Motivation & Approach

In this paper, I attempt two improvements over the DCGAN that I coded for the GAN assignment, you can refer to the pdf at this link. In that assignment, I attempted to train a DCGAN to generate excerpts of sheet music using 1000x1000 grids from the ~4000x3000 original images from DeepScore dataset. I then reduced the dimensions of the grids into 64x64, or 128x128 sized images for training. The results of the original GAN for 30 epochs can be viewed here.

I made two hypotheses:

1. Hypothesis 1: If I use a cleaner dataset, I will probably get a better result.
2. Hypothesis 2: If I train a conditional-GAN, conditioning on the first note appearing in each image, I will probably get a better result.

Here is how I acted upon each hypothesis:

Hypothesis 1:

Instead of using the DeepScore dataset of images, I took the Bach Chorale midi dataset that I used for my sequential prediction assignment. I then used Music21 to convert the midi files into png images of sheet music. This allowed me to specify exactly what I allow in each picture (i.e. which notes, or time steps in the song etc.). I decided to only allow a single melody line in each picture (the chorales have four melody lines each). Furthermore, I only allowed four time steps per picture (a time step is defined as a quarter-note). Given that most of the chorales are in 4/4 time (4 quarter notes per bar) this ensured that each picture generally consisted of around one to four notes. I hypothesised that doing this was a better way to split up the image than arbitrarily splitting the picture into 1000x1000 grids. I wanted to make the task easier for my GAN, at least initially. Finally, in order to get square images, I padded each rectangular picture of a melody line with black. I kept the data to one melody line for the sake of simplicity and to see if the GAN works well with this problem, but I will talk about extending to multiple lines in the Future Works Section. For reference, Figure 1 shows what the grid of input images looked like for my original dataset, and Figure 2 shows what the grid of input images looks like in my new dataset.

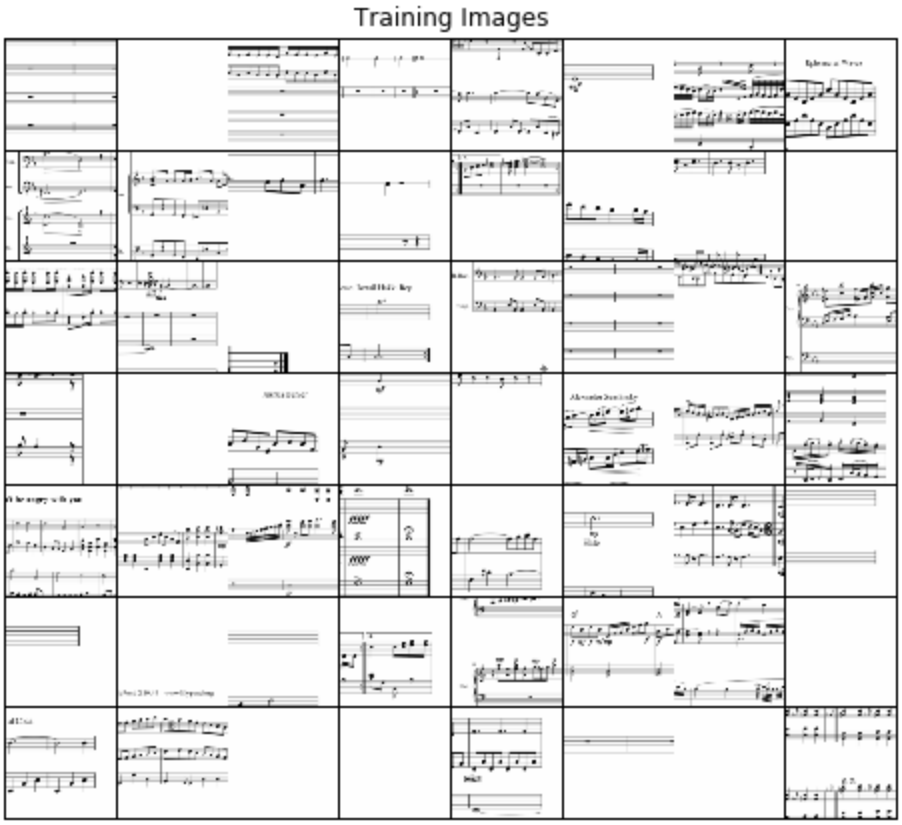


Figure 1. Training images from original DeepScore dataset.

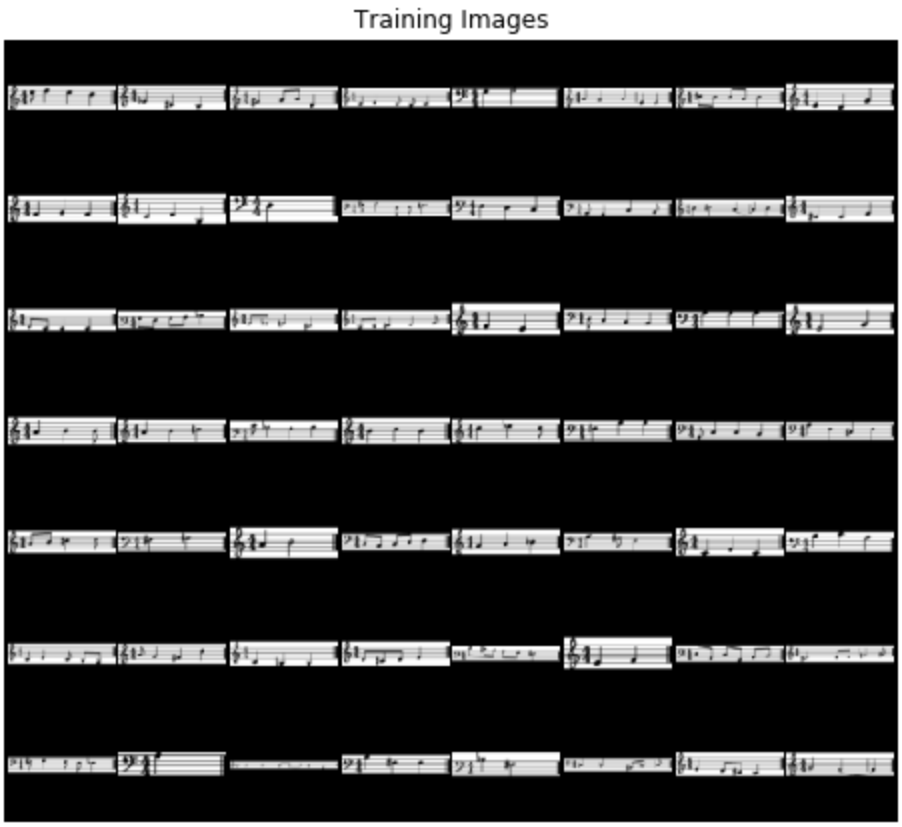


Figure 2. Training images from new dataset.

One thing that I thought might affect my results is that I used 110 of Bach's chorales which were split into 7521 images, as opposed to >20000 images I had in the original dataset. But using such a large dataset would further reduce the turnover rate of me fiddling with hyperparameters, so I kept it at 7521, though the dataset has 371 chorales in total.

Hypothesis 2:

A CGAN is a GAN that adds a label to each training image. We know that the MNIST dataset is often trained using a CGAN, with class labels. I hypothesised that adding class labels would improve my result as well, by virtue of providing both the generator and discriminator with a relevant feature of the image. Instead of labelling all the notes in the picture, I chose to provide labels for the first note only — in this way, we can imagine using this GAN to generate music by providing it with the first note — much like the way many sequential music models work. We might also imagine using a variant of the CGAN that I trained here that takes in all the notes in the image as a condition, being used to train a model that generates sheet music from a MIDI file

(though there are much more sophisticated models for this). More likely, the discriminator might be used in optical music recognition systems.

Notes on Implementation:

I have three sections to my code, and I explain them here, for logistical purposes. The first section is the code that I used to preprocess the images. The second section is the code that I ended up using for the CGAN, and its unconditional control (eriklindernoren, 2019). The third section is the original code that I used for the GAN assignment (Inkawhich, 2017).

I had quite a lot of trouble implementing the CGAN from the DCGAN that I had originally used due to dimensionality issues & the convolutions. Therefore, I finally decided to just use a separate CGAN code which I modified from eriklindernoren(2019) — not ideal and not modular, but I did it due to time constraints. As a note, the main difference that I noticed between the new code and my original code was just that they used linear layers instead of convolutional layers. So, my original implementation was a Deep Convolutional Generative Adversarial Network (DCGAN) but the code that I used was a CGAN that did not utilize convolutions. I will explain the difference in results more in the next section, and how I tried to weed out the distinguishing features which made the new GAN successful.

Result 1 [original code, new dataset]:

I originally tried simply using my old code on the newly prepared dataset, and was disappointed to find that it produced very bad results, even after lots of fiddling around with the learning rates and trying some label smoothing (so much, that I ran out Colab GPU compute time and had to move to Nimblebox). Here are the results, these are already some of the best results, and they are quite terrible.

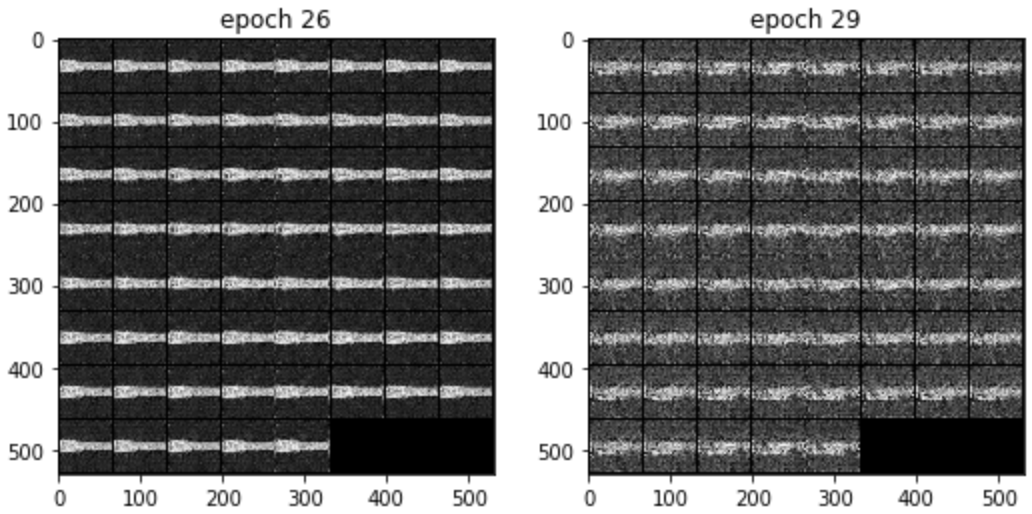


Figure 3.DCGAN model with convolutional layers & new dataset. Default hyperparameters, except the learning rate of the discriminator was set to 0.0001 instead of 0.0002.

We can see that the model is very obviously in a state of mode collapse, the loss indicated the same, with the discriminator and generator loss remaining the same for many epochs, and the generator loss being much higher than that of the discriminator. I did try reducing the learning rate of the discriminator up to a quarter that of the generator (so 0.00005), but this did not help.

