Honey, Andy

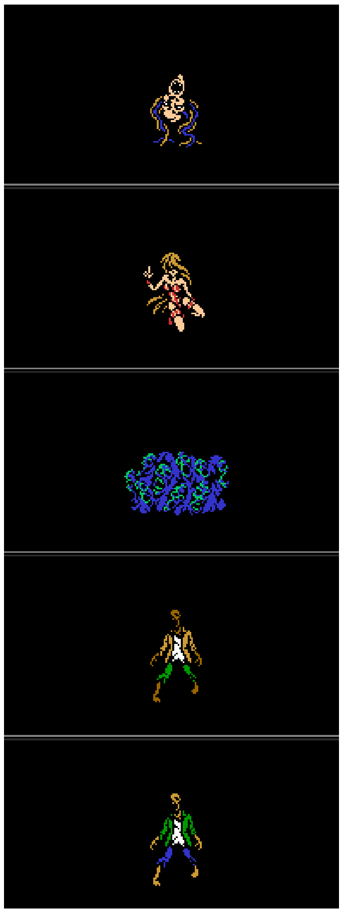
CSC 578

Class Project Report

Introduction

My project is using Option #1, wherein I will create a GAN to produce new images of demons based on the designs of the spritework of Shin Megami Tensei series.

The dataset being used is the compilation of sprites at http://f46.aaacafe.ne.jp/~aqul/aton/. The sprites used were from SMT1 and SMT2, examples of which can be seen on the left below. These two games were chosen over the others for the grittier designs and similar theming, rather than games such as demikids (right).



Data Cleaning

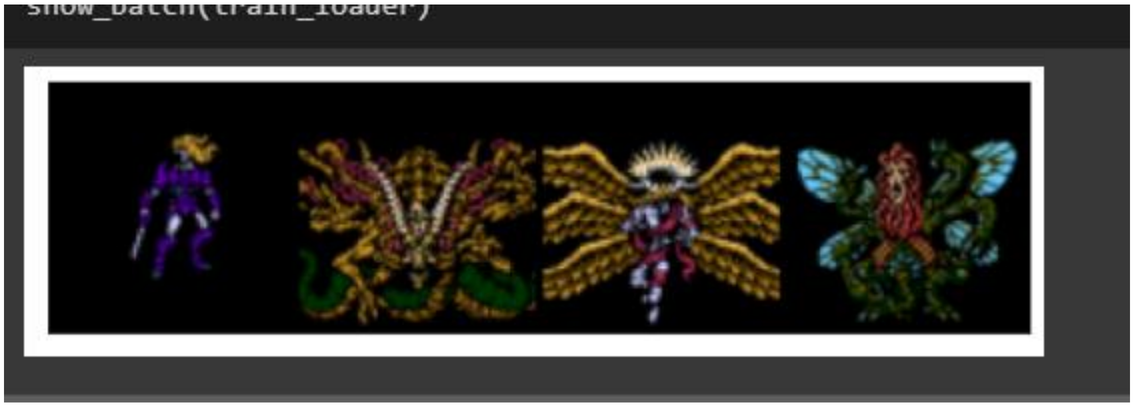
Because all of the sprites are animated, they came in the format of GIF images. For this project, I wanted to work only with PNG images, and used the following script in a locally run Jupyter notebook to convert the first frame of each to a still image.

```python
import glob
import os
from PIL import Image

files = glob.glob("/Users/User/Desktop/AI/Project/SMT1/All/Files/*.gif")

for imageFile in files:
    print ("Processing: " + imageFile)
    filepath,filename = os.path.split(imageFile)
    filterame,exts = os.path.splitext(filename)
    try:
        im = Image.open(imageFile)
#          im.save( "/Users/User/Desktop/AI/Project/SMT1/Neutral/New/"+filterame+'.png', "PNG" )
        im.save( "All/"+filterame+'.png', "PNG" )
    except Exception as exc:
        print ("Error: " + str(exc))

```

With this dataset cleaned, I moved to Colab to build the GAN notebook. Firstly, because there was a slight difference in the size of the sprites from both games, as well as a few special cases, each image was resized and cropped for each dataset. Each was then a 64x64 pixel image.

