CSE 5523 Final Project
Music Gneration With Generative Adversarial Network

Pavani Komati, Soham Mukherjee, and Rupen Mitra

December 11, 2018

Abstract

In this project we tried to implement a Generative Adversarial Network (GAN) using PyTorch and compose a piano melody using this. 10 out of 68 piano melodies stored in midi format was provided as input to the GAN. The main aim is to expect a melody sequence which is similar to the input melodies.

1 Introduction

In our search for the project ideas, we have come across an Long Short-Term Memory (LSTM) Recurrent Neural Network (RNN) model which tries to produce a melody given a chord sequence. The goal of this LSTM model was to generate a melody which cannot be distinguished as either human generated or machine generated. We have realized that this problem statement could be apt for a Generator, Discriminator model where the Generator tries to fake the data based on the input provided and the Discriminator trying to discriminate between fake and input data.

2 Brief Theory

2.1 Introduction to GAN

Figure 1 Gives an overview of GA Network. Details can be found in literature [1].

- **R**: The original data set
- **I**:The random noise that goes into the generator as a source of entropy
- **G**:The generator which tries to copy/mimic the original data set
- **D**:The discriminator which tries to tell apart **G**'s output from R
- The actual training part where we teach **G** to trick **D** and **D** to beware **G**.

Figure 1: Block Diagram of GAN

2.2 Our Model

The schematic model used in the project is described below (see figure 2).

- R: Real data in this case is the data consisting of 10 out of 68 different melody taken from 16 different Beatles songs in MIDI format.
- I: Uniform distributed data with the combination of latent space (to match the dimensions of data) is passed into the generator as noise as a source of entropy.
- G: The generator is a standard feedforward graph with two hidden layers, three linear maps. A sigmoid function is used. G is going to get the uniformly distributed data samples from I and somehow mimic the piano roll samples from Rwithout ever seeing R.
- D: The discriminator is very similar to the Generator, i.e., a feedforward graph with two hidden layers, three linear maps and a sigmoid activation function. The Discriminator will get samples from either R or G and will output a single scalar between 0 and 1, interpreted as fake or real.
- Finally, the training loop alternates between two modes: first training D on real data vs. fake data; and then training G to fool D for given number of epochs

3 Implementation in PyTorch

3.1 Handling Data I/O

3.1.1 From MIDI to GAN Input

The necessary information needed for this project is a notes pitch, start time and length. We did not consider other rich information available in audio file such as bending of pitch to keep the input data simple and reasonably informative. Thus we used MIDI format file instead of audio formatted file such as mp3.

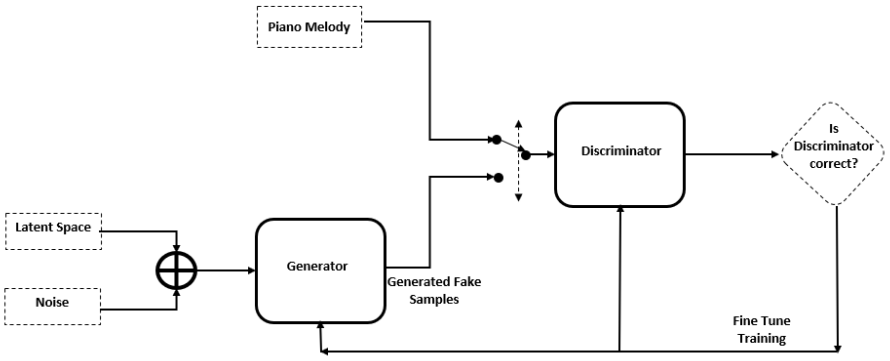

Figure 2:

To represent the incoming MIDI data in a manner that the GAN can understand, the piano roll representation has been chosen as per [2].As done originally in [2] the piano roll representation is transformed to a two-dimensional matrix with pitch as the first dimension and time as the second dimension. If a note from the piano roll is being played at one particular timestep, this will be denoted with a 1 in the matrix at this timestep and the notes pitch. If a note is not being played, this will be denoted with a 0 in the matrix.That is how music is represented in the form of a piano roll matrix consisting of ones if a note is on and zeros if a note is off.

3.1.2 From GAN Output to MIDI

At the end of the composition process Generator outputs a Prediction Matrix, which consists of values between zero and ones. In the next step, the Prediction Matrix is transformed into a piano roll matrix. This is done by iterating through each time-step (row) of the Prediction Matrix and finding the highest value within that time-step. If this value is higher than 0.6, it will be replaced by a one. All other entries of one timestep will be set to zero. If the highest value of one timestep is below the threshold, all entries of the timestep will be set to zero. After obtaining the piano roll representation we reverse the process of extracting information from MIDI,i.e. we create a MIDI from the piano roll representation.

For more details on handling input and output we refer to [2].