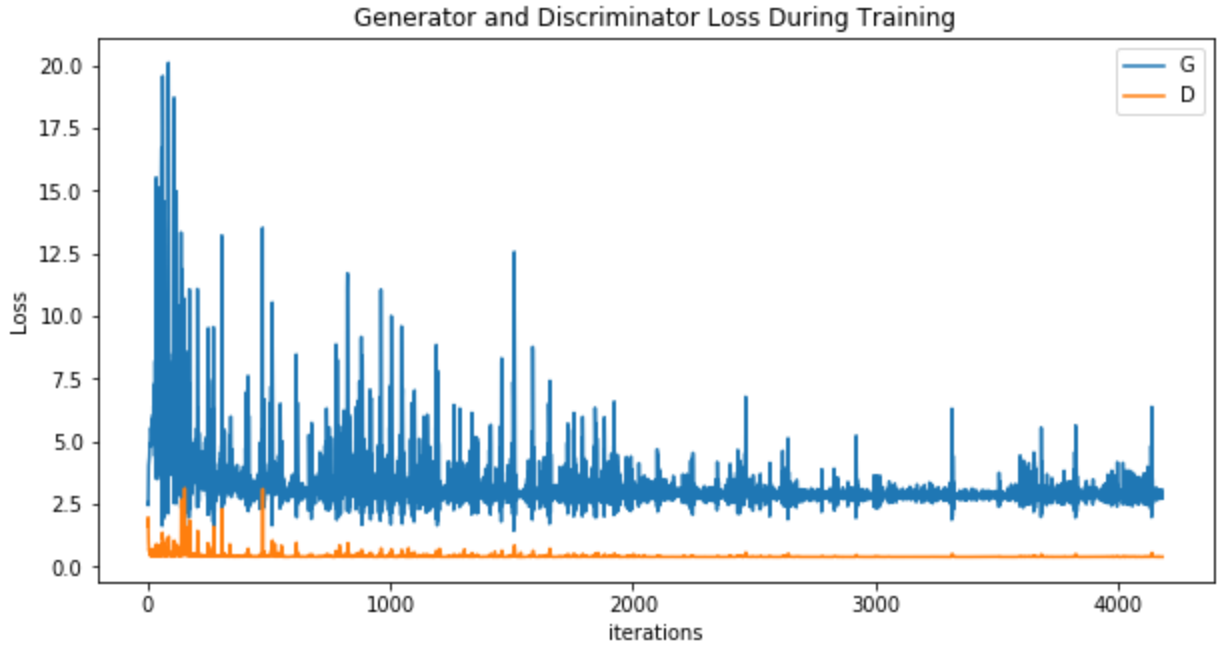


Figure 4. Loss for DCGAN model with convolutional layers & new dataset. Measured after ~30 epochs.

Result 2 [new code, conditional variant, new dataset]

Whilst I was training the first model, I also worked on the CGAN, and finally switched to the new code (as mentioned earlier). This produced stunningly better results (I doubted that the improvement in results was so drastic only due to introducing conditionality, and I explain my narrowing down process in results #3 and #4). To introduce conditionality, I simply concatenated the vector that tells me the first note that appears in a picture. You can view the full visualization of the results (I saved generated images here, they are labelled by batch number, not epoch). The model was also hundreds of times faster per epoch than the original model despite using the same batch size of 56(by my estimation & calculation, at least 100 times faster). Here is a visualization of the loss & the result at the final epoch (I trained for 2100 epochs, and this took 4-5 hours on the Nimblebox GPU):

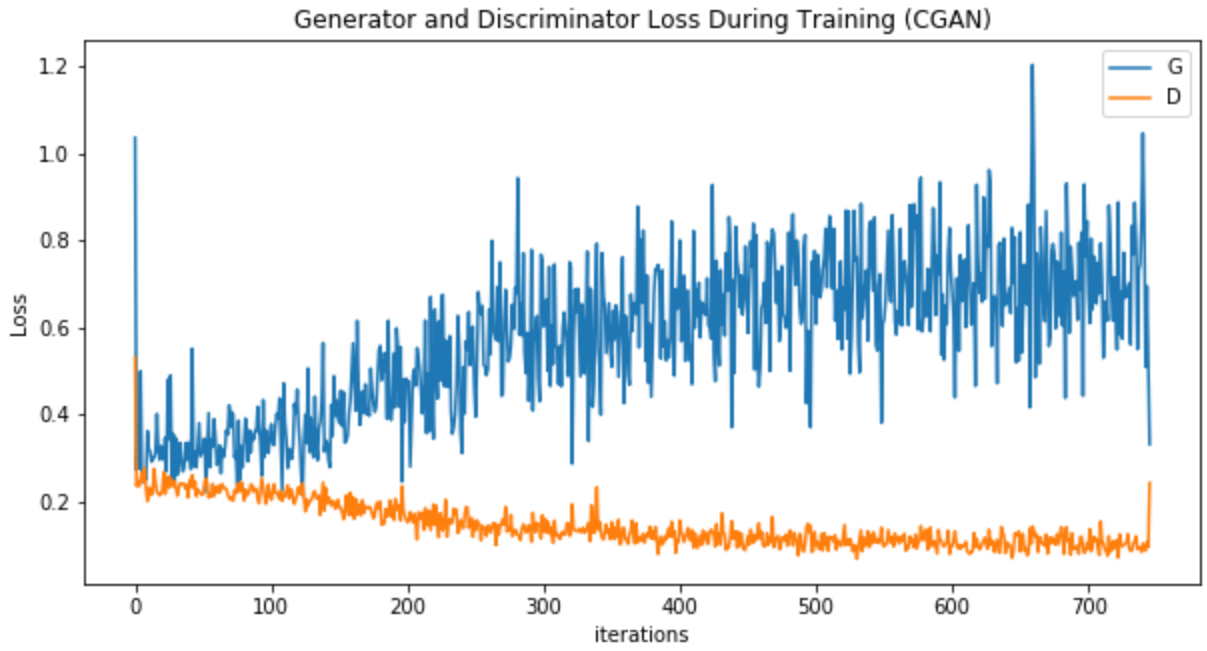


Figure 5. Loss for CGAN model with linear layers & new dataset. Measured after ~2100 epochs. Default hyperparameters.

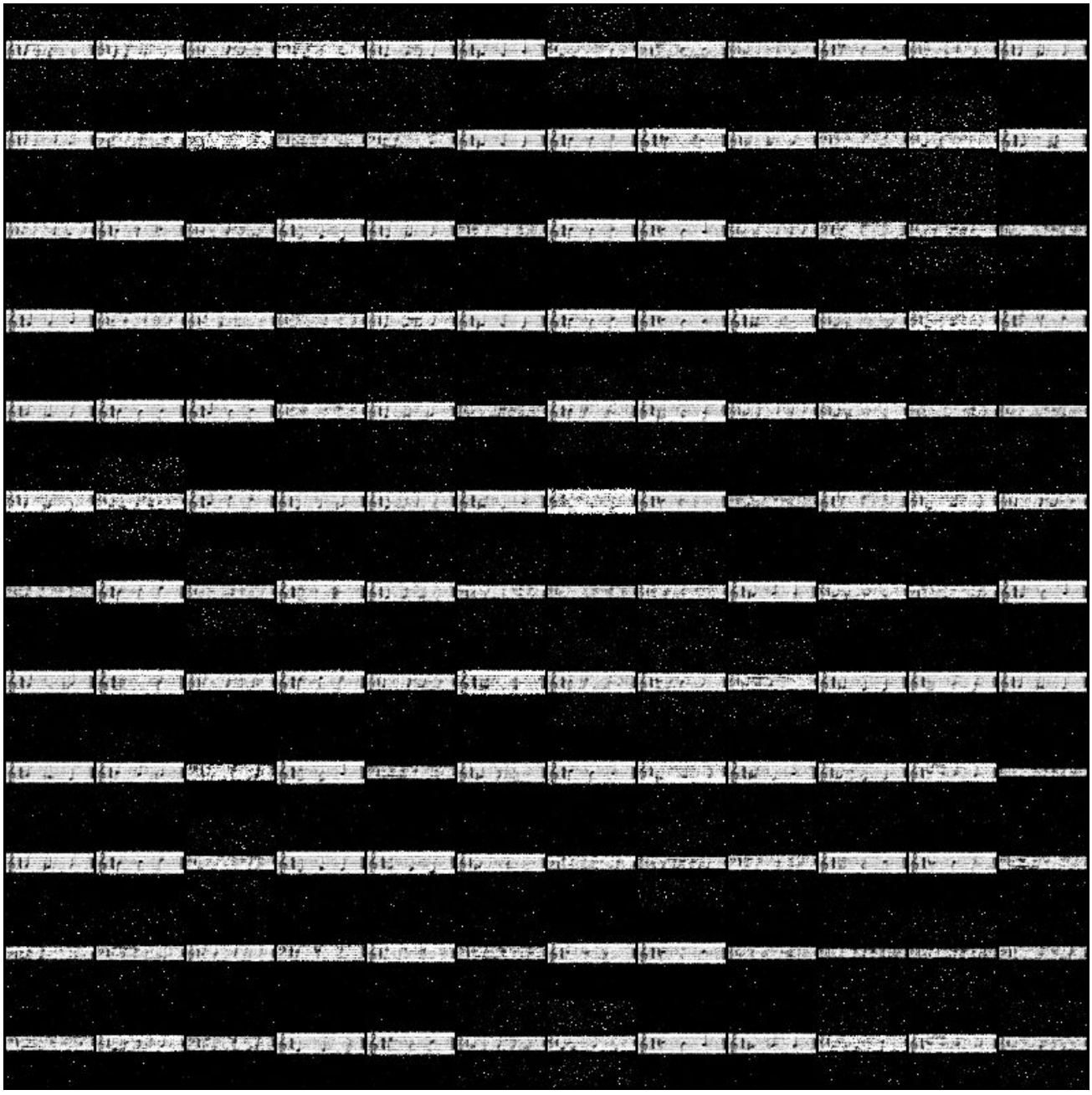


Figure 6. Results for CGAN model with linear layers & new dataset. Measured after ~2100 epochs. Default hyperparameters. {0: 'G', 1: 'B', 2: 'A, 3: 'D', 4: 'E', 5: 'F#', 6: 'C', 7: 'C#', 8: 'G#', 9: 'Eb', 10: 'Bb', 11: 'F'}

I am reasonably happy with these results. First of all, you can clearly read notes from a number of the results. Second of all, a reasonable number notes do seem to be accorded the correct labels. For example, looking at column 0, the 4th, 5th, 6th, 8th, 9th and 10th rows all can be said to start with the G-note. This is true in multiple cases (for the B note, C note, F# note etc.). I cannot claim that there is no mode collapse, because I do see repetition of some melodies, but for

a single type of note, we see that there is usually more than one pattern produced, which is a somewhat good sign. These results are clearly much better than that of the original model, but at this point in my experimentation process, I could not yet attribute this change to the change in dataset, or due to imposing conditionality, or to some other change in the new code (I suspect linear layers). This is why I continued my experiments. Firstly, I changed the new conditional GAN to an unconditional GAN by using a concatenation of zeros, instead of correct labels (Result 3). Then, I trained the new unconditional GAN on the old dataset(Result 4). For result 3 and 4, I only ran the model for 500 epochs, because this was just to test my hypotheses.

