final_project

July 31, 2020

CSE 190 Project

Name: Andy Duong
PID: A13528452

1 Introduction

Ask anyone in this day and age if they know who Mario is. You'll probably find that most if not all people can recognize the name. Even more might immediately recognize the tune of the original Super Mario Bros game. All of the these pieces were written by a composer by the name of Koji Kondo. Koji is a lead composer at Nintendo and has led the creation of soundtracks for multiple Mario and even Legend of Zelda games. Personally, his music is very memorable to me because I grew up playing Mario games. I wanted to take this opportunity to see if a network was capable of emulating the type of creativity required to compose some interesting songs.

2 Background

In this project, I will be using a generative adversarial network (GAN) to generate new Mario music. The GAN framework consists of 2 separate networks, each with their own role: a generator and a discriminator.

A generator is responsible for converting the latent vector into a prediciton. A discriminator is responsible for validating the predictions of the generator. A generator starts with a random latent vector and a random set of weights in its internal nodes. The discriminator then tries to distinguish the generated "fake" output from the training "real" data.

At start-up, both networks incur high levels of error, but as training continues, the generator and discriminator should be learning at around the same rate. There is a feedback loop between these two networks that allows them to improve each other.

3 Generative Adversarial Network Architecture

This code is adapted from a tutorial on a similar project, in which the researchers use a GAN to generate Pokemon music. I adapted their code to work with input MIDI files that I found.

The MIDI files were songs from several Mario games. * Super Mario Bros: Overworld Main Theme, Rescue Fanfare, Starman Theme, Underwater Theme, Underworld Theme, Castle Theme, Ending Theme * Paper Mario 64: Crystal Palace, Koopa Village, Starborn Valley, Title Screen, Tubba Blubba Battle, Yoshi Island 2 * Super Mario 64: Cool Cool Mountain, Dire Dire Docks, Koopa Theme, Lava Lava Island, Title Theme, Inside the Castle Walls, Bob-omb Battlefield

Below is all of the code that was used to generate the output files.
The GitHub link is https://github.com/axpecial/cse190-mario-music.

3.0.1 Import TensorFlow and other libraries

[4]:
```
import tensorflow as tf
```

[5]:
```
tf.__version__
```

[5]: '2.2.0'

[6]:
```
import glob
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
from tensorflow.keras import layers
import time
import pickle
from music21 import converter, instrument, note, chord, stream
from keras.layers import Input, Dense, Reshape, Dropout, LSTM, Bidirectional
from keras.layers import BatchNormalization, Activation, ZeroPadding2D
from keras.layers.advanced_activations import LeakyReLU
from keras.models import Sequential, Model
from keras.optimizers import Adam
from keras.utils import np_utils
```

```
Bad key "text.kerning_factor" on line 4 in
/Users/andyduong/opt/anaconda3/lib/python3.7/site-packages/matplotlib/mpl-
data/stylelib/_classic_test_patch.mplstyle.
You probably need to get an updated matplotlibrc file from
https://github.com/matplotlib/matplotlib/blob/v3.1.3/matplotlibrc.template
or from the matplotlib source distribution
```

3.0.2 Load and prepare the dataset

[7]:
```
# Number of notes passed into the model so it can predict the subsequent notes
SEQ_INPUT_LEN = 100
SEQ_INPUT_SHAPE = (SEQ_INPUT_LEN, 1)
```

[8]:
```
def get_notes():
    notes = []
    for file in glob.glob("mario_midi/**/*.mid", recursive=True):
        midi = converter.parse(file)
        print("Parsing %s" % file)
        notes_to_parse = None
        try: # file has instrument parts
            s2 = instrument.partitionByInstrument(midi)
            notes_to_parse = s2.parts[0].recurse()
        except: # file has notes in a flat structure
            notes_to_parse = midi.flat.notes

        for element in notes_to_parse:
            if isinstance(element, note.Note):
                notes.append(str(element.pitch))
            elif isinstance(element, chord.Chord):
                notes.append('.'.join(str(n) for n in element.normalOrder))

    pickle.dump(notes, open('notes.p', 'wb'))

    return notes

def prepare_sequences(notes, n_vocab):
    """ Prepare the sequences used by the Neural Network """
    sequence_length = SEQ_INPUT_LEN

    pitchnames = sorted(set(item for item in notes))

    note_to_int = dict((note, number) for number, note in enumerate(pitchnames))

    network_input = []
    network_output = []

    for i in range(0, len(notes) - sequence_length, 1):
        sequence_in = notes[i:i + sequence_length]
        sequence_out = notes[i + sequence_length]
        network_input.append([note_to_int[char] for char in sequence_in])
        network_output.append(note_to_int[sequence_out])

    n_patterns = len(network_input)
```

```
    # reshape the input into a format compatible with LSTM layers
    network_input = np.reshape(network_input, (n_patterns, sequence_length, 1))
    # normalize input between -1 and 1
    network_input = (network_input - float(n_vocab)/2) / (float(n_vocab)/2)

    network_output = np_utils.to_categorical(network_output)

    return (network_input, network_output)
```

[10]:
```
notes = get_notes()
```

3.1 Create the models

Both the generator and discriminator are defined using the Keras Sequential API.

