Lucas Lyon
CSCI 4802 Data Science Team
Final Report

I was initially involved in Machine Learning on speech data to predict emotions. I planned to build off of my previous work where I used the TensorFlow wrapper Keras to input an audio file and output the most likely emotions. I quickly grew tired of this challenge because of my previous exposure to the subject. Instead, I switched to doing Machine Learning to generate artwork through a Generative Adversarial Network (GAN). Since my experience with Machine Learning was restricted to audio analysis, I used multiple Github repositories of previous work to start from.

After switching my focus from audio files to images and artwork, I began a proof of concept to see how a GAN would work with handwritten digits in the MNIST dataset. See example outputs in the appendix. After verifying that convincible outputs are possible by training on my desktop computer, I moved on to generating artwork. I identified multiple datasets that could be useful, including WikiArt and the Behance Artistic Media Dataset (BAM). WikiArt proved to be more useful due to its organization. I was unable to generate convincing outputs with my own home-made Neural Network, but was able to replicate results that other researchers have reached.

Generative Adversarial Networks are Neural Networks that have two players: one that generates data, and one that determines if the generated data is real or fake. The generator takes in a sample of random visual noise with a normal distribution, and transforms it thousands to millions of times. The output of those transformations is fed into the discriminator, whose job is to determine if the data it receives is a real piece of sample training data, or is generated by the generator. The goal of the Neural Network is to be able to produce convincing generalized outputs that could be considered real data.

With this in mind, I crafted twelve Neural Networks in Keras based on work by Felix Mohr, Robbie Barrat, and Soumith Chintala. To ensure my personal hardware could handle training a Neural Network on images between 64x64px and 512x512px, I started with Mohr's Ipython Notebook tutorial *Teaching a Deep Convolutional Generative Adversarial Network (DCGAN) to draw MNIST characters.* The sample code provided was thoroughly explained, and provided an excellent basis to adapt to the WikiArt dataset. With some adaptations, my code from this exercise is titled *gan128MNIST.py*.

After working with the MNIST dataset, I quickly added in the WikiArt dataset to begin training on artwork. After 9300 training epochs, the generated images were not looking like art, and instead looking like random noise. Part of the difficulty in generating artwork versus MNIST characters was the number of color channels, where the artwork used RGB and the MNIST characters were in grayscale. This made training more difficult, and allowed the discriminator to get very good while the generator was left behind. I learned through research online and via the information Robbie Barrat provided that a balance in power between the discriminator and generator was important, otherwise the outputs would not be high quality. My code for this exercise is titled *gan128Paintings.py.*

To counteract the increased difficulty of training on three color channels instead of one, I adjusted the learning rates of the generator and discriminator. The generator now learned at twice the rate of the discriminator. Additionally, I adjusted the input of the model from 128x128px images to 256x256px images, and decreased the generator's *momentum*, which is the level of decay for stochastic gradient descent. I took these steps with the hope that outputs would look less "rigid", where similar splotches and patterns repeat throughout the outputs.

Although the outputs of the generator changed, they still fell into a pattern of similar results no matter how long I trained for. My code for this exercise is titled *gan256Impres.py.*

After adjusting the learning rates and momentum and not seeing positive results, I attempted to increase the level of dropout in the generator and discriminator. As dropout increases, the amount of nodes in the hidden layers that are randomly ignored during training increases. Since the output images I was getting from *gan256Impres.py* appeared to be overfitted, I hoped that increasing dropout would help give higher-quality looking outputs. For the sake of exploring the WikiArt dataset, I chose to train this model on artwork exclusively from the artist Chuck Close, an artist famous for photorealism. Although the first few hundred training epochs produced promising results, the generator fell into the same trap of high-contrast randomness. My code for this exercise is titled *gan256Chuck.py.*

I once again doubled the input size, this time from 256x256px to 512x512px. Each time I changed the input size I had to also crop the images into the correct shape, so I wrote a script titled *imageResizer.py* to do this. Also, I reverted the level of dropout in the generator and discriminator to a more reasonable 20%, down from 60% earlier. The last tweak I made on this version was the threshold for when to train the generator and discriminator. At this point, the algorithm was training the generator and discriminator only when their performance was poor compared to their counterpart. I reduced the threshold from 1.5 times to 1.35 times, so when the performance of the discriminator is 1.35 times better than the generator, it stops being trained until the generator can catch up to the performance of the discriminator. Outputs from this revision again looked promising, but the extended training times from increasing the input to 512x512px made this model untrainable on my personal computer. My code from this exercise is titled *gan512.py.*

After seeing poor results coding in vanila TensorFlow, I decided to convert the code I was using to Keras, with the help of Erik Linder-Norén's *Collection of Keras implementations of Generative Adversarial Networks.* The network structure I used was a blend of Linder-Norén's Keras DCGAN and Mohr's MNIST DCGAN, with modifications to dropout, momentum, normalization methods, and the saving mechanism to output images and model training checkpoints. Although my understanding of the Neural Network in this revision was greater because of the use of Keras syntax instead of TensorFlow code, the results were still poor: similar images of randomly transformed noise. The output images appeared to be consistent, so the model was learning some pattern from the training data, but not the patterns I was hoping it would learn. My code from this exercise is titled *kerasGan.py*.