Result 3[new code, unconditional variant, new dataset]

Here is the loss plot for the unconditional GAN:

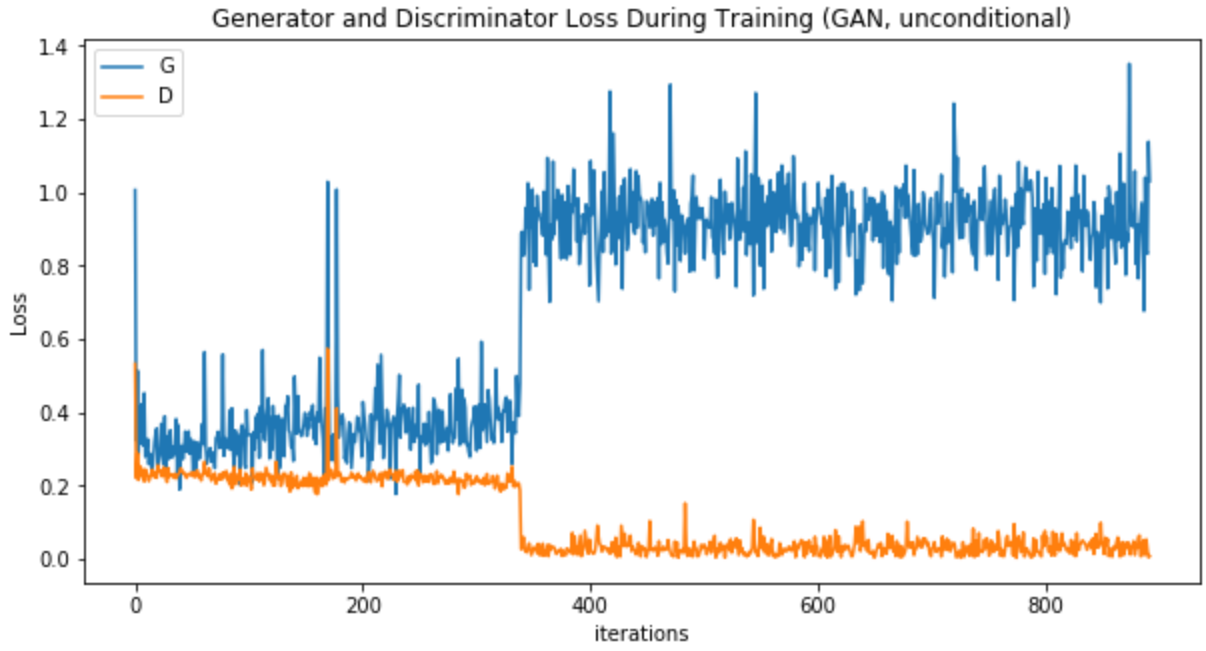


Figure 7. Loss for GAN model with linear layers & new dataset. Measured after ~500 epochs. Default hyperparameters.

It does seem that the conditional GAN has a much less sudden "winning" of the discriminator, and the "winning" occurs very early on in the unconditional GAN. This indicates that perhaps adding the conditionality to both the discriminator and the generator helps the quality of images produced by the GAN and makes sure that the discriminator does not overpower too early on.

Here is a visualization of the generated fakes at epoch 500 of this unconditional GAN(Figure 6) and the same epoch for the earlier CGAN (Figure 7):

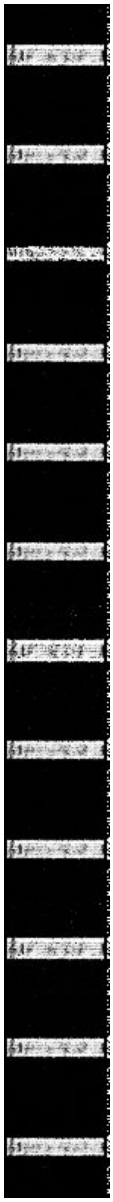


Figure 8. Results for GAN model(unconditional) with linear layers & new dataset. Measured after ~500 epochs. Default hyperparameters.

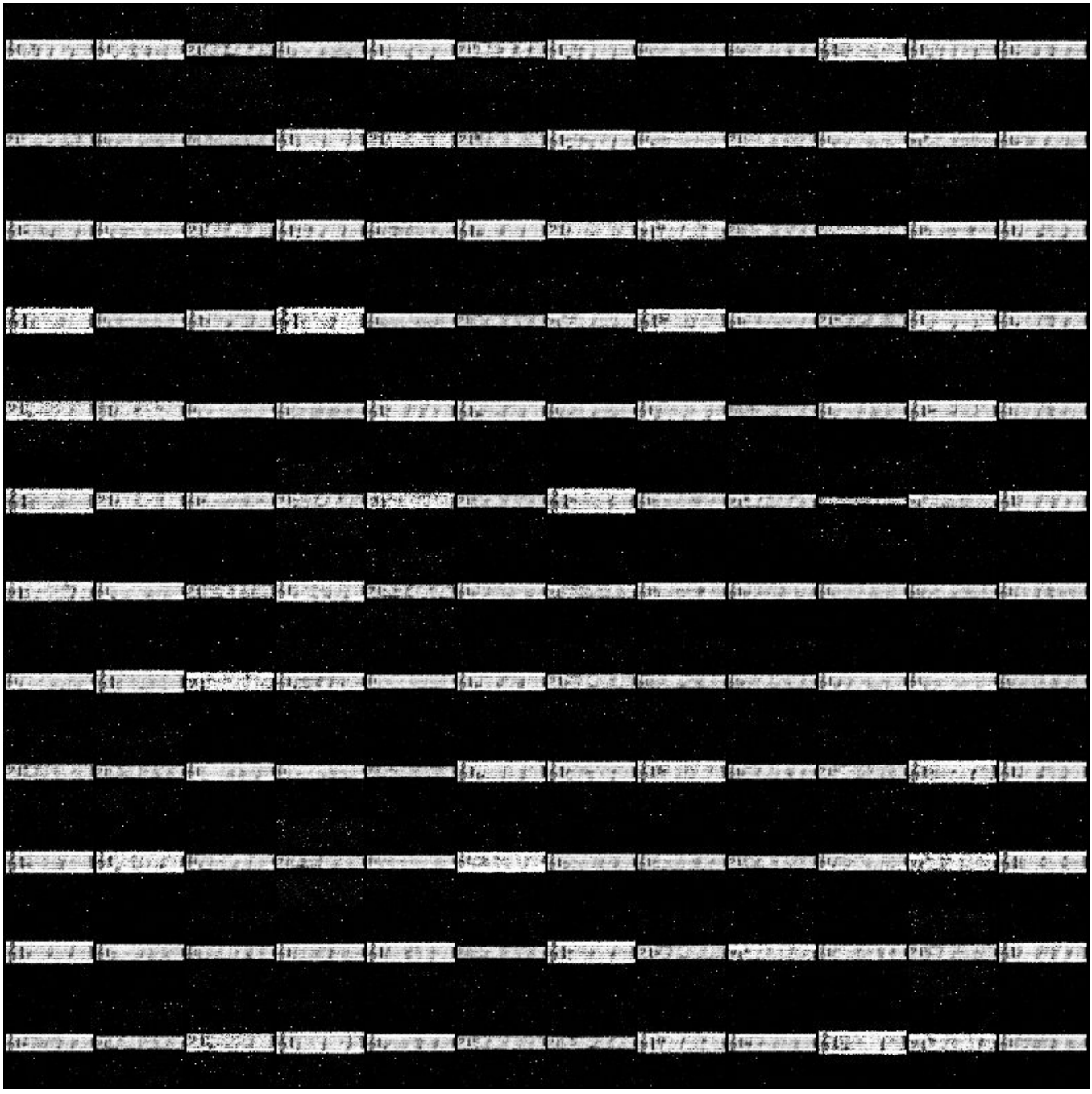


Figure 9. Results for CGAN model with linear layers & new dataset. Measured after ~500 epochs[different from Figure 6 which measures after 2100 epochs]. Default hyperparameters. {0: 'G', 1: 'B', 2: 'A, 3: 'D', 4: 'E', 5: 'F#', 6: 'C', 7: 'C#', 8: 'G#', 9: 'Eb', 10: 'Bb', 11: 'F'}

From a quick eyeballing, the CGAN performs slightly better than the vanilla GAN (i.e. you can see the shapes of notes a bit more clearly at epoch 500. I think it is likely that imposing conditionality sped up the training of the GAN (by giving more information about what should be in a given picture). This would certainly support our loss plot comparison from earlier. You

can view more of the generated images labelled by batch number here. Note that you should look only at the first column in each image, because all the images are labelled zero (to remove conditional information in a hacky manner). However, the distinction in results between the CGAN and DCGAN at epoch 500 is clearly not very strong, which is expected due to similar architecture.

Result 4: unconditional variant, old dataset

Finally, I wanted to see how much of a difference using the old dataset would make, so I tried out the unconditional version of the new code on the old data:

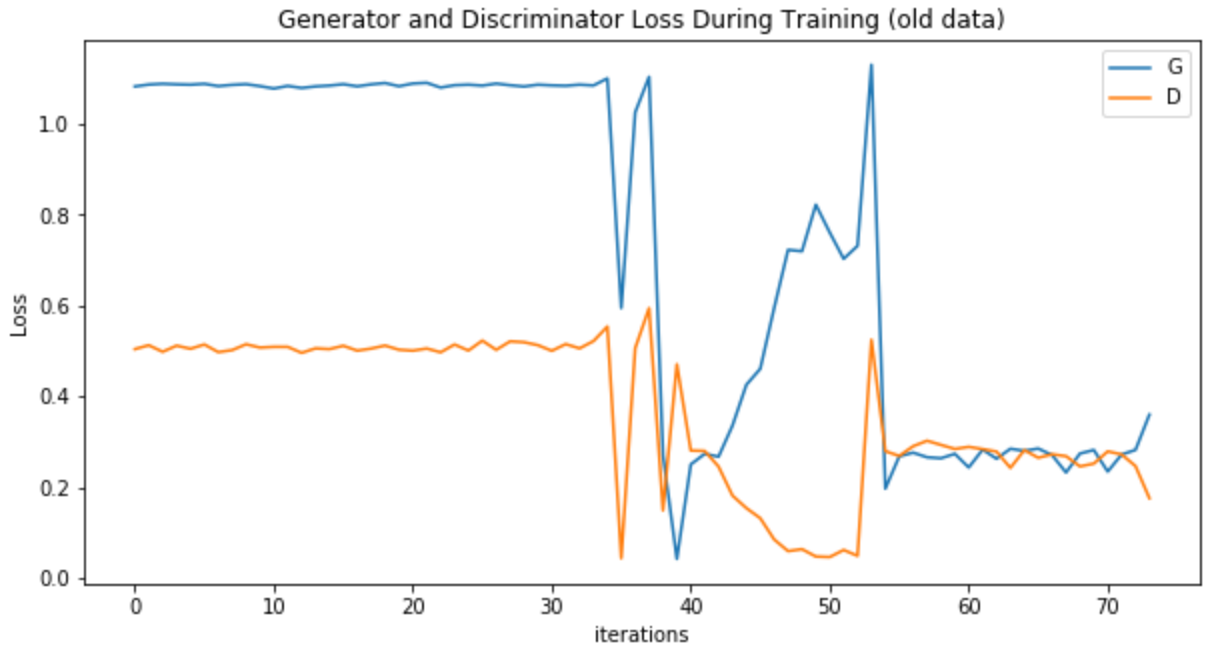


Figure 10. Loss for GAN model with linear layers & old* dataset. Measured after ~85 epochs. Default hyperparameters.