3.1.1 The Generator

[8]:
```
# Size of latent space for the model to work with
LATENT_DIM = 25
```

[9]:
```
def build_generator():
    model = tf.keras.Sequential()
    model.add(layers.Dense(256, input_dim=LATENT_DIM))
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.BatchNormalization(momentum=0.8))
    model.add(layers.Dropout(0.3))
    model.add(layers.Dense(512))
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.BatchNormalization(momentum=0.8))
    model.add(layers.Dropout(0.3))
    model.add(layers.Dense(1024))
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.BatchNormalization(momentum=0.8))
    model.add(layers.Dropout(0.3))
    model.add(layers.Dense(np.prod(SEQ_INPUT_SHAPE), activation='tanh'))
    model.add(layers.Reshape(SEQ_INPUT_SHAPE))
    model.summary()

    noise = Input(shape=(LATENT_DIM,))
    seq = model(noise)

    return Model(noise, seq)
```

3.1.2 The Discriminator

[10]:
```
def build_discriminator():
    model = tf.keras.Sequential()
    model.add(layers.LSTM(512, input_shape=SEQ_INPUT_SHAPE,␣
 ↪return_sequences=True))
    model.add(layers.Bidirectional(layers.LSTM(512)))
    model.add(layers.Dense(512))
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.Dropout(0.3))
    model.add(layers.Dense(256))
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.Dropout(0.3))
    model.add(layers.Dense(1, activation='sigmoid'))
    model.summary()

    seq = Input(shape=SEQ_INPUT_SHAPE)
    validity = model(seq)

    return Model(seq, validity)
```

3.1.3 GAN (Generator + Discriminator)

[11]:
```
optimizer = Adam(0.0002, 0.5)

# Build and compile the discriminator
discriminator = build_discriminator()
discriminator.compile(loss='binary_crossentropy', optimizer=optimizer,␣
 ↪metrics=['accuracy'])

# Build the generator
generator = build_generator()

# The generator takes noise as input and generates note sequences
z = Input(shape=(LATENT_DIM,))
generated_seq = generator(z)

# For the combined model we will only train the generator
discriminator.trainable = False

# The discriminator takes generated images as input and determines validity
validity = discriminator(generated_seq)

# The combined model  (stacked generator and discriminator)
# Trains the generator to fool the discriminator
gan = Model(z, validity)
```

```python
gan.compile(loss='binary_crossentropy', optimizer=optimizer)
gan.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 100, 512)          1052672
_________________________________________________________________
bidirectional (Bidirectional (None, 1024)              4198400
_________________________________________________________________
dense (Dense)                (None, 512)               524800
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 512)               0
_________________________________________________________________
dropout (Dropout)            (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               131328
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 256)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 257
=================================================================
Total params: 5,907,457
Trainable params: 5,907,457
Non-trainable params: 0
_________________________________________________________________
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 256)               6656
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 256)               0
_________________________________________________________________
batch_normalization (BatchNo (None, 256)               1024
_________________________________________________________________
dropout_2 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_4 (Dense)              (None, 512)               131584
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 512)               0
_________________________________________________________________
batch_normalization_1 (Batch (None, 512)               2048
_________________________________________________________________
```

```
dropout_3 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_5 (Dense)              (None, 1024)              525312
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 1024)              0
_________________________________________________________________
batch_normalization_2 (Batch (None, 1024)              4096
_________________________________________________________________
dropout_4 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_6 (Dense)              (None, 100)               102500
_________________________________________________________________
reshape (Reshape)            (None, 100, 1)            0
=================================================================
Total params: 773,220
Trainable params: 769,636
Non-trainable params: 3,584
_________________________________________________________________
Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 25)]              0
_________________________________________________________________
model_1 (Model)              (None, 100, 1)            773220
_________________________________________________________________
model (Model)                (None, 1)                 5907457
=================================================================
Total params: 6,680,677
Trainable params: 769,636
Non-trainable params: 5,911,041
_________________________________________________________________
```

3.2 Define the training loop

[17]:
```
# number of iterations to run the network
EPOCHS = 50
# number of sequences of length SEQ_INPUT_LEN that will be used as training data
BATCH_SIZE = 128
# interval for saving checkpoints
SAMPLING_INTERVAL = 10

disc_loss = []
gen_loss = []
```

The training loop begins with generator receiving a random seed as input. That seed is used to produce an image. The discriminator is then used to classify real images (drawn from the training set) and fakes images (produced by the generator). The loss is calculated for each of these models, and the gradients are used to update the generator and discriminator.