My Neural Networks were not meeting my expectations, so I attempted to simplify my code to see what was causing the consistent but inaccurate results. In my previous work on audio file emotion analysis, I found that the more complicated my model became, the more likely I was to make bad predictions; this wasn't a lesson against complexity, but a warning that adding extra layers can introduce or even amplify prediction errors. The reduction in complexity improved my level of comfort with the Neural Network, but still did not produce high quality outputs. My code from this exercise is titled *kerasGANv2.py.*

To speed up the prototyping process, I reduced the dimensions of the input from 50x50px images to 40x40px images, and reduced dropout to 0%. Also, the source images I was using to train were of dogs instead of artwork. I thought that the increased uniformity that images of dogs provided would help produce higher quality output images, since the difference between pictures of dogs was relatively low, and the differences between pictures of artwork could be very high. My assumptions about training time proved true, and I was able to go through 49,000 training epochs. Unfortunately, the generated images of dogs were low quality random noise. My code from this exercise is titled *kerasGANv3.py.*

Simplifying the structure of my Neural Network did not prove to be helpful in identifying what was causing the poor output results. In an attempt to see if the model was learning any pattern or was just producing random noise, I increased the resolution of the input images from 40x40px to 250x250px images. Although the training times increased significantly, I was not able to visually determine if the outputs were trending towards high quality, even with additional pixels that could help in distinguishing contours from randomness. My code from this exercise is titled *kerasGANv4.py.*

I was now on my ninth model iteration, and had yet to produce any noteworthy outputs. I found Robbie Barrat's refinement of work by Soumith Chintala implementing the theoretical code from the paper titled *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.* Barrat had managed to adapt Chintala's code to produce beautiful artwork that emulated many categories from the dataset I had been using, WikiArt. I downloaded Barrat's Github repository and was able to emulate the results he got.

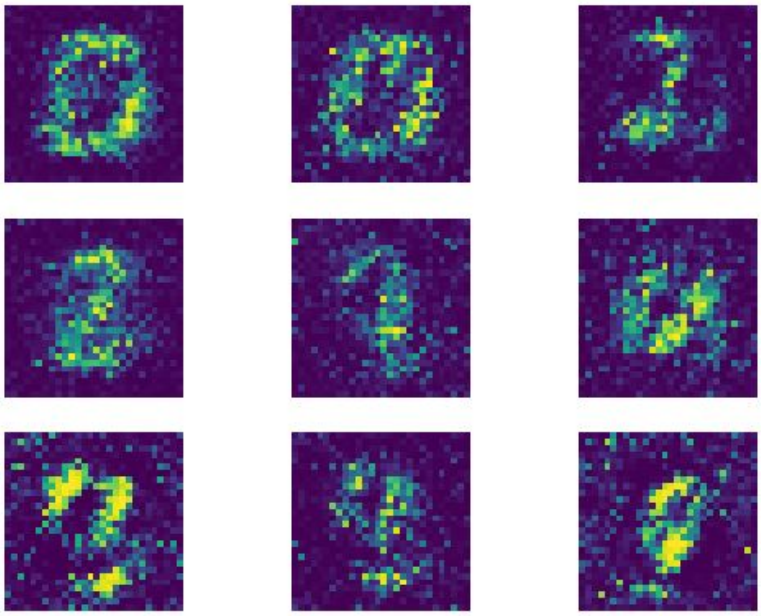
Chintala and Barrat both used Torch and Lua to implement their Deep Convolutional Generative Adversarial Networks, but I was most comfortable with Keras and TensorFlow. To emulate the results that Barrat got, I had to do a crash-course on Torch and Lua. Learning a new programming language was time-consuming, and I am far from an expert in Torch, but this was one of the biggest learning moments for me during the semester. After watching a sufficient amount of YouTube videos on Torch and DCGAN's, I was able to understand the essentials of what Barrat had accomplished. To test my understanding of Barrat's implementation, I attempted to convert the Torch Neural Network into Keras code. The functions in Torch had respective equivalents in Keras, and it just took some Googling to figure out how to convert the model structure of Barrat's code into Keras code. My code from this exercise is titled *kerasGANv5.py*.