3.1.3 Tuning Speed

The melody generated by GAN is not unisonous with the input. To make that we can train another network. But for our project we chose an online sequencer and reduced the speed (BPM) to obtain somewhat melodious piano music.

3.2 Generator Network

Generator consists of 2 Hidden Layers of dimension 50 (As per the original code) ,1 input layer and 1 output layer. All the layers are linear. And the activation functions are ELU [3] (see Equation 1) and sigmoid. With ELU is given as,

$$ELU(x) = max(0, x) + min\Big(0, \alpha * \big(exp(x) - 1\big)\Big) \tag{1}$$

```
class Generator(nn.Module):
def __init__(self, input_size, hidden_size, output_size):
    super(Generator, self).__init__()
    self.map1 = nn.Linear(input_size, hidden_size)
    self.map2 = nn.Linear(hidden_size, hidden_size)
    self.map3 = nn.Linear(hidden_size, output_size)

def forward(self, x):
    x = F.elu(self.map1(x))
    x = torch.sigmoid(self.map2(x))
    return self.map3(x)
```

3.3 Discriminator Network

The discriminator is very similar to the Generator, i.e., a feedforward graph with two hidden layers, three linear maps and a sigmoid activation function. The Discriminator will get samples from either R or G and will output a single scalar between 0 and 1, interpreted as fake or real.

```
class Discriminator(nn.Module):
def __init__(self, input_size, hidden_size, output_size):
    super(Discriminator, self).__init__()
    self.map1 = nn.Linear(input_size, hidden_size)
    self.map2 = nn.Linear(hidden_size, hidden_size)
    self.map3 = nn.Linear(hidden_size, output_size)

def forward(self, x):
    x = F.elu(self.map1(x))
    x = F.elu(self.map2(x))
    return torch.sigmoid(self.map3(x))
```

3.4 Training Step

First we train the Discriminator(Henceforth mentioned as D) on Real Data. Next D is trained on fake data,*i.e.* on data generated by Generator(Henceforth mentioned as G). We

then train G on D's response. But D is not trained on G's response. This will be done in the next epoch.

```
for epoch in range(num_epochs):
    for d_index in range(d_steps):
        # 1. Train D on real+fake
        D.zero_grad()

        #  1A: Train D on real
        d_real_data = Variable(dataset)
        d_real_decision = D(d_real_data)
        d_real_error = criterion(d_real_decision,
           Variable(torch.ones(d_real_decision.shape)))
                                    # ones = true
        d_real_error.backward()  # compute/store gradients, but don't change params

        #  1B: Train D on fake
        d_gen_input = Variable(noise())
        d_fake_data = G(d_gen_input).detach()
                # detach to avoid training G on these labels
        d_fake_decision = D(d_fake_data)
        d_fake_error = criterion(d_fake_decision,
        Variable(torch.zeros(d_fake_decision.shape)))
                                               # zeros = fake
        d_fake_error.backward()
        d_optimizer.step()     # Only optimizes D's parameters;
                changes based on stored gradients from backward()
        D_error.append(d_fake_error.item())

    for g_index in range(g_steps):
        # 2. Train G on D's response (but DO NOT train D on these labels)
        G.zero_grad()

        gen_input = Variable(noise())
        g_fake_data = G(gen_input)
        dg_fake_decision = D(g_fake_data)
        g_error = criterion(dg_fake_decision,
         Variable(torch.ones(dg_fake_decision.shape)))  # we want to fool,
        G_error.append(g_error.item())
        g_error.backward()
        g_optimizer.step()  # Only optimizes G's parameters
        print(epoch, " : D: ", d_fake_error.item(), " G:", g_error.item())
```

3.5 Model Parameters

Input dimension of D and G are both 12. Output dimension for D is 1 (0 for fake, 1 for real) whereas for G is 12. Learning rate,η, is 0.0002 , $\beta = (0.9, 0.999)$,D training frequency is 1 and G training frequency is also 1 as proposed in original paper [1]. The epoch is taken as user input. For this project at max we have taken 3000 epochs.

```
input_dim = input_data.shape[2]
output_dim = input_data.shape[2]

# Model params
g_input_size = input_dim    # Random noise dimension coming into
                            # generator, per output vector
g_hidden_size = 50   # Generator complexity
g_output_size = output_dim    # size of generated output vector
‘   Q
d_input_size = input_dim   # Minibatch size - cardinality of distributions
d_hidden_size = 50   # Discriminator complexity
d_output_size = 1    # Single dimension for 'real' vs. 'fake'
d_learning_rate = 2e-4  # 2e-4
g_learning_rate = 2e-4
optim_betas = (0.9, 0.999)
print("For how many epochs do you wanna train?")
num_epochs = int(input('Num of Epochs:'))
print_interval = 1
d_steps = 1  # 'k' steps in the original GAN paper. Can put the
            # discriminator on higher training freq than generator
g_steps = 1
```