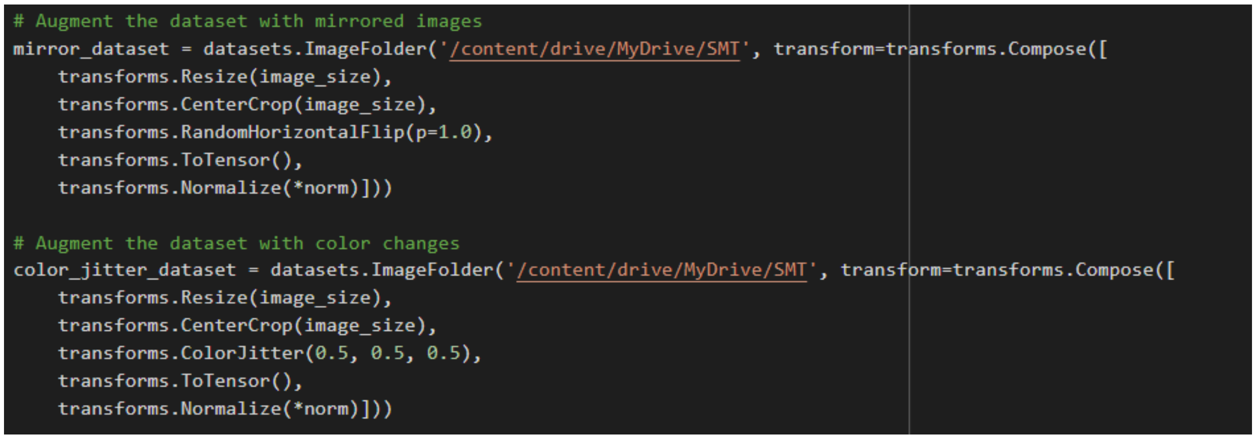
```python
normal_dataset = datasets.ImageFolder('/content/drive/MyDrive/SMT', transform=
    transforms.Compose([
    transforms.Resize(image_size),
    transforms.CenterCrop(image_size),
    transforms.ToTensor(),
    transforms.Normalize(*norm)]))
```

I was concerned that I may be causing data loss by doing this, however. To check this, I printed a test batch of images to make sure I knew exactly what it was doing to them. The result showed that a few of the very large ones lost the tips of their limbs, but nothing too extreme. The majority of the dataset included was not affected by this change.



```
show_batch(train_loader)
```

Another decision made was to mirror the data and invert the colors on them to create a larger data pool, similar to the Pokegan project I was studying.

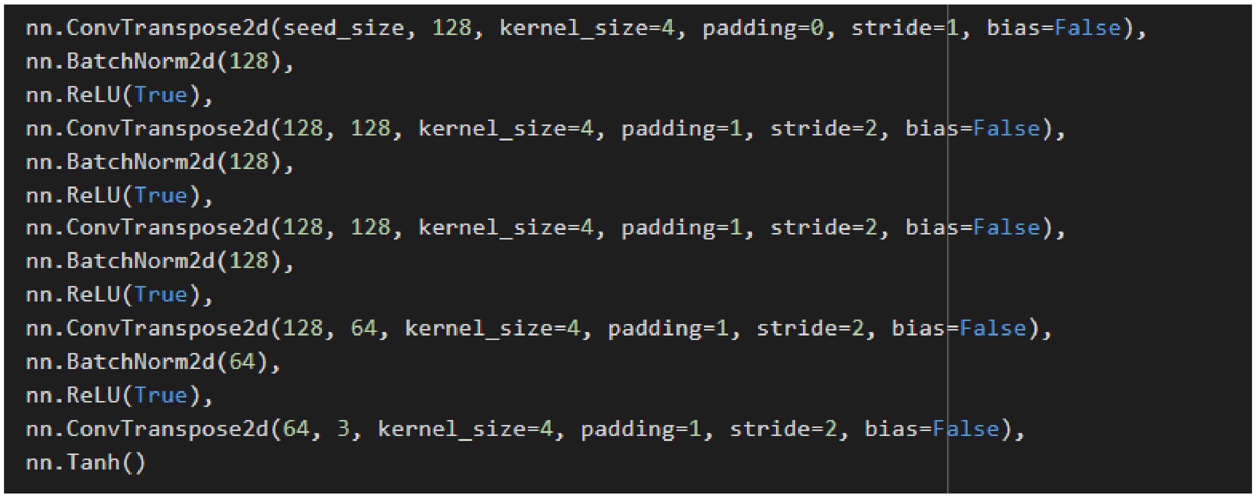


```
# Augment the dataset with mirrored images
mirror_dataset = datasets.ImageFolder('/content/drive/MyDrive/SMT', transform=transforms.Compose([
    transforms.Resize(image_size),
    transforms.CenterCrop(image_size),
    transforms.RandomHorizontalFlip(p=1.0),
    transforms.ToTensor(),
    transforms.Normalize(*norm)]))

# Augment the dataset with color changes
color_jitter_dataset = datasets.ImageFolder('/content/drive/MyDrive/SMT', transform=transforms.Compose([
    transforms.Resize(image_size),
    transforms.CenterCrop(image_size),
    transforms.ColorJitter(0.5, 0.5, 0.5),
    transforms.ToTensor(),
    transforms.Normalize(*norm)]))
```