At this point, I had completed one of my midterm report goals, which was to convert Barrat's code into Keras. My next stated goal was to increase the resolution of the input from 128x128px to 256x256px, so that outputs look prettier and more presentable. I hit a computational limit at this point. My personal computer has 8GB RAM, and during training of *kerasGANv5.py,* my RAM usage was nearly 100%. The code would not run if my computer was doing anything else than the Keras code, and increasing the input image dimensions made the code completely unusable. I am in the process of upgrading my computer with more RAM, and will attempt further training once I have more resources to devote to training.

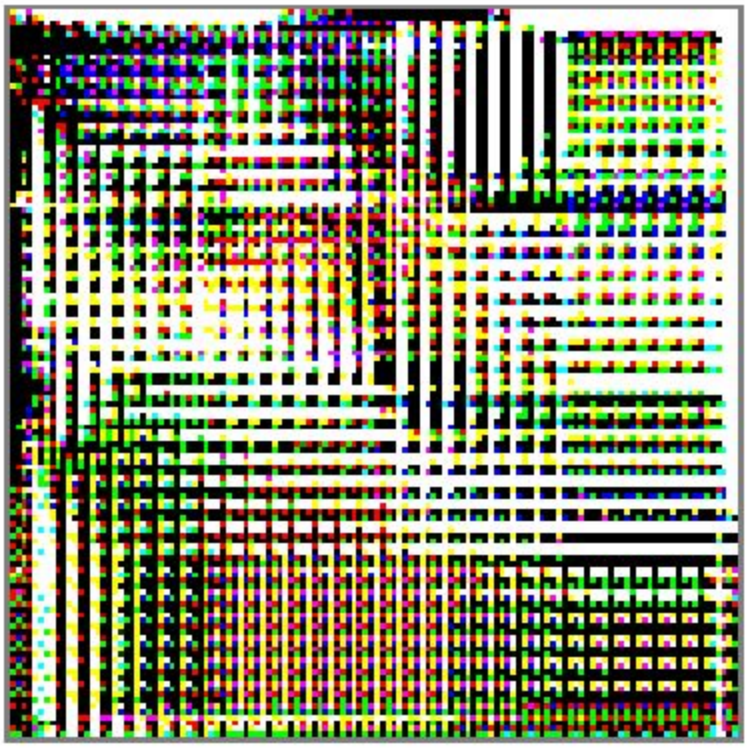
From my midterm report, I had four goals:

1. Emulate what Barrat has already accomplished
2. Attempt to convert Barrat's code to TensorFlow/Keras (I'm more familiar with these than Torch and Lua, which Barrat used)
3. Adapt Barrat's generator and discriminator algorithms to work on 256px by 256px images (Barrat's models can only handle up to 128px by 128px)
4. Make the model exportable and importable so that re-training isn't necessary for each type of artwork desired

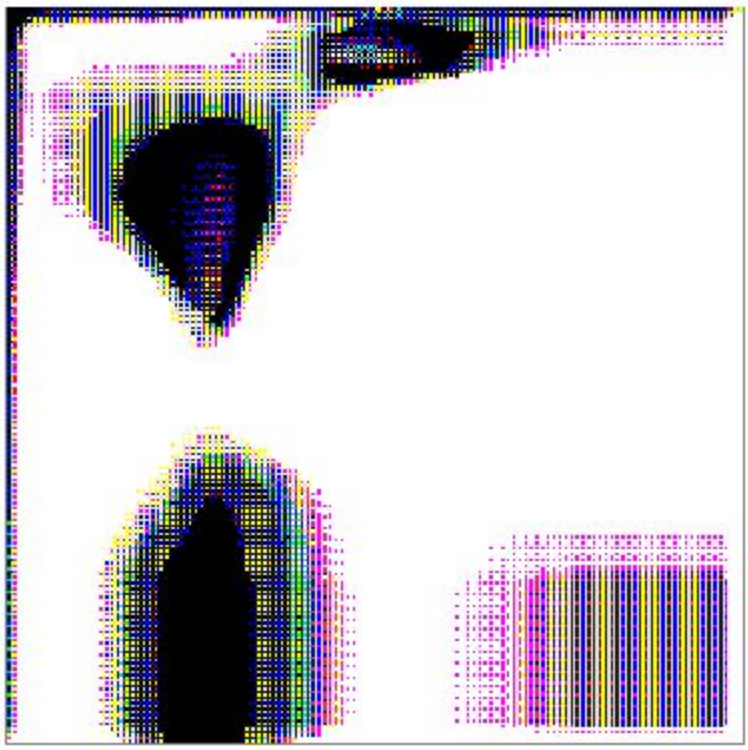
I was able to accomplish goals #1 and #2. I could not accomplish goal #3 due to limited computational resources. Goal #4 was already implemented by Barrat, and therefore unnecessary. I did implement a custom saving/checkpoint mechanism in my first few Keras implementations. Attached below are example outputs from each iteration of my Neural Network.