[18]:
```
def train():
    # Load and convert the data
    notes = pickle.load(open('notes.p', 'rb'))
    n_vocab = len(set(notes))
    network_input, network_output = prepare_sequences(notes, n_vocab)

    # Adversarial ground truths
    real = np.ones((BATCH_SIZE, 1))
    fake = np.zeros((BATCH_SIZE, 1))

    # Training the model
    for epoch in range(EPOCHS):

        # Training the discriminator
        # Select a random batch of note sequences
        idx = np.random.randint(0, network_input.shape[0], BATCH_SIZE)
        real_seqs = network_input[idx]

        noise = np.random.normal(0, 1, (BATCH_SIZE, LATENT_DIM))

        # Generate a batch of new note sequences
        gen_seqs = generator.predict(noise)

        # Train the discriminator
        d_loss_real = discriminator.train_on_batch(real_seqs, real)
        d_loss_fake = discriminator.train_on_batch(gen_seqs, fake)
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        #  Training the Generator
        noise = np.random.normal(0, 1, (BATCH_SIZE, LATENT_DIM))

        # Train the generator (to have the discriminator label samples as real)
        g_loss = gan.train_on_batch(noise, real)

        # Print the progress and save into loss lists
        if (epoch + 1) % SAMPLING_INTERVAL == 0:
            print ("%d [D loss: %f, acc.: %.2f%%] [G loss: %f]" % (epoch,␣
↪d_loss[0], 100*d_loss[1], g_loss))
            disc_loss.append(d_loss[0])
            gen_loss.append(g_loss)

    # Generate after the final epoch
    generate_and_save_midi(notes)
    plot_loss()
```

Generate MIDI

[19]:
```
def generate_and_save_midi(notes):
    # Get pitch names and store in a dictionary
    notes = pickle.load(open('notes.p', 'rb'))
    pitchnames = sorted(set(item for item in notes))
    int_to_note = dict((number, note) for number, note in enumerate(pitchnames))

    # Use random noise to generate sequences
    noise = np.random.normal(0, 1, (1, LATENT_DIM))
    predictions = generator.predict(noise)

    pred_notes = [min(len(pitchnames)-1, np.abs(x*len(pitchnames))) for x in␣
↪predictions[0]]
    pred_notes = [int_to_note[int(x)] for x in pred_notes]

    create_midi(pred_notes)
```

[20]:
```
def create_midi(prediction_output):
    offset = 0
    output_notes = []
    for pattern in prediction_output:
        if ('.' in pattern) or pattern.isdigit():
            notes_in_chord = pattern.split('.')
            notes = []
            for current_note in notes_in_chord:
                new_note = note.Note(int(current_note))
                new_note.storedInstrument = instrument.Piano()
                notes.append(new_note)
            new_chord = chord.Chord(notes)
            new_chord.offset = offset
            output_notes.append(new_chord)
        else:
            new_note = note.Note(pattern)
            new_note.offset = offset
            new_note.storedInstrument = instrument.Piano()
            output_notes.append(new_note)
        offset += 0.5
    midi_stream = stream.Stream(output_notes)
    midi_stream.write('midi', fp='gan_output.mid')
```

Plot Losses

[21]:
```
def plot_loss():
    plt.plot(disc_loss, c='red')
    plt.plot(gen_loss, c='blue')
    plt.title("GAN Loss per Epoch")
    plt.legend(['Discriminator', 'Generator'])
```

```
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.savefig('GAN_Loss_per_Epoch_final.png', transparent=True)
    plt.close()
```

3.3 Train the model

Call the `train()` method defined above to train the generator and discriminator simultaneously. Note, training GANs can be tricky. It's important that the generator and discriminator do not overpower each other (e.g., that they train at a similar rate).

At the beginning of the training, the generated images look like random noise. As training progresses, the generated digits will look increasingly real. After about 50 epochs, they resemble MNIST digits. This may take about one minute / epoch with the default settings on Colab.