Model Selection

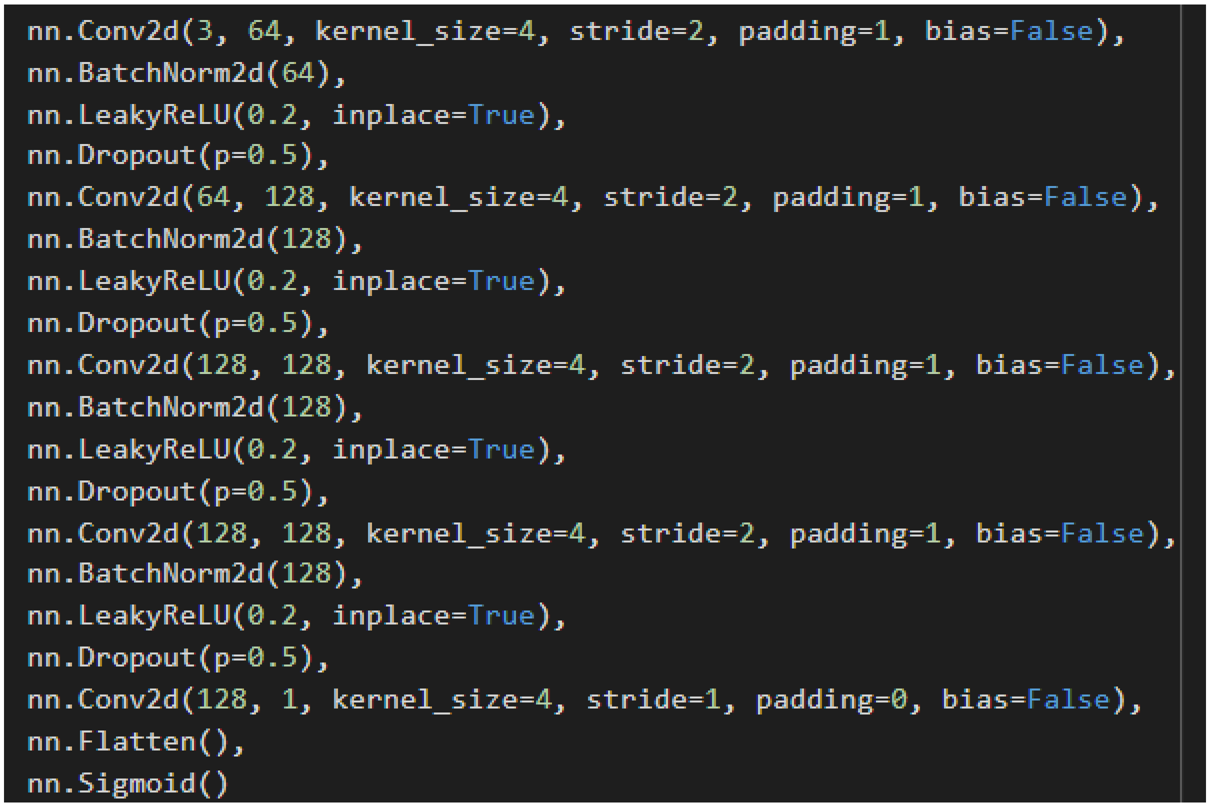
For the models, I first models mostly involving linear layers, which unsurprisingly didn't yield amazing results. I then turned to the Pokegan project for inspiration once more, and used similar models, adding dropout layers to the discriminator because I found that it was becoming too strong. For both models, I also lessened the layers used and received better results, because I was working with less detailed images, I figured there would be less features so I shouldn't be working with massive numbers of layers and features.