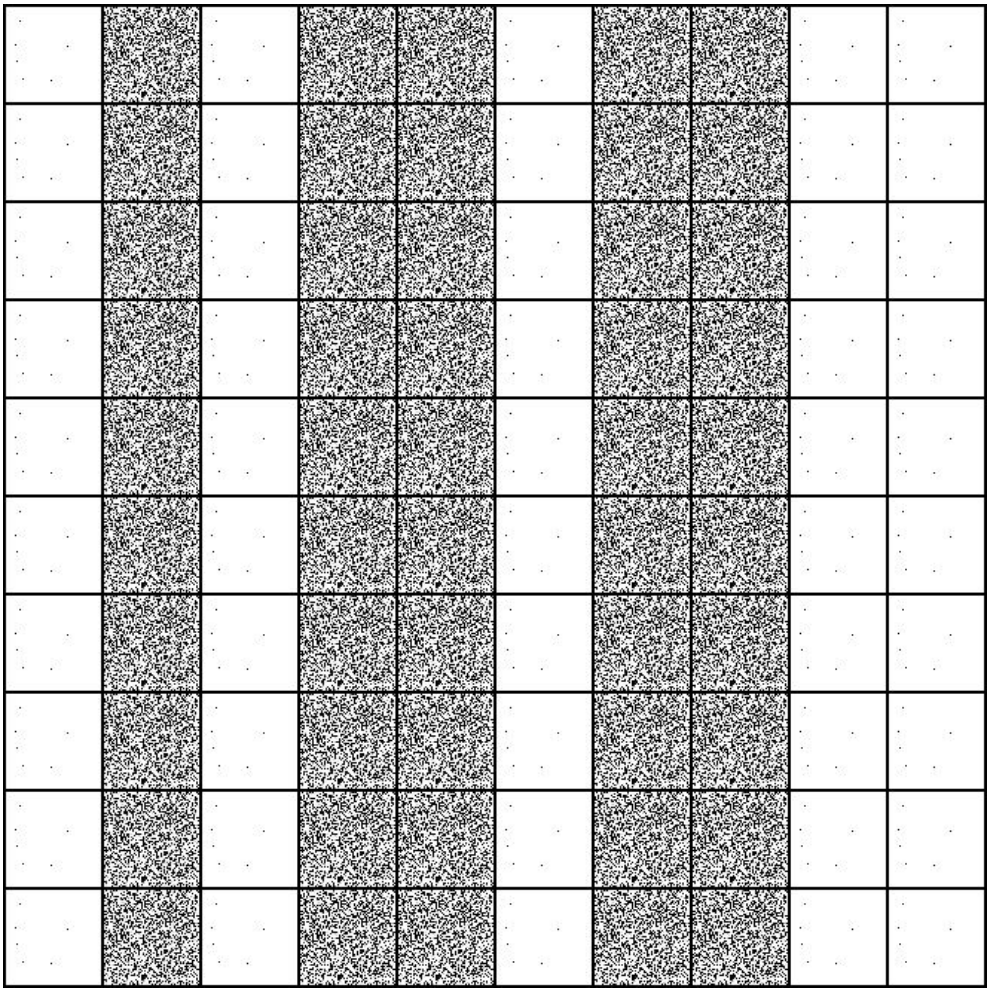


Figure 11. Results for GAN model with linear layers & old dataset. Measured after ~85 epochs. LR of discriminator set to one-hundredth that of discriminator, otherwise default parameters.

The discriminator kept overpowering, and in the end I set it to about one-hundredth that of the discriminator in order to stabilize it for a little while(though clearly it did not really work even at this low learning rate). As we can see, the results are by far the worst — even though you can see that playing with the discriminator learning rate did stabilize training for a little while– but this did not last, and there was a quick convergence to a local optimum. Admittedly, I only ran this model for 85 epochs, but the model trained on the newer dataset with this model had better results even in the first few epochs. This might be because the convolutions really did help the larger and dirtier dataset to learn, whilst the linear layers are insufficient here. In any case, the default parameters for the DCGAN worked well with the larger (~20000), dirtier dataset, whilst the default parameters for the linear GAN & linear DCGAN worked well on the smaller (~7500), cleaner dataset.

Concluding Remarks & Future Work

In conclusion, from my experiments, I think that the neural network with many convolutional layers works better on the larger & more complicated dataset, but works badly on the smaller and simpler dataset. This seems intuitively plausible because convolutions are able to capture the most important pieces of information, rather than take all the information in each picture. If the dataset is already quite clean, then this is less necessary, and can make us accidentally lose information that might make training quicker.

One unintuitive result from this whole series of experiments was that the linear layer model worked faster than the convolutional layer model. It could be that the structure of the convolutional model was still more taxing. But I am personally unconvinced by this argument because the only difference in the number of layers was that the convolutional layer model used one additional convolutional layer in the discriminator.

In terms of the CGAN, I would call it experimentally successful in that it did identify notes reasonably well in the 2100th epoch. However, it did still suffer from some degree of mode collapse. I sincerely think that more training would help, especially because the loss has not diverged too much between the discriminator and generator such that more training would be useless.

Future Work

First of all, I really want to find out why the linear model performed so much faster but still more accurately than the convolutional model — in hindsight it might be due to some unnecessary overhead that I inflicted on the convolutional model in terms of storing models quite frequently.

Second of all, I am excited to try out a Wasserstein GAN because it seems that it is a little easier to tell if a Wasserstein GAN converges due to its loss function (and a large part of time I spent on this assignment was wondering whether or not to click “interrupt kernel” and try again).

Thirdly, I am excited to expand my music generating pipeline in two ways:

1. add an OCR system at the end of the pipeline so that we can convert the resulting sheet music into midi to be played
2. Use more notes in one picture, but this might be too computationally taxing due to having to ensure a sufficiently good image dimension for the notes to be legible. Additionally, use multiple melody lines in one picture, to see if the GAN can learn to harmonize as well.

Fourthly, I want to continue to train my CGAN to see if the results do continue to improve.

References

eriklindernoren(2019). PyTorch-GAN. Cgan.py. Retrieved from https://github.com/eriklindernoren/PyTorch-GAN/blob/master/implementations/cgan/cgan.py

Inkawhich,N., 2017. DCGAN Tutorial. PyTorch. Retrieved from https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv preprint arXiv:1511.06434. Retrieved from https://arxiv.org/abs/1511.06434

Appendix

Link to results:
https://drive.google.com/drive/folders/10-ns6GDTRVrpr76qD0AGCMKdbdyNMVk2?usp=sharing

convert-score-2-image

December 21, 2019

In [1]:
```
%matplotlib inline
```

In [3]:
```
import argparse
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
from torch.utils.data import Dataset, DataLoader
from torch.autograd import Variable
```

In [4]:
```
import music21
import os
os.environ["musescoreDirectPNGPath"] = "/Applications/MuseScore\ 3/bin/MuseScore.exe"
os.environ["musicxmlPath"] = "/Applications/MuseScore\ 3/bin/MuseScore.exe"
song = music21.converter.parse('bach/chor002.midi')

from torch.utils.data import Dataset, DataLoader
from torch.autograd import Variable
import torchvision.transforms as transforms
import torch
import torchvision.utils as vutils

#import necessary modules, and set path to use musescore
```

In [296]:
```
#split images according to criterion specified
#i.e. four quarter notes per image
#then save to file
songs = []
```

```
        for i in range(1, 111):
            if len(str(i))==1:
                e = "00" + str(i)
            elif len(str(i))==2:
                e = "0" + str(i)
            else:
                e = str(i)
            song = music21.converter.parse('Downloads/chor{}.mid'.format(e))

            for voice in range(len(song)):
                prev = 0
                for note in range(len(song[voice].notes)):
                    if song[voice].notes.stream()[:note].highestOffset%3==0 and \
                    song[voice].notes.stream()[:note].highestOffset>0:
                        s1 = music21.stream.Stream()
                        for k in range(prev, note):
                            s1.append(song[voice].notes.stream()[k])
                        prev = note
                        tmp_path = s1.write(fmt = 'musicxml.png')
                        #write to permanent file,
                        #a hack bc music21's path specifying feature does not work for me
                        in_ = open(tmp_path, "rb")
                        with open("bach/img{}.png".format(e+str(voice)+str(note)), "wb")\
                        as file:
                            file.write(in_.read())

                in_.close()


        #         break

        #         song.flat.notes.stream()[:10].show()
```

In [61]:
```
        #finally, the png images have "transparencies"
        #which need to be changed into white pixels
        #(a gap in the PIL functionality that PyTorch uses)
        from PIL import Image
        import re

        for i in os.listdir("bach"):
            if re.match("img\d+.png", i) != None:
                # Load the image and make into Numpy array
                rgba = np.array(Image.open("bach/" + i))

                # Make image transparent white anywhere it is transparent
                rgba[rgba[...,-1]==0] = [255,255,255,0]

                # Make back into PIL Image and save
```

```
                Image.fromarray(rgba).save("bach/" + i)
```

In [62]:
```
#making labels for cgan
songs = []
file = open("labels.txt", "a+")
t = 0
for i in range(78, 79):
    if len(str(i))==1:
        e = "00" + str(i)
    elif len(str(i))==2:
        e = "0" + str(i)
    else:
        e = str(i)
    song = music21.converter.parse('bach2/chor{}.midi'.format(e))

    for voice in range(len(song)):
        prev = 0
        for note in range(len(song[voice].notes)):
            if song[voice].notes.stream()[:note].highestOffset%3==0 and\
            song[voice].notes.stream()[:note].highestOffset>0:
                s1 = music21.stream.Stream()
                for k in range(prev, note):
                    s1.append(song[voice].notes.stream()[k])
                if str(song[voice].notes.stream()[prev])[-2]=="#" \
                or str(song[voice].notes.stream()[prev])[-2]=="-":
                    file.write(e+str(voice)+str(note) + "\t" + \
                               str(song[voice].notes.stream()[prev])[-3:-1])
                else:
                    file.write(e+str(voice)+str(note) + "\t" + \
                               str(song[voice].notes.stream()[prev])[-2])
                file.write("\n")
                t+=1
#                 print(str(song[voice].notes.stream()[prev])[-2])
                prev = note
#                 tmp_path = s1.write(fmt = 'musicxml.png')


file.close()

#         in_.close()


#         break

#         song.flat.notes.stream()[:10].show()
```