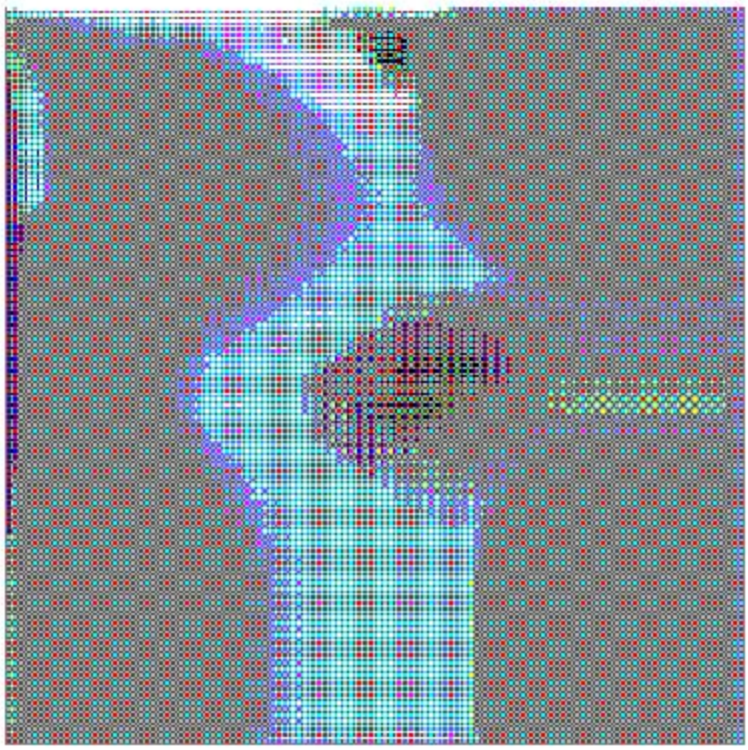
Output from training epoch 2900, 3x3 grid, gan128MNIST.py



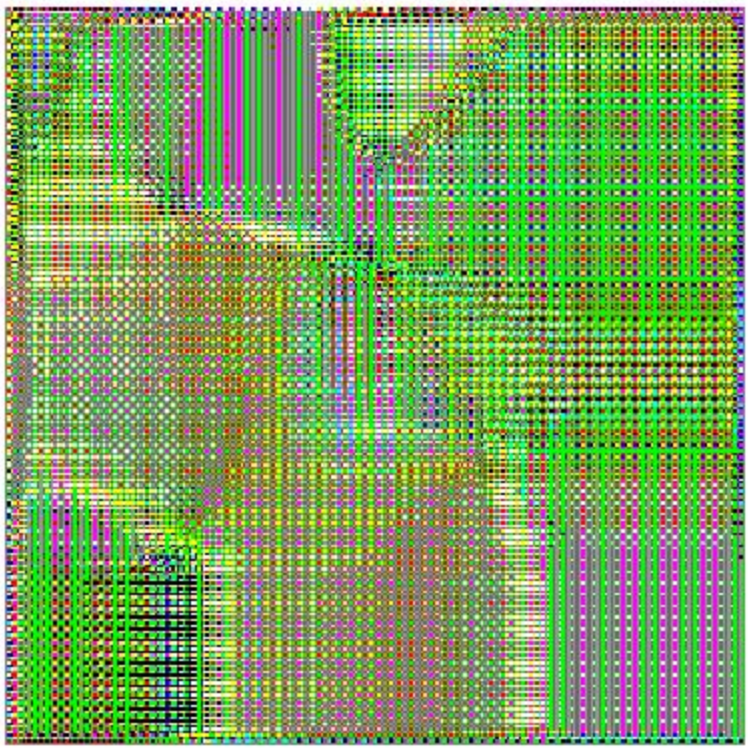
Output from training epoch 9300, 1x1 grid, gan128Paintings.py