Generator



```
nn.ConvTranspose2d(seed_size, 128, kernel_size=4, padding=0, stride=1, bias=False),
nn.BatchNorm2d(128),
nn.ReLU(True),
nn.ConvTranspose2d(128, 128, kernel_size=4, padding=1, stride=2, bias=False),
nn.BatchNorm2d(128),
nn.ReLU(True),
nn.ConvTranspose2d(128, 128, kernel_size=4, padding=1, stride=2, bias=False),
nn.BatchNorm2d(128),
nn.ReLU(True),
nn.ConvTranspose2d(128, 64, kernel_size=4, padding=1, stride=2, bias=False),
nn.BatchNorm2d(64),
nn.ReLU(True),
nn.ConvTranspose2d(64, 3, kernel_size=4, padding=1, stride=2, bias=False),
nn.Tanh()
```

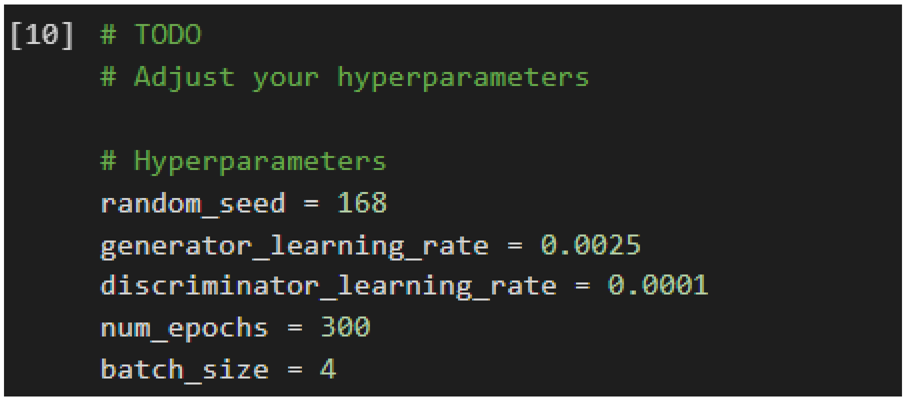
Discriminator



```
nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1, bias=False),
nn.BatchNorm2d(64),
nn.LeakyReLU(0.2, inplace=True),
nn.Dropout(p=0.5),
nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1, bias=False),
nn.BatchNorm2d(128),
nn.LeakyReLU(0.2, inplace=True),
nn.Dropout(p=0.5),
nn.Conv2d(128, 128, kernel_size=4, stride=2, padding=1, bias=False),
nn.BatchNorm2d(128),
nn.LeakyReLU(0.2, inplace=True),
nn.Dropout(p=0.5),
nn.Conv2d(128, 128, kernel_size=4, stride=2, padding=1, bias=False),
nn.BatchNorm2d(128),
nn.LeakyReLU(0.2, inplace=True),
nn.Dropout(p=0.5),
nn.Conv2d(128, 1, kernel_size=4, stride=1, padding=0, bias=False),
nn.Flatten(),
nn.Sigmoid()
```

Hyperparameters

Observing some early test runs, I found that the discriminator was too powerful consistently, and so in addition to the dropout layers, I ruled that it should have a significantly lower le arning rate. While my initial learning rates were closer to 0.005, I found turning the generator's learning rate down also helped it, as the details of the images were quite small, and so a more subtle learning approach helped it to generate better results. For the number of epochs, I ran it at 50, and wasn't proud of it, and so I turned it up to 300 to look for more definitive shapes. The batch size chosen was 4, to allow more batches per epoch due to the relatively small data pool I had.