CGAN Code-f

December 21, 2019

In [1]:
```
# # np.array([[1, 2, 3], [4, 5, 6]]).reshape(0, 3, 2)
# ! unzip cr-20191220T131909Z-001.zip
```

In [2]:
```
#adding padding so that we can resize to square without losing information
from PIL import ImageOps
from PIL import Image
import os

#finally, the png images have "transparencies"
#which need to be changed into white pixels
#(a gap in the PIL functionality that PyTorch uses)
from PIL import Image
import re

#https://stackoverflow.com/questions/44231209/resize-rectangular-image-to-square-keepi
# def make_square(im, min_size=256, fill_color=(0, 0, 0, 0)):
#     x, y = im.size
#     size = max(min_size, x, y)
#     new_im = Image.new('RGBA', (size, size), fill_color)
#     new_im.paste(im, (int((size - x) / 2), \
#                       int((size - y) / 2)))
#     return new_im

# for i in os.listdir\
# ("/mnt/disks/user/project/bach-images/bach2"):
#   if re.match("img\d+.png", i) \
# != None:
#     test_image = Image.open\
#     ("/mnt/disks/user/project/bach-images/bach2/" + i)
#     new_image = make_square(test_image)
#     new_image.save("/mnt/disks/user/project/bach/bach-images2/" + i)
```

In [3]:
```
%matplotlib inline
```

In [4]:
```
#get labels
import pandas as pd

label_frame = pd.read_csv("labels.txt", "\t", \
```

```
                                dtype=str, header = None, \
                                names = ["imageid", "note"])
        enc = [i for i in range(len(label_frame["note"].unique()))]
        dic = dict(zip(enc, label_frame["note"].unique()))
        for i in enc:
            label_frame.loc[label_frame["note"]==dic[i], "note"] = i
```

In [5]:
```
label_frame[:10]
len(enc)
print(dic)
```

```
{0: 'G', 1: 'B', 2: 'A', 3: 'D', 4: 'E', 5: 'F#', 6: 'C', 7: 'C#', 8: 'G#', 9: 'E-', 10: 'B-',
```

In [16]:
```
import argparse
import os
import numpy as np
import math

import torchvision.transforms as transforms
from torchvision.utils import save_image

from torch.utils.data import DataLoader
from torchvision import datasets
from torch.autograd import Variable

import torch.nn as nn
import torch.nn.functional as F
import torch

# os.makedirs("images", exist_ok=True)
os.makedirs("images-unc", exist_ok=True)
os.makedirs("GAN-Image-Folder-Unc", exist_ok=True)




#3422 images of high quality instead of the 20000 lower quality images
dataroot = "/mnt/disks/user/project/bach/bach-images2"
workers = 4
batch_size = 56
img_size = 64 #change this to 64 to replicate model 1
channels = 1 #color channels, i change to grayscale
latent_dim = 100 #noise vector dim
ngf = 10 #generator feature dim
ndf = 10 #discriminator feature dim
n_epochs =50
lr = 0.0002 #learning rate
```

```
# Beta1 hyperparam for Adam optimizers
b1 = 0.5
b2 = 0.999
ngpu = 1
n_classes=12
sample_interval = 400
conditional = False
loss_file = "loss.txt"
model_file = "GAN-Image-Folder2"
image_file = "images"



# size_data= len(os.listdir("bach/bach-images2"))

img_shape = (channels, img_size, img_size)

cuda = True if torch.cuda.is_available() else False


class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.label_emb = nn.Embedding(n_classes, n_classes)

        def block(in_feat, out_feat, normalize=True):
            layers = [nn.Linear(in_feat, out_feat)]
            if normalize:
                layers.append(nn.BatchNorm1d(out_feat, 0.8))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers

        self.model = nn.Sequential(
            *block(latent_dim + n_classes, 128, normalize=False),
            *block(128, 256),
            *block(256, 512),
            *block(512, 1024),
            nn.Linear(1024, int(np.prod(img_shape))),
            nn.Tanh()
        )

    def forward(self, noise, labels):
        # Concatenate label embedding and image to produce input
        gen_input = torch.cat((self.label_emb(labels), noise), -1)
        img = self.model(gen_input)
        img = img.view(img.size(0), *img_shape)
        return img
```

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.label_embedding = nn.Embedding(n_classes, n_classes)

        self.model = nn.Sequential(
            nn.Linear(n_classes + int(np.prod(img_shape)), 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 512),
            nn.Dropout(0.4),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 512),
            nn.Dropout(0.4),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 1),
        )

    def forward(self, img, labels):
        # Concatenate label embedding and image to produce input
        d_in = torch.cat((img.view(img.size(0), -1), \
                          self.label_embedding(labels)), -1)
        validity = self.model(d_in)
        return validity


# Loss functions
adversarial_loss = torch.nn.MSELoss()

# Initialize generator and discriminator
generator = Generator()
discriminator = Discriminator()

if cuda:
    generator.cuda()
    discriminator.cuda()
    adversarial_loss.cuda()
```

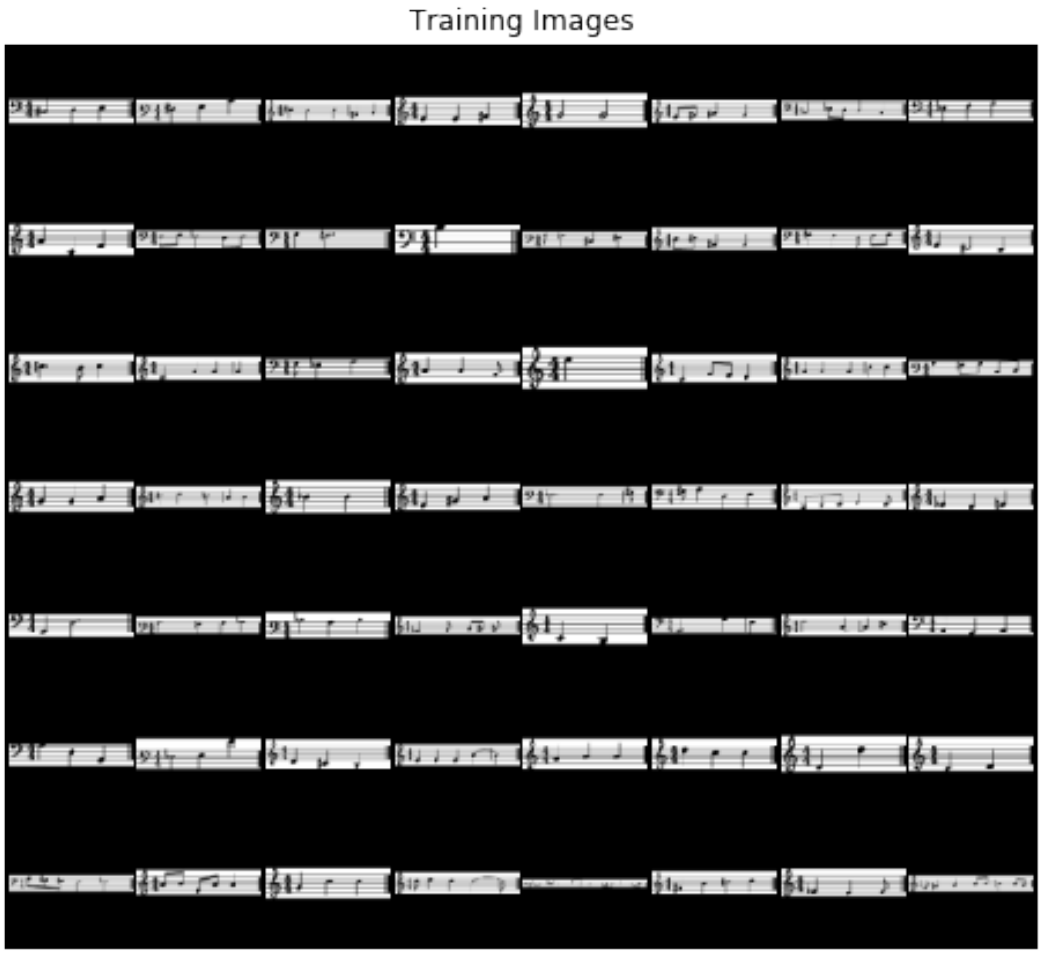
In [17]:
```
import re
from PIL import Image
from torch.utils.data import Dataset, DataLoader
from torch.autograd import Variable
import matplotlib.pyplot as plt
import torchvision.utils as vutils

class Bach(Dataset):
```

```
    def __init__(self, transform=None):
        self.transform = transform

        #placeholder
        self.images = []
        labels = []
        for i in os.listdir(dataroot):
#                if re.match("img\d\d+.png", i)!=None:
#                  print(i)
                 with Image.open(dataroot + "/"+i).convert('L')\
    as image:

                     self.images.append(np.array(image).astype\
                                        ("uint8"))
                     if conditional==True:
                         if len(label_frame[label_frame.iloc[:,0]\
                                            ==i[3:i.find(".")]])==0:
                             print("o boi", i)
                         labels.append(label_frame[label_frame.iloc\
                                                   [:,0]==i[3:i.find(".")]].\
                                       iloc[:,1:].iloc[0])
        if conditional==False:
            self.labels = Variable(torch.LongTensor(np.zeros\
                                                    (len(self.images))\
                                                    .astype("uint8").\
                                                    flatten()))
        else:
            self.labels = Variable(torch.LongTensor(np.array(labels).\
                                                    astype("uint8").\
                                                    flatten()))

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        label = self.labels[idx]
        img = Image.fromarray(self.images[idx])

        if self.transform:
            img = self.transform(img)

        return img, label

print("transforms")
transform = transforms.Compose([
                            transforms.Grayscale(num_output_channels=1),
                            #note that I load in grayscale
                             transforms.Resize(img_size),
```

```
                                    transforms.CenterCrop(img_size),
                                    transforms.ToTensor(),
                                   transforms.Normalize([0.5], [0.5])
                                ])
        dataset = Bach(transform=transform)

        print("loader")
        dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,\
                                                  shuffle=True, \
                                                   num_workers = 4)
        device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0)\
                              else "cpu")

        print("batch load")
        real_batch = next(iter(dataloader))
        plt.figure(figsize=(8,8))
        plt.axis("off")
        plt.title("Training Images")
        plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64],\
                                                  padding=2, normalize=True).cpu(),\
                                (1,2,0)))
```

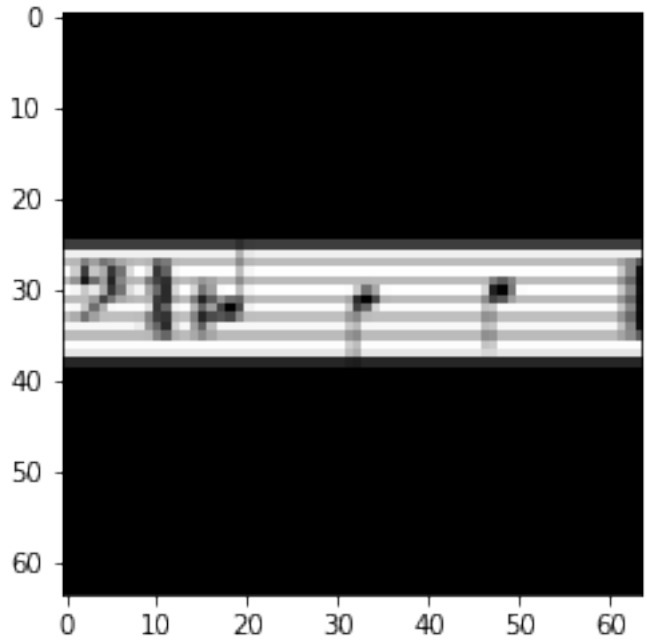
```
transforms
loader
batch load
```

Out[17]: <matplotlib.image.AxesImage at 0x7ff6343eb278>



In [18]:
```
#one such image
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].\
                                         to(device)[:1],\
                                         padding=2, \
                                         normalize=True).cpu(),\
                        (1,2,0)))
```

Out[18]: <matplotlib.image.AxesImage at 0x7ff632a82fd0>



In [19]:
```
# Optimizers
optimizer_G = torch.optim.Adam(generator.parameters(),\
                               lr=lr, betas=(b1, b2))
optimizer_D = torch.optim.Adam(discriminator.parameters(),\
                               lr=lr, betas=(b1, b2))

