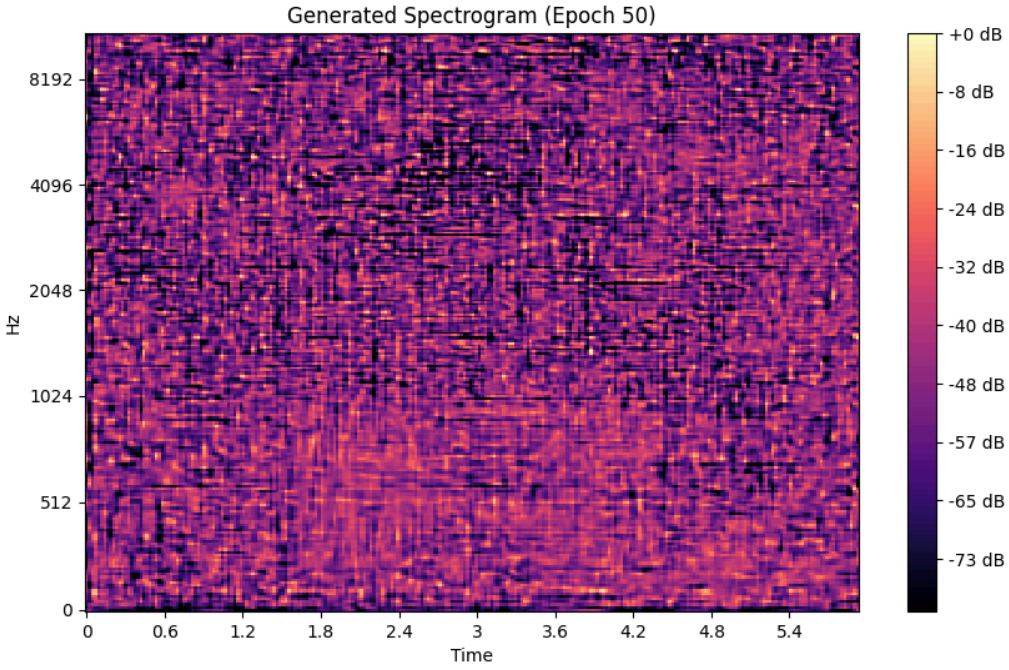


Note the “Swiss Cheese” pattern I mentioned earlier, with negative decibel holes.


Generated Spectrogram (Epoch 20)
8192
4096
2048
1024
512
0
0
0.6
1.2
1.8
2.4
3
3.6
4.2
4.8
5.4
Time
+0 dB
-8 dB
-16 dB
-24 dB
-32 dB
-40 dB
-48 dB
-57 dB
-65 dB
-73 dB


Generated Spectrogram (Epoch 30)
8192
4096
2048
1024
512
0
0
0.6
1.2
1.8
2.4
3
3.6
4.2
4.8
5.4
Time
+0 dB
-8 dB
-16 dB
-24 dB
-32 dB
-40 dB
-48 dB
-57 dB
-65 dB
-73 dB


Generated Spectrogram (Epoch 40)
8192
4096
2048
1024
512
0
Hz
0
0.6
1.2
1.8
2.4
3
3.6
4.2
4.8
5.4
Time
+0 dB
-8 dB
-16 dB
-24 dB
-32 dB
-40 dB
-48 dB
-57 dB
-65 dB
-73 dB


Generated Spectrogram (Epoch 50)
8192
4096
2048
1024
512
0
Hz
0
0.6
1.2
1.8
2.4
3
3.6
4.2
4.8
5.4
Time
+0 dB
-8 dB
-16 dB
-24 dB
-32 dB
-40 dB
-48 dB
-57 dB
-65 dB
-73 dB

Heuristically speaking, while these images are visually interesting, they are not identifiable as spectrograms really, and on top of that, even though there aren't clear checkerboarding artifacts in the images, the audio they generate is not pleasant to listen to. To somewhat make up for the disappointing heuristic data, I decided to implement gathering RMSE and Frechet Instance Distance (FID) for some objective metrics. RMSE measures pixel level accuracy and the average difference between the pixel values of the real images and the generated images, giving an indication of how accurately the GAN can reproduce the real images on a pixel-by-pixel basis. The FID is concerned with feature level and perceptual level quality, measuring and comparing the distribution of features between the two datasets and just how well the generated images actually pass.

Using a couple of scripts, with the FID script using a pre-trained InceptionV3 model, I measured the RMSE and FID of my generated images. Surprisingly, the FID script produced an average of 21.57; when a perfect FID score is 0, this would indicate that the spectrograms are actually of a good quality. The RMSE average of 0.5487 on a tanh scale tells a more believable story, indicating that there's substantial deviation from the norm.

Conclusion

I think overall while the model did not perform according to my wishes, I did make a lot of progress and learn quite a bit about how GANs work and how much experimentation and training is required to get them to converge. I think in my model's case, the issue could come down to additional layers being necessary, different upsampling methods, kernel sizes, strides length, or just finding a way to stably increase the batch size and training size without impeding responsiveness or adding an exorbitant amount of time to training. I think that the dataset itself could be partly an issue, as many of the samples already had noise or a great amount of

uniqueness and variety that could confuse the model, and would require additional methods to preprocess the dataset. I was encouraged by the emergence of structures in the generated images, like the bands, borders, and maze-like patterns present in some of the samples, because these indicated some learning and exploration of the feature and latent dimensions was taking place. I would like to continue the project in the future, with hopefully more time and experience to better guide what path forward I can take.

References

1. Descriptinc. “Descriptinc/Melgan-Neurips: Gan-Based Mel-Spectrogram Inversion Network for Text-to-Speech Synthesis.” GitHub, 5 Dec. 2019, github.com/descriptinc/melgan-neurips.
2. Donahue, Chris, et al. “Adversarial Audio Synthesis.” arXiv.Org, 9 Feb. 2019, arxiv.org/abs/1802.04208.
3. Donahue, Chris. “Chrisdonahue/Wavegan: Wavegan: Learn to Synthesize Raw Audio with Generative Adversarial Networks.” GitHub, 27 Nov. 2022, github.com/chrisdonahue/wavegan/tree/master.
4. Doshi, Ketan. “Audio Deep Learning Made Simple: Sound Classification, Step-by-Step.” Medium, Towards Data Science, 21 May 2021, towardsdatascience.com/audio-deep-learning-made-simple-sound-classification-step-by-step-cebc936bbe5.
5. Engel, Jesse, et al. “Gansynth: Adversarial Neural Audio Synthesis.” arXiv.Org, 15 Apr. 2019, arxiv.org/abs/1902.08710.
6. Kumar, Kundan, et al. “Melgan: Generative Adversarial Networks for Conditional Waveform Synthesis.” arXiv.Org, 9 Dec. 2019, arxiv.org/abs/1910.06711.
7. Salamon, Justin, et al. “UrbanSound8K.” Urban Sound Datasets, 2014, urbansounddataset.weebly.com/urbansound8k.html?c=mkt_w_chnl%3Aaff_geo%3Aall_prtnr%3Asas_subprtnr%3A1538097_camp%3Abrand_adtype%3Atxtlnk_ag%3Aweebly_lptype%3Ahp_var%3A358504&sscid=c1k8_h8wiq&utm_source=ShareASale.

8. Team, Magenta. “Magenta/Magenta/Models/Gansynth/Readme.Md at Main · Magenta/Magenta.” GitHub, 30 June 2021, github.com/magenta/magenta/blob/main/magenta/models/gansynth/README.md.