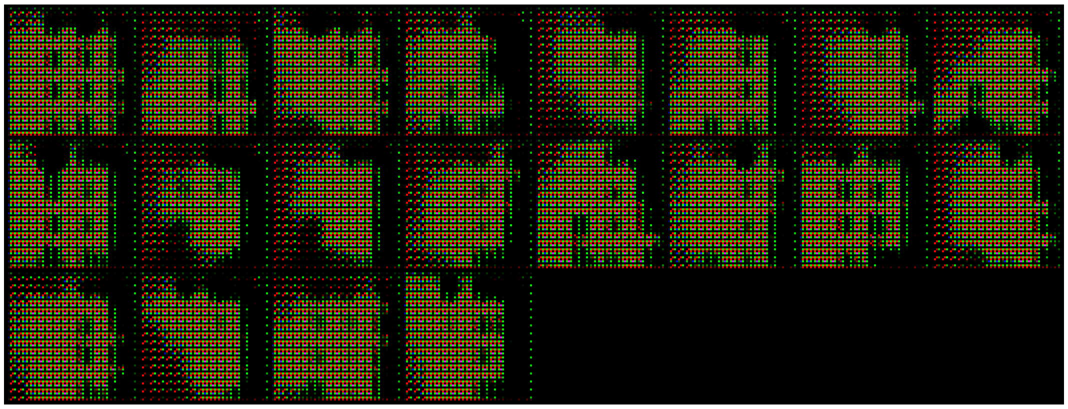


```
[10] # TODO
     # Adjust your hyperparameters

     # Hyperparameters
     random_seed = 168
     generator_learning_rate = 0.0025
     discriminator_learning_rate = 0.0001
     num_epochs = 300
     batch_size = 4
```

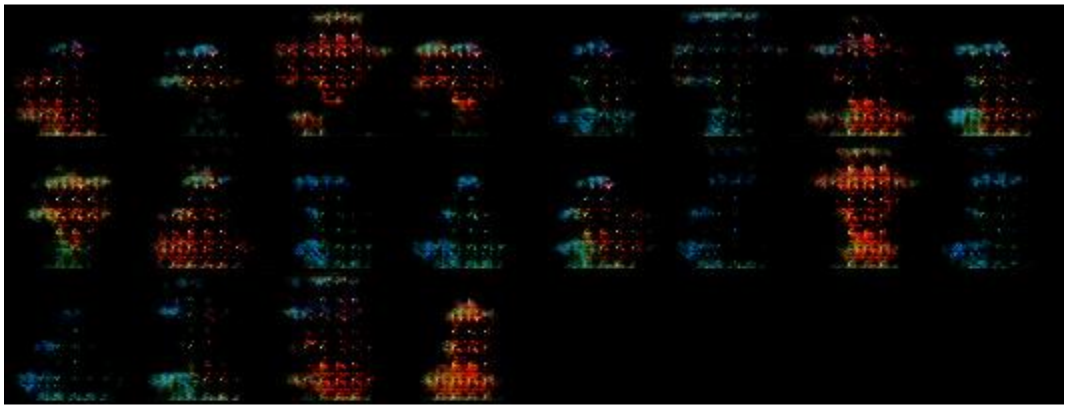
Results

My first run that yielded decent results can be seen below. I found the problem with this run was that the dataset was not shuffled, so I did not get the variety I'd been expecting, but some shapes did begin to show over time.