FloatTensor = torch.cuda.FloatTensor if cuda else torch.FloatTensor
LongTensor = torch.cuda.LongTensor if cuda else torch.LongTensor

def sample_image(n_row, batches_done, loss, loss_file):
    """Saves a grid of generated digits ranging from 0 to n_classes"""
    # Sample noise
    z = Variable(FloatTensor(np.random.normal\
                             (0, 1, (n_row ** 2, latent_dim))))
    # Get labels ranging from 0 to n_classes for n rows
    labels = np.array([num for _ in range(n_row)\
                       for num in range(n_row)])
    labels = Variable(LongTensor(labels))
    gen_imgs = generator(z, labels)
    save_image\
    (gen_imgs.data, image_file+"/%d.png" % batches_done, nrow=n_row,\
     normalize=True)
    with open(loss_file, "a+") \
    as loss_file:
        loss_file.write(str(loss[0])+ "\t"+ str(loss[1]) )
        loss_file.write("\n")
```

In [21]:
```
# ----------
#  Training
# ----------

#you do not see outputs bc i commented removed it to make the notebook neater
#it was just printing out information such as loss from each epoch

n_epochs = 500

for epoch in range(0, n_epochs):

    for i, (imgs, labels) in enumerate(dataloader):

        batch_size = imgs.shape[0]

        # Adversarial ground truths
        valid = Variable(FloatTensor(batch_size, 1).fill_(1.0),\
                         requires_grad=False)
        fake = Variable(FloatTensor(batch_size, 1).fill_(0.0),\
                        requires_grad=False)

        # Configure input
        real_imgs = Variable(imgs.type(FloatTensor))
        labels = Variable(labels.type(LongTensor))

        # ---------------
        #  Train Generator
        # -----------------

        optimizer_G.zero_grad()

        # Sample noise and labels as generator input
        z = Variable(FloatTensor(np.random.normal(0, 1, \
                                                  (batch_size, latent_dim))))
        gen_labels = Variable(LongTensor(np.random.randint\
                                         (0, n_classes, batch_size)))

        # Generate a batch of images
        gen_imgs = generator(z, gen_labels)

        # Loss measures generator's ability to fool the discriminator
        validity = discriminator(gen_imgs, gen_labels)
        g_loss = adversarial_loss(validity, valid)

        g_loss.backward()
        optimizer_G.step()

        # ---------------------
```

```
                #  Train Discriminator
                # ---------------------

                optimizer_D.zero_grad()

                # Loss for real images
                validity_real = discriminator(real_imgs, labels)
                d_real_loss = adversarial_loss(validity_real, valid)

                # Loss for fake images
                validity_fake = discriminator(gen_imgs.detach(), gen_labels)
                d_fake_loss = adversarial_loss(validity_fake, fake)

                # Total discriminator loss
                d_loss = (d_real_loss + d_fake_loss) / 2

                d_loss.backward()
                optimizer_D.step()

                print(
                    "[Epoch %d/%d] [Batch %d/%d] [D loss: %f] [G loss: %f]"
                    % (epoch, n_epochs, i, len(dataloader), \
                       d_loss.item(), g_loss.item())
                )

                batches_done = epoch * len(dataloader) + i
                if batches_done % sample_interval == 0:
                    sample_image(n_row=12, batches_done=batches_done,\
                                 loss=[g_loss.item(), d_loss.item()],\
                                 loss_file=loss_file)

                    torch.save(generator.state_dict(), \
                                 model_file+"/netG".\
                                 format(batches_done))
                    torch.save(discriminator.state_dict(), \
                                 model_file+"/netD".\
                                 format(batches_done))
```

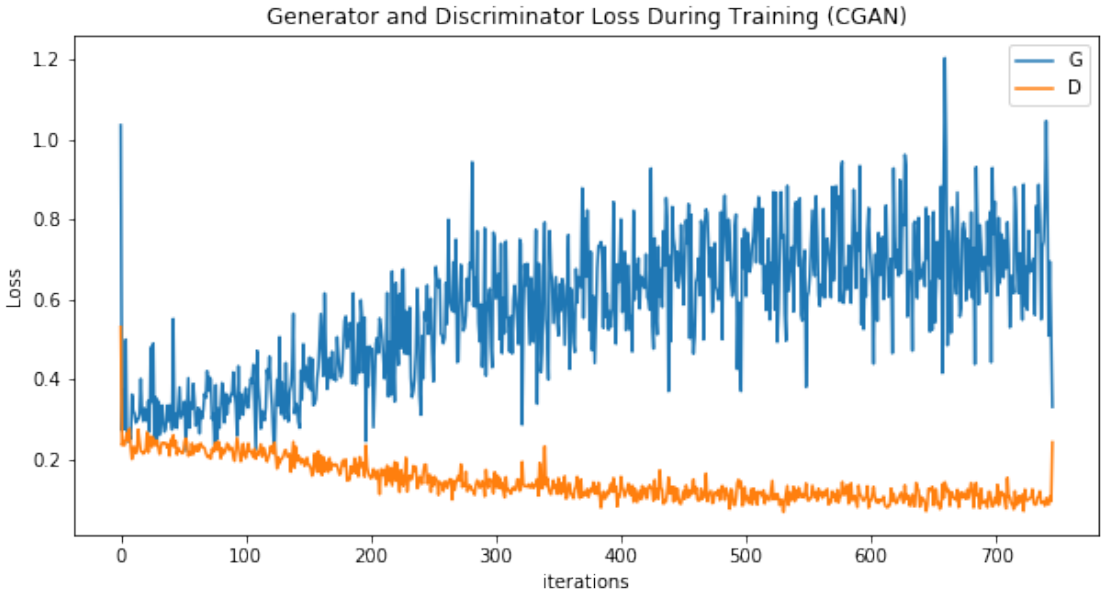
In [59]:
```
i = "img00103.png"
print(np.array([0, 1]).shape)

print(np.random.randint(0, 12, 7521).shape)
```

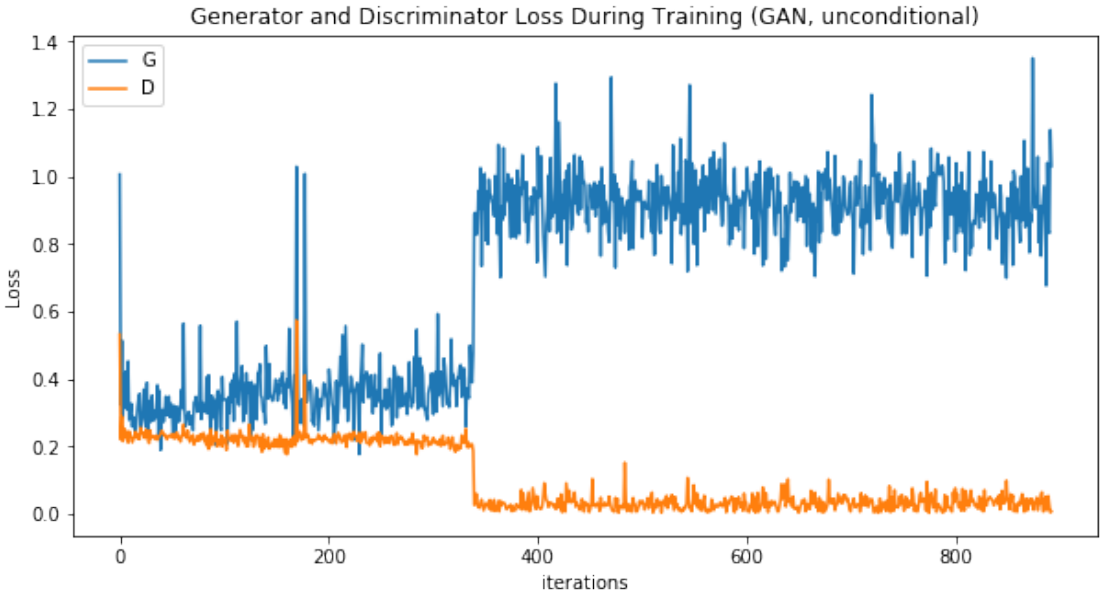
```
(2,)
(7521,)
```

In [53]:
```
# import shutil
# shutil.make_archive("images-unc", 'zip', "images-unc")
```

```
Out[53]: '/mnt/disks/user/project/images-unc.zip'
```

```
In [65]: for f in ["loss.txt", "loss-unc.txt"]:
             G_losses = []
             D_losses = []
             with open(f, "r") as file:
                 for i in file.readlines():
                     a = i.split("\t")
                     G_losses.append(float(a[0]))
                     D_losses.append(float(a[1][:-1]))


             plt.figure(figsize=(10,5))
             if f=="loss.txt":
                 plt.title("Generator and Discriminator Loss During Training (CGAN)")
             else:
                 plt.title("Generator and Discriminator Loss During Training (GAN, unconditiona
             plt.plot(G_losses,label="G")
             plt.plot(D_losses,label="D")
             plt.xlabel("iterations")
             plt.ylabel("Loss")
             plt.legend()
             plt.show()
             # D_losses
```




Generator and Discriminator Loss During Training (GAN, unconditional)
G
D
Loss
iterations
0
200
400
600
800
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4

Original DCGAN Code

December 21, 2019

In [2]:
```
%matplotlib inline
import os
#code inspired from
#https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html
```

In [3]:
```
from __future__ import print_function
#%matplotlib inline
import argparse
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