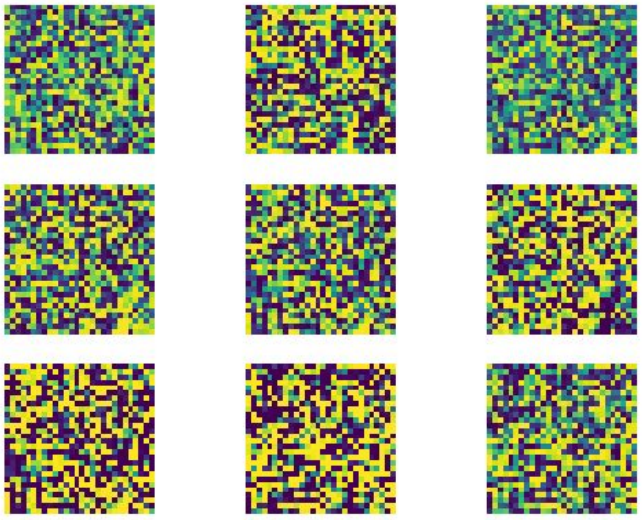
Output from training epoch 14740, 1x1 grid, gan256Impres.py



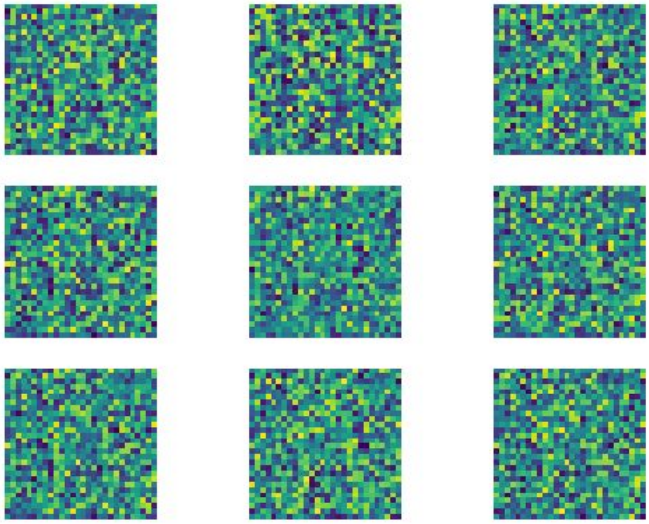
Output from training epoch 1200, 1x1 grid, ganChuck256.py



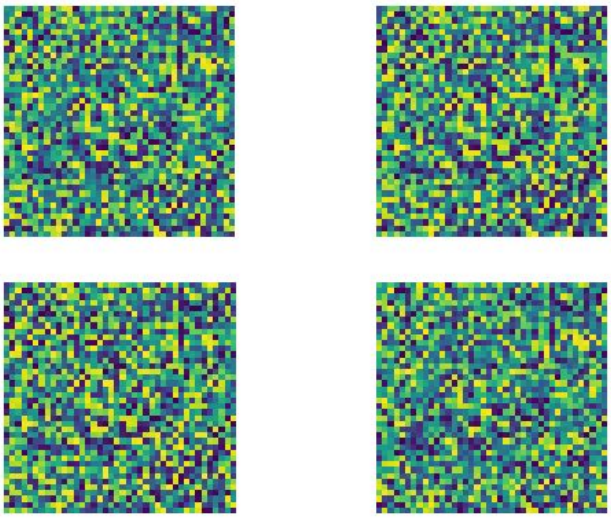
Output from training epoch 510, 1x1 grid, gan512.py



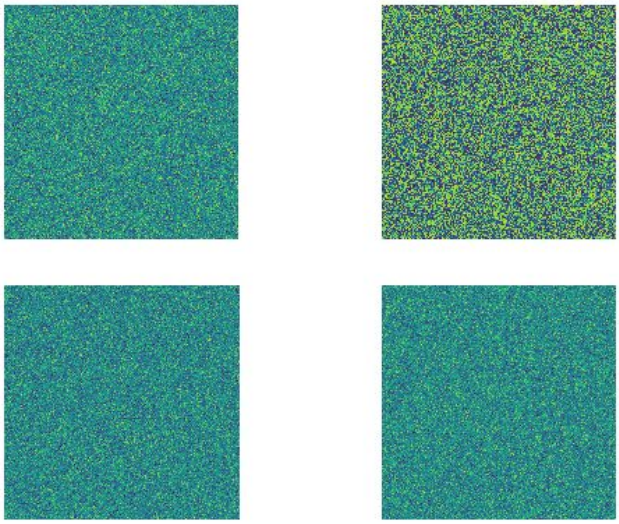
Output from training epoch 4900, 3x3 grid, kerasGAN.py



Output from training epoch 4900, 3x3 grid, kerasGANv2.py



Output from training epoch 30500, 2x2 grid, kerasGANv3.py



Output from training epoch 2800, 2x2 grid, kerasGANv4.py