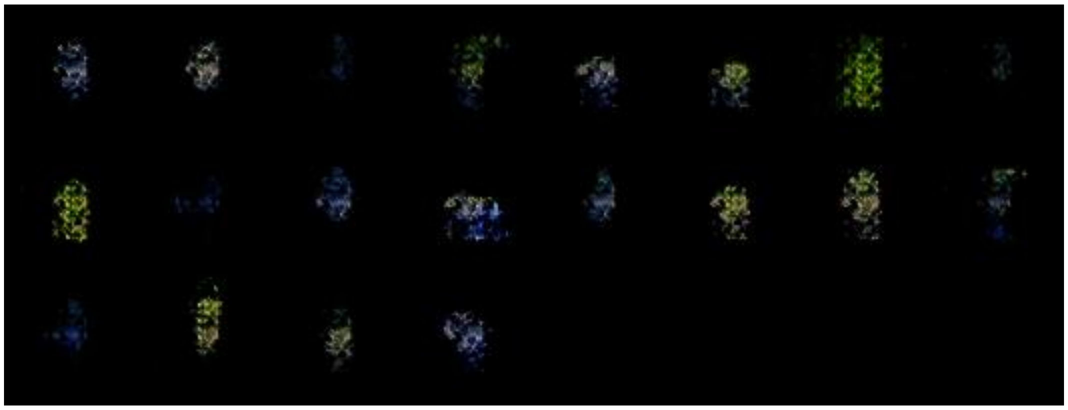
Epoch 1:



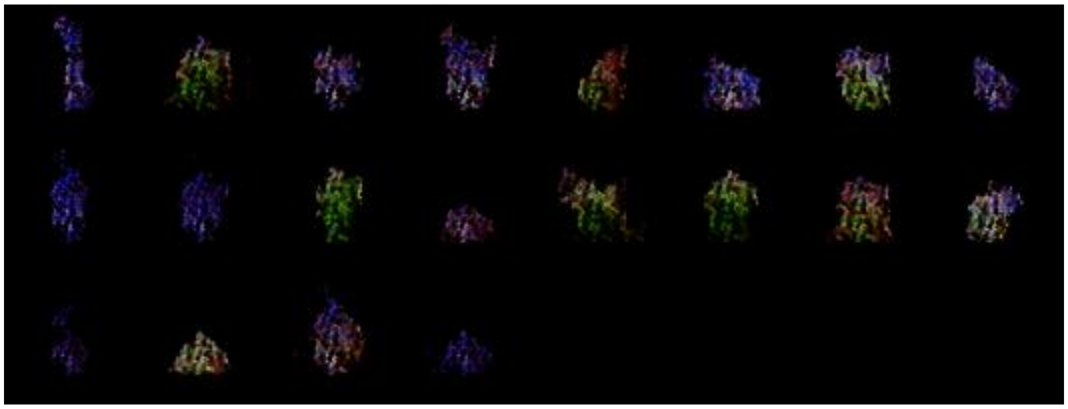
Epoch 50:



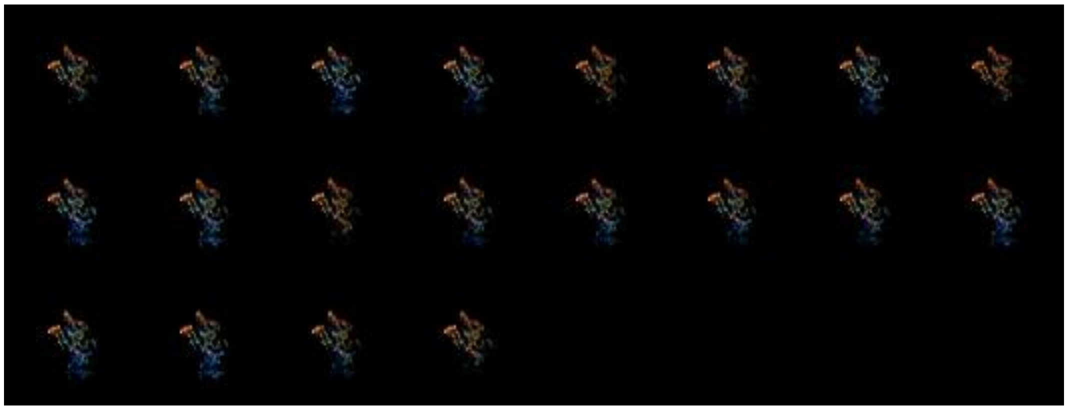
Epoch 100:



Epoch 200:

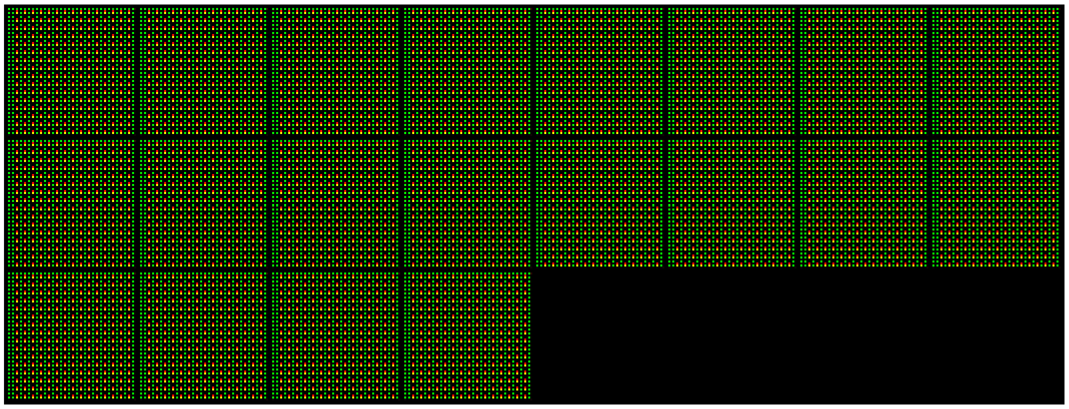


Final:

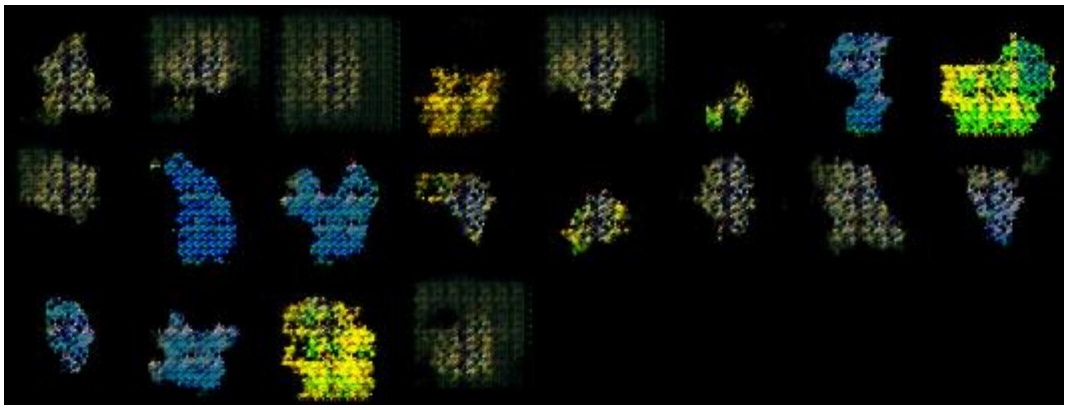


Once more, this first run lacked a shuffled testing batch, so the results in some cases , such as the last, ended up being not unique.

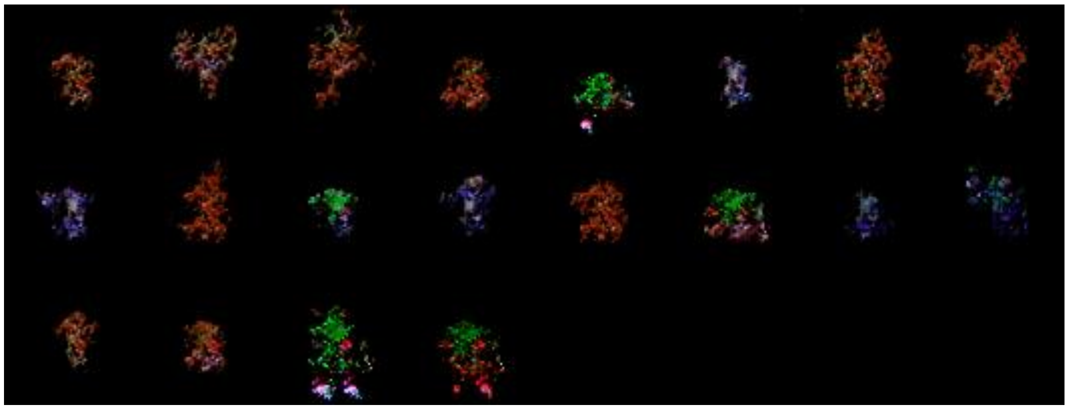
Epoch 1:



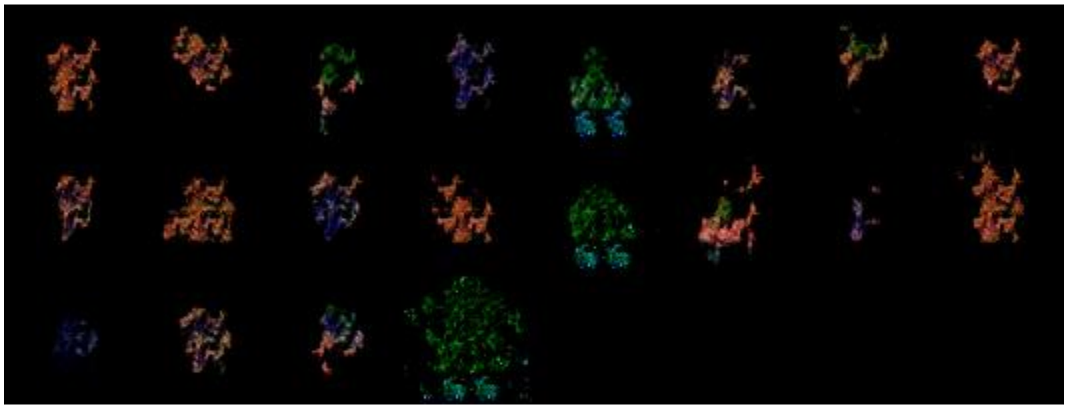
Epoch 137:



Epoch 297:

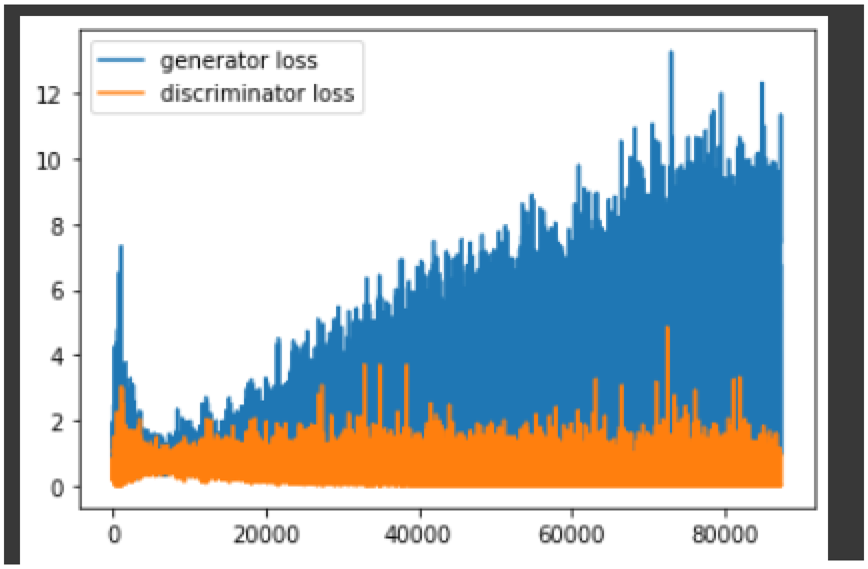


Epoch 300:



Here we can see a bit more variety.

Loss



With our loss graph, we can see that the generator increases steadily over time, and the discriminator drops, but they both still have definite spikes. What I can tell from this is that they both seem to have failings in certain areas, while growing strong in others. If we could address some of the issues, I believe it would make some more of the different types very defined.

References:

Dataset: http://f46.aaacafe.ne.jp/~aqul/aton/

Pokegan: https://www.kaggle.com/jkleiber/pokegan