#for reproducibility, important for the
#fixed noise to be the same
manualSeed = 999
random.seed(manualSeed)
torch.manual_seed(manualSeed)
```

Out[3]:
```
<torch._C.Generator at 0x7f51ba039870>
```

In [4]:
```
dataroot = "/mnt/disks/user/project/bach"
workers = 4
batch_size = 56
image_size = 64
nc = 1 #color channels, i change to grayscale
nz = 100 #noise vector dim
ngf = 64 #generator feature dim
```

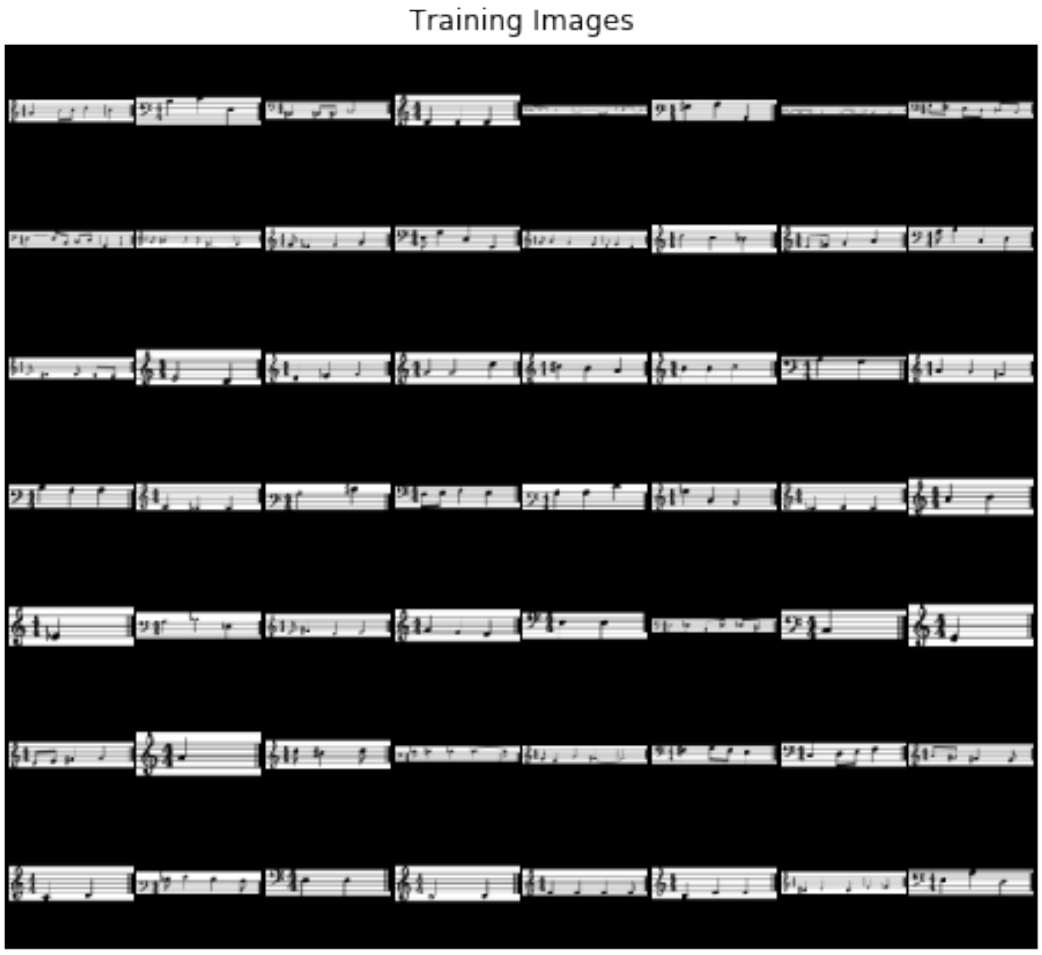
```
        ndf = 64 #discriminator feature dim
        num_epochs =50
        lr = 0.0002 #learning rate
        # Beta1 hyperparam for Adam optimizers
        beta1 = 0.5
        ngpu = 1
```

In [5]:
```
len(os.listdir(dataroot))
```

Out[5]: 3

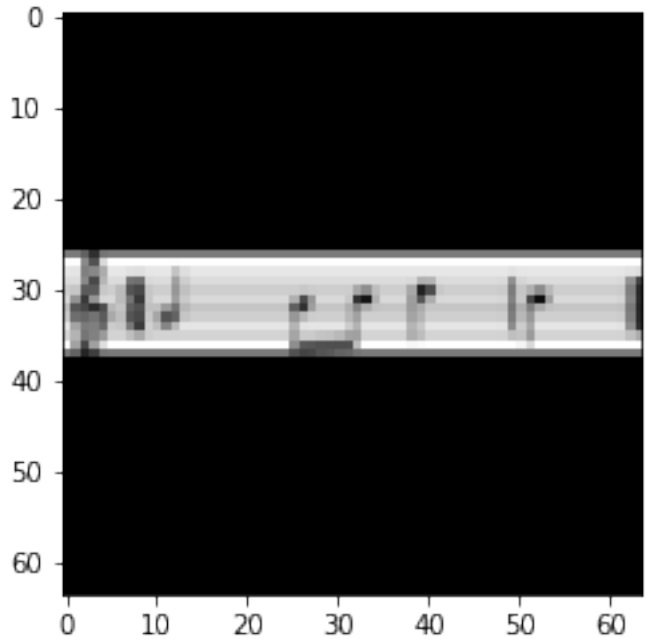
In [6]:
```
#I use the ImageFolder function which locates images
#in subfolders and puts them into our dataset.
#mostly modified this from tutorial
dataset = dset.ImageFolder(root=dataroot,
                           #
                           transform=transforms.Compose([
                              transforms.Grayscale(num_output_channels=1),
                              #note that I load in grayscale
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                               transforms.ToTensor(),
                              transforms.Normalize([0.5], [0.5])
                              #add grayscale & change normalization
                          #  transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                           ]))

dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)

device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0)\
                      else "cpu")

real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64],\
                                         padding=2, normalize=True).cpu(),\
                        (1,2,0)))
```

Out[6]: <matplotlib.image.AxesImage at 0x7f51b3fc1080>




```
In [7]: plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:1],\
                                                padding=2, normalize=True).cpu(),\
                                (1,2,0)))
```

```
Out[7]: <matplotlib.image.AxesImage at 0x7f51b3fb4e80>
```



In [8]:
```
#this function allows us to generalize all the weights
#for the discriminator and generator network.
#Note that this is different from the generator input
#which is also generated randomly.
# note that we initialize weights differently for the
#convolutional layers vs the batch norm layers i.e.
# for the convolutional layer,we do not need to have a
#randomly initialized bias
#because we introduce the bias in the batchnorm layers

def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
```

In [9]:
```
#most of this code is from the tutorial but i add the larger image cases
class Generator(nn.Module):
    def __init__(self, ngpu, image_size):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        if image_size==64: # i defined separate networks for the
        #64 and 128 dim image cases so we
        # can easily try both without constantly changing the
```

```
#dimensions of our layers
  self.main = nn.Sequential(
      nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
      nn.BatchNorm2d(ngf * 8),
      nn.ReLU(True),

      nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
      nn.BatchNorm2d(ngf * 4),
      nn.ReLU(True),

      nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
      nn.BatchNorm2d(ngf * 2),
      nn.ReLU(True),

      nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
      nn.BatchNorm2d(ngf),
      nn.ReLU(True),

      nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
      nn.Tanh()
  )
elif image_size ==128:
    self.main = nn.Sequential(

      nn.ConvTranspose2d( nz, ngf * 16, 4, 1, 0, bias=False),
      nn.BatchNorm2d(ngf * 16),
      nn.ReLU(True),

      nn.ConvTranspose2d( ngf * 16, ngf * 8, 4, 2, 1, bias=False),
      nn.BatchNorm2d(ngf * 8),
      nn.ReLU(True),

      nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
      nn.BatchNorm2d(ngf * 4),
      nn.ReLU(True),

      nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
      nn.BatchNorm2d(ngf * 2),
      nn.ReLU(True),

      nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
      nn.BatchNorm2d(ngf),
      nn.ReLU(True),

      nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False), #because input
      nn.Tanh() #tanh activation (so the output will be between 1 and -1)
  )
```

```
            def forward(self, input):
                return self.main(input)

In [10]: netG = Generator(ngpu, image_size).to(device)

         if (device.type == 'cuda') and (ngpu > 1):
             netG = nn.DataParallel(netG, list(range(ngpu)))
             #i only use one GPU

         netG.apply(weights_init)

         # Print the model
         print(netG)

Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=Fals
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=Fals
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=Falso
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace)
    (12): ConvTranspose2d(64, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)


In [3]: #most of this code is from the tutorial but i add the 128x128 case
        class Discriminator(nn.Module):
            def __init__(self, ngpu, image_size): #added image_size param for ease
                super(Discriminator, self).__init__()
                self.ngpu = ngpu
                if image_size==64:
                  self.main = nn.Sequential(
                      nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
                      nn.LeakyReLU(0.2, inplace=True),

                      nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
```

```
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )
    elif image_size == 128:
        self.main = nn.Sequential(
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),

            # nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            # nn.BatchNorm2d(ndf * 2),
            # nn.LeakyReLU(0.2, inplace=True),

            # nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            # nn.BatchNorm2d(ndf * 4),
            # nn.LeakyReLU(0.2, inplace=True), #the layers i removed to
            #speed up computation are commented out

            nn.Conv2d(ndf, ndf * 2, 4, 4, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),

            # nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            # nn.BatchNorm2d(ndf * 8),
            # nn.LeakyReLU(0.2, inplace=True),

            # nn.Conv2d(ndf * 8, ndf * 16, 4, 2, 1, bias=False),
            # nn.BatchNorm2d(ndf * 16),
            # nn.LeakyReLU(0.2, inplace=True),
            #those commented out are the removed layers or
            #computational eff.
            nn.Conv2d(ndf * 2, ndf * 4, 4, 4, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf * 4, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )
```

```
            def forward(self, input):
                return self.main(input)
```

In [12]:
```
#most of this code is from the tutorial
netD = Discriminator(ngpu, image_size).to(device)


if (device.type == 'cuda') and (ngpu > 1):
    netD = nn.DataParallel(netD, list(range(ngpu)))

netD.apply(weights_init)

print(netD)
```

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(1, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

In [13]:
```
#most of this code is from the tutorial
criterion = nn.BCELoss()

#latent vector for input into generator
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

real_label = 0.95
fake_label = 0.05# smoothing labels

optimizerD = optim.Adam(netD.parameters(), lr=lr/2, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
```

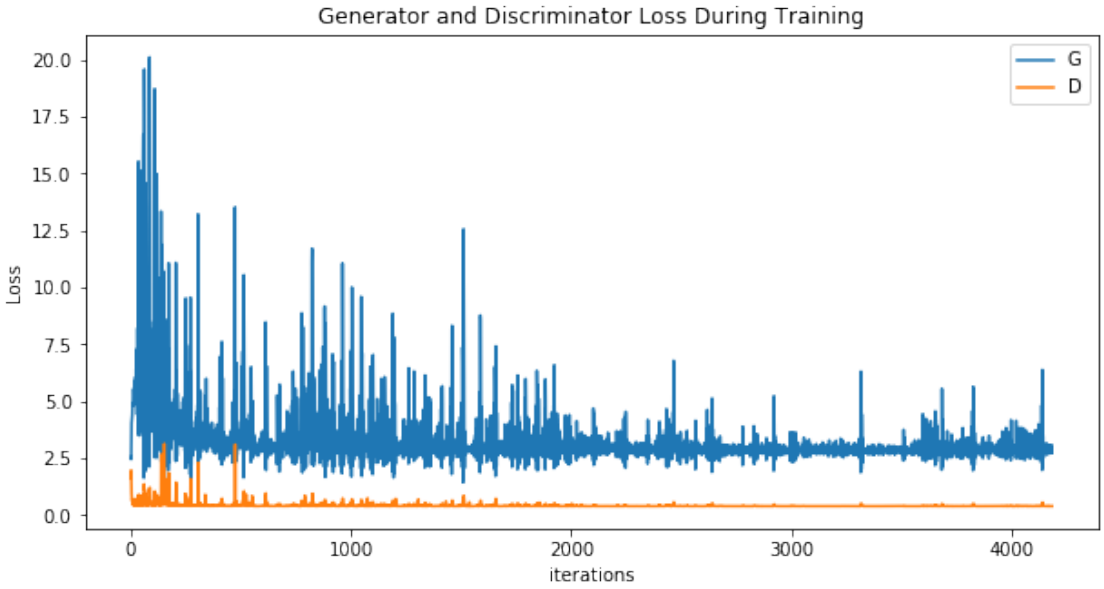
In [1]:
```
#you do not see outputs because i removed it for submission
def return_model_for_training(directory, epoch, image_size):
```