[22]:
```
train()
```

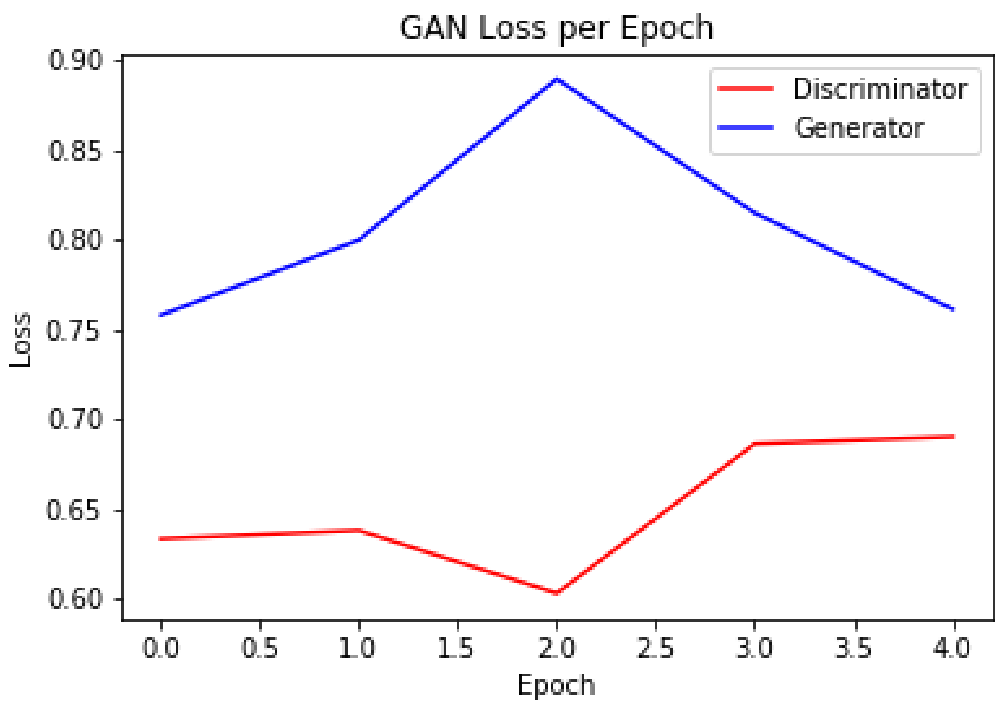
```
9 [D loss: 0.588488, acc.: 63.67%] [G loss: 0.785792]
19 [D loss: 0.654489, acc.: 62.11%] [G loss: 0.901782]
29 [D loss: 0.684950, acc.: 58.59%] [G loss: 1.015698]
39 [D loss: 0.705766, acc.: 50.39%] [G loss: 0.837107]
49 [D loss: 0.693023, acc.: 48.83%] [G loss: 0.853782]
```

4 Experimentation + Results

4.1 Iteration 0: Adaptation of Tutorial

In iteration 0 of the network, I adapted the code provided in the tutorial to work with the Mario MIDI files.