3.6 Optimizer and Loss Functions

Optimizer is the conventional torch.optim.Adam. The Loss function is taken as Binary Cross Entropy Loss (BCE Loss) given by Equation 2.

$$H(p, q) = -\sum_{x} p(x) \log q(x) \tag{2}$$

For the GANs, the loss that we need to minimize is,

$$V(D, G) = E_{p_{data}}\big[\log D(x)\big] + E_{p_z}\big[\log\left(1 - D(G(z))\right)\big] \tag{3}$$

Instead of optimizing $\log\left(1 - D(G(z))\right.$ we optimize the quantity, $\log D(G(Z))$, and Equation 3 takes form of Equation 2.

4 Results

- Sample inputs and outputs can be found in here.

- Figure 3 shows the Graph of $D(G(z))$ and $G(z)$. As expected when D's error rate is high G's error rate decreases. That means D faces a hard time to discriminate between Real and Fake samples. However after sometime D gets better and G's error rate increases. That means D easily distinguishes Fake samples generated by G. This nonzero sum game between D and G ultimately forces G to produce a sample that D fails to detect as a fake.

- Due to time and resource constraints we could not show that gradually this oscillations stabilizes as it must happen theoretically. D's error should increase and G's error should decrease at a steady rate. However this takes a lot of epochs generally after 30k epochs this oscillations wane off.

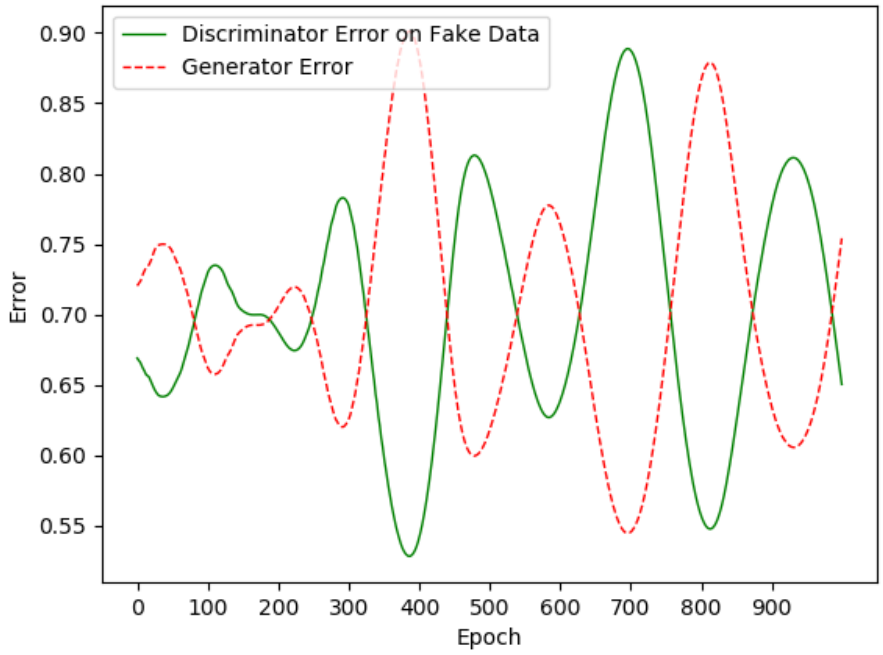

Figure 3: Discriminator and Generator Error

5 Conclusion

For this project a simple architecture for GANs were chosen. More complicated GANs such as museGAN, DC-GAN can be used to obtain a melody and already exists. Also LSTM-RNN [2],upon which our project is based can be a good choice.

6 Limitations

- We could not test whether melodies generated by our model is pleasant or not.

- More epochs were needed to run so that G converged. We feel like at least 30k epochs or maybe higher were needed so that G converged.
- Complexity of both Discriminator and Generator network should be increased.
- Different Optimization and Gradient functions for GANs can be a good area to venture.

7 Acknowledgement

We thank Dr. Alan Ritter to give us such a wonderful opportunity to work on this exciting project. Sincere thanks to various blog posts [2, 4, 5] that described GAN in detail.

References

[1] Generative adversarial nets I Goodfellow et al. *Advances in neural information processing systems*, 2014

[2] *Bachelor's Thesis*, Konstantin Lackner, https://konstilackner.github.io/LSTM-RNN-Melody-Composer-Website/Thesis_final01.pdf.

[3] *PyTorch Documentation*,https://pytorch.org/docs/stable/nn.html#torch-nn-functional

[4] https://medium.com/@devnag/generative-adversarial-networks-gans-in-50-lines-of-code-pytorch-e81b79659e3f

[5] https://medium.com/ai-society/gans-from-scratch-1-a-deep-introduction-with-code-in-pytorch-and-tensorflow-cb03cdcdba0f