```
  #wrote this function to continue training if i stop at some epoch
      netG = Generator(ngpu, image_size).to(device)

      if (device.type == 'cuda') and (ngpu > 1):
          netG = nn.DataParallel(netG, list(range(ngpu)))

      netG.apply(weights_init)

      netD = Discriminator(ngpu, image_size).to(device)

      if (device.type == 'cuda') and (ngpu > 1):
          netD = nn.DataParallel(netD, list(range(ngpu)))

      netD.apply(weights_init)

      state_dictG = torch.load(directory +"/netG{}".format(epoch))
      state_dictD = torch.load(directory +"/netD{}".format(epoch))

      netG.load_state_dict(state_dictG)
      netD.load_state_dict(state_dictD)
      return netG, netD# # Training Loop #most of this code is from the tutorial
from timeit import default_timer
time_st = default_timer()
from PIL import Image

latest =0
# netG, netD = return_model_for_training\
# ("GAN-Image-FolderOri",\
#  latest, image_size)
# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):
        netD.zero_grad()
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size,), real_label, device=device)
        # Forward propagation
        output = netD(real_cpu).view(-1)
        # loss
        errD_real = criterion(output, label)
```

```
# gradients
errD_real.backward()
D_x = output.mean().item()

# Generate batch of latent vectors
noise = torch.randn(b_size, nz, 1, 1, device=device)
# Generate fake image batch with G
fake = netG(noise)
label.fill_(fake_label)
# Classify all fake batch with D
output = netD(fake.detach()).view(-1)
# Calculate D's loss on the all-fake batch
errD_fake = criterion(output, label)
# Calculate the gradients for this batch
errD_fake.backward()
D_G_z1 = output.mean().item()
# Add the gradients from the all-real and all-fake batches
errD = errD_real + errD_fake
# Update D
optimizerD.step()

netG.zero_grad()
label.fill_(real_label)
# forward pass through D
output = netD(fake).view(-1)
# loss
errG = criterion(output, label)
# gradients
errG.backward()
D_G_z2 = output.mean().item()
# Update
optimizerG.step()
if i % 50 == 0:
    print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x):\
     %.4f\tD(G(z)): %.4f / %.4f'
          % (epoch, num_epochs, i, len(dataloader),
             errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

# save losses
G_losses.append(errG.item())
D_losses.append(errD.item())

if (epoch % 5 == 0) or ((epoch == num_epochs-1) and (i == len(dataloader)-1)):
    with torch.no_grad():
        fake = netG(fixed_noise).detach().cpu()
    img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

#i add code to save models every x epochs (set to 1 now)
```

```
            if (epoch % 1 == 0):
              torch.save(netG.state_dict(), \
                         "GAN-Image-FolderOri/netG{}".\
                         format(epoch+latest+1))
              torch.save(netD.state_dict(), \
                         "GAN-Image-FolderOri/netD{}".\
                         format(epoch+latest+1))
            iters += 1

        print(default_timer()-time_st)
```

In [23]:
```
plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



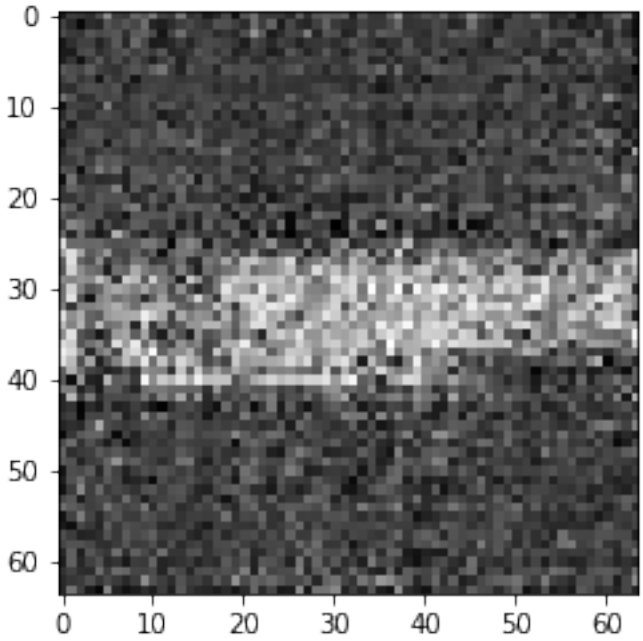
In [ ]:
```
real_batch = next(iter(dataloader))

# real
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
plt.axis("off")
plt.title("Real Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64],\
```

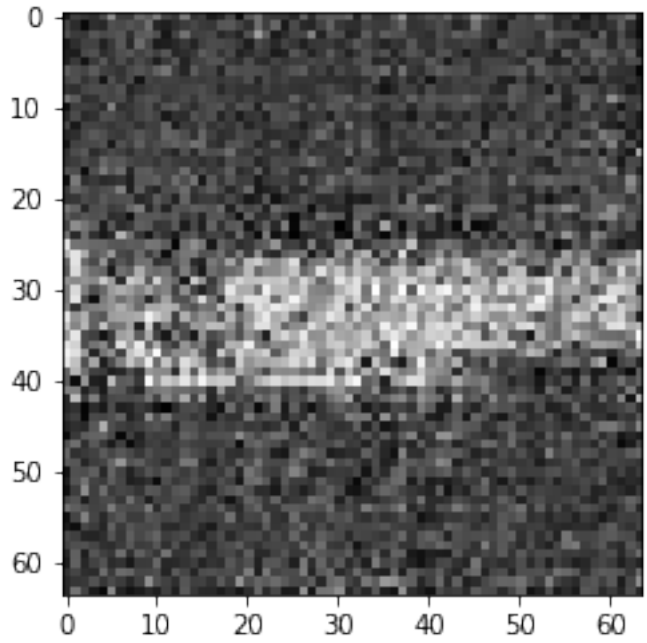
```
                                                    padding=5, normalize=True).cpu(),\
                                (1,2,0)))

        # fake from last epoch
        plt.subplot(1,2,2)
        plt.axis("off")
        plt.title("Fake Images")
        plt.imshow(np.transpose(img_list[-1],(1,2,0)))
        plt.show()
        #e12
```

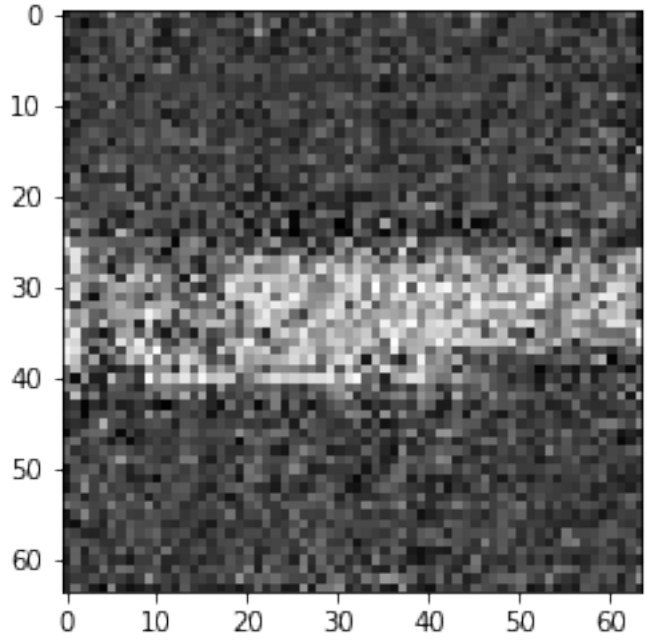
In [15]:
```
#i wrote this function to allow me to look at fake images generated by
 # old stored models
 def get_fake_for_specific_model(directory, image_size):
   lossG = []
   lossD = []
   fake_images = []
   for epoch in range(1, int(len(os.listdir(directory))/2)):
     if epoch%1==0:
       netG = Generator(ngpu, image_size).to(device)

       if (device.type == 'cuda') and (ngpu > 1):
           netG = nn.DataParallel(netG, list(range(ngpu)))

       netG.apply(weights_init)

       netD = Discriminator(ngpu, image_size).to(device)

       if (device.type == 'cuda') and (ngpu > 1):
           netD = nn.DataParallel(netD, list(range(ngpu)))

       netD.apply(weights_init)

       #we load saved models, we have try and except because
       #the last epoch
       #is often incomplete because I interrupt it
       try:
         state_dictG = torch.load(directory +"/netG{}".format(epoch))
         state_dictD = torch.load(directory +"/netD{}".format(epoch))
       except RuntimeError:
         print(epoch)
         next
       netG.load_state_dict(state_dictG)
       netD.load_state_dict(state_dictD)

       #noise
       fixed_noise = torch.randn(61, nz, 1, 1, device=device)
       with torch.no_grad():
```

```
            #make fake batch
            fake = netG(fixed_noise).detach().cpu()
            fake_images.append(fake)
            # break
        del netG
        del netD
    return fake_images

#only visualize every epoch
# we can choose which directory we want to visualize fakes

directory = "GAN-Image-FolderOri"
fake_images = get_fake_for_specific_model(directory, image_size)
```

In [16]:
```
import matplotlib.pyplot as plt
plt.imshow(np.transpose(vutils.make_grid(fake_images[-1][16], padding=2,\
                                         normalize=True), (1, 2, 0)))
plt.show()
plt.imshow(np.transpose(vutils.make_grid(fake_images[-1][10], padding=2, \
                                         normalize=True), (1, 2, 0)))
plt.show()
plt.imshow(np.transpose(vutils.make_grid(fake_images[-1][30], padding=2, \
                                         normalize=True), (1, 2, 0)))
plt.show()
```








```
In [1]: # for i in range(len(fake_images)):
        #   plt.imshow(np.transpose(vutils.make_grid(fake_images[i], \
        #                                            padding=2, normalize=True)\
```

```
#                             , (1, 2, 0)))
#   plt.title("epoch {}".format(i))
#   plt.show()
# #downloaded these images and used online tool to make the gifs in report
```