4.1.1 Results:

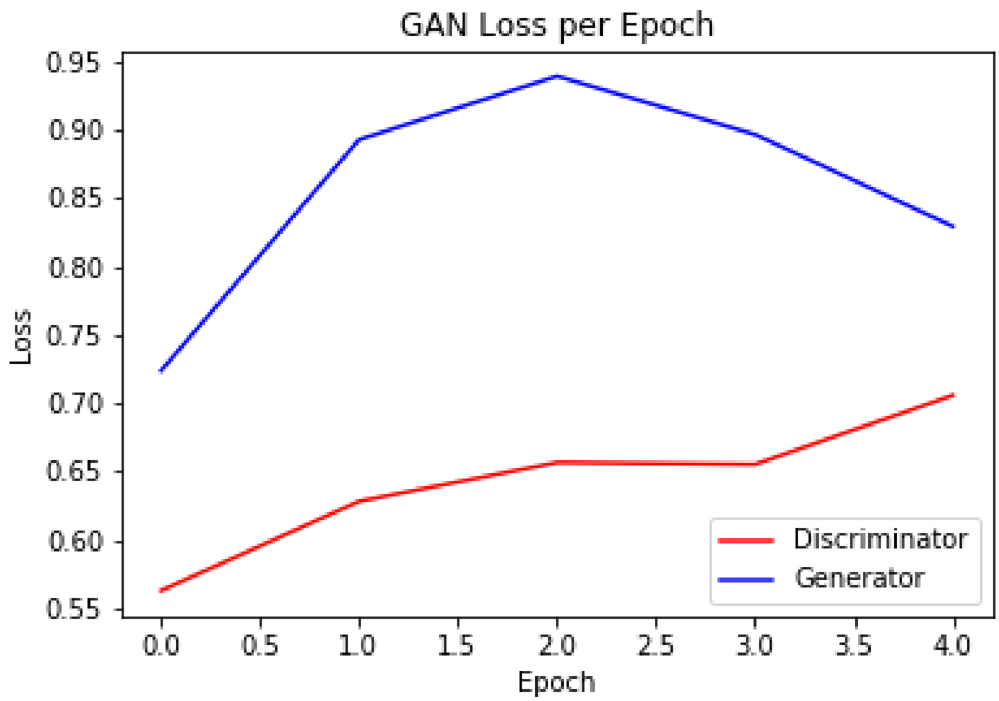


Audio: https://github.com/axpecial/cse190-mario-music/tree/master/initial_output

4.2 Iteration 1: Dropout Layers in Generator

In iteration 1, I looked to the RNN construction in Assignment 3 for inspiration on how to modify my current generator. I decided on included several Dropout layers.

4.2.1 Results:

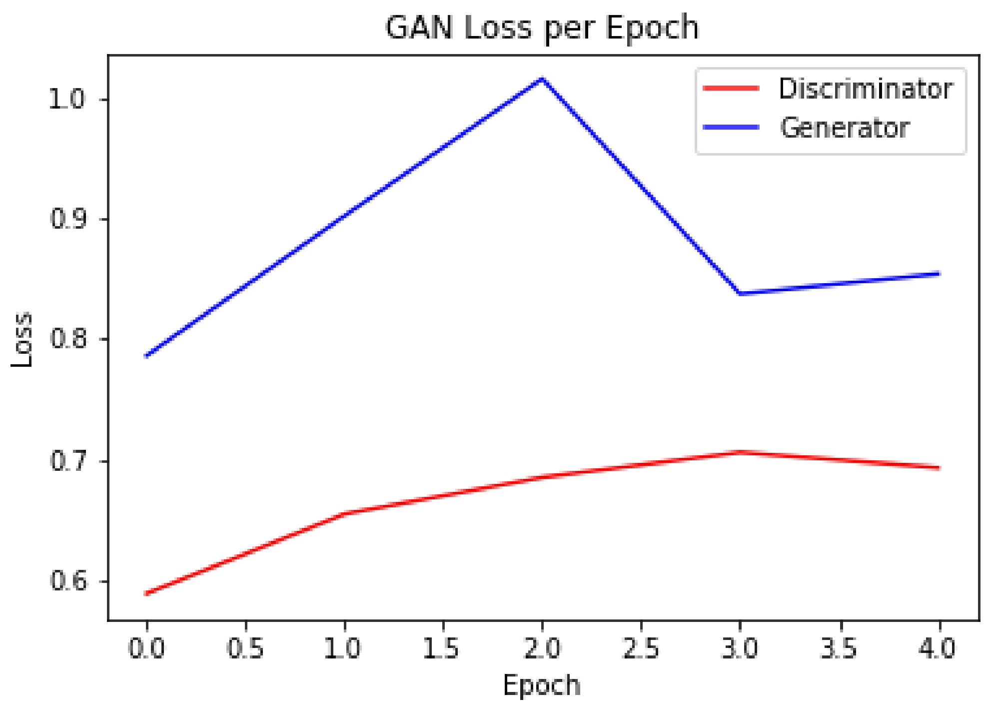


Audio: https://github.com/axpecial/cse190-mario-music/tree/master/dropout_gen_output

4.3 Iteration 2: Dropout Layers in Generator and Discriminator

In iteration 2, I thought back to the idea that the generator and discriminator should be learning at the same rate so that one doesn't overpower the other. I wondered if the dropout layers added to the generator did help it learn faster and that the discriminator was falling behind. I tested this hypothesis by adding Dropout layers to both the generator and discriminator.

4.3.1 Results:



Audio: https://github.com/axpecial/cse190-mario-music/tree/master/dropout_gen_dis_output

5 Conclusion

Overall, this a challenging yet fun project to work on. If I had more time, I'd want to revisit this prompt and consider pursuing some of the following paths: * train the network with more Mario MIDI files * preprocess the MIDI files to separate instruments * group Mario songs that have a similar mood * create new encoding that considers rhythm * experiment with different layer types * explore other papers on GAN music generation