DeepGlitch
By Josh Bevan

Abstract:
DeepGlitch was a project of discovery and self-education. In the past I had worked with machine learning algorithms as though they were big black boxes. To ‘do machine learning’ had 'really' meant exploring AWS virtual machines, the Docker container workflow, and building pipelines that would combine the almost magical abilities of these stochastic algorithms with each other such as in my earlier projects DeepCollage (2018) and SilentNET (2019).

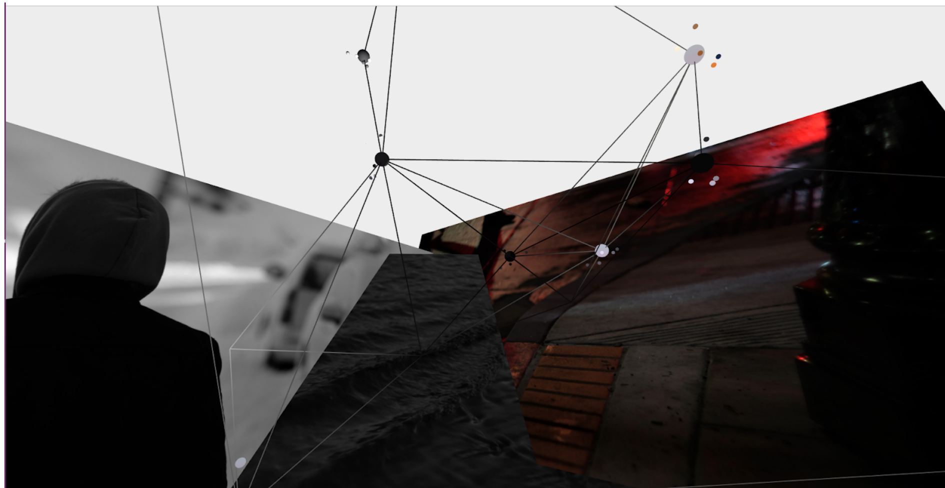
This project was my opportunity to a) become adept at using machine learning techniques so I could control the outcomes with greater intentionality and B) to create effects for my film Echo, a dark cyberdelic romance film I have been working on for over a year now.

The final concrete objective to the project was to recreate what others have already achieved: style transfer for audio. That specific journey can be found in this report under the header: *TraVelGAN hacking*.

Story:
The initial proposal for the course was to create a real time openGL post-processing pipeline that used a parallel instance of the pix2pix style transfer network for each image in the scene. The intended outcome would be explorable 3D scenes were the grass could be styled after Monet, the characters could be styled after Francis Bacon, the sky after Van Gogh, and so on.

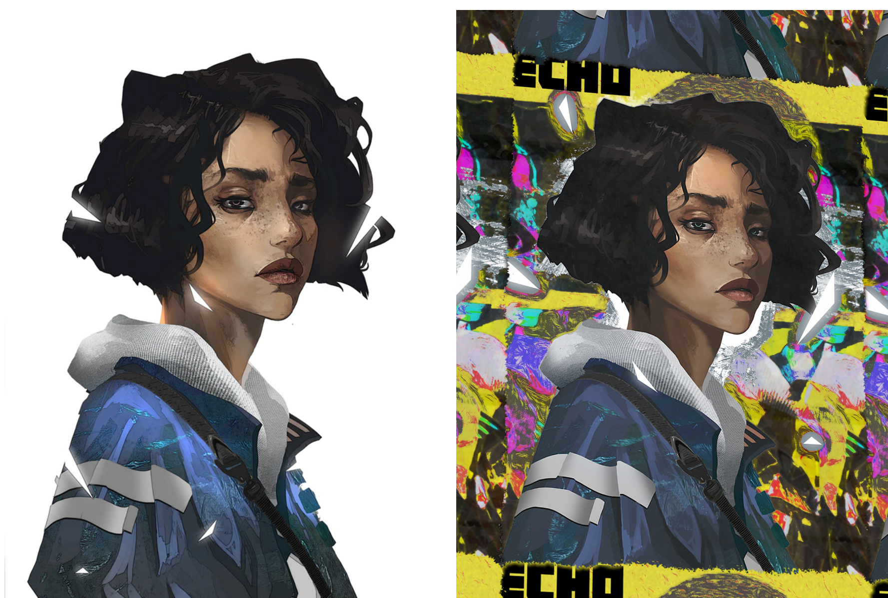
I began by learning how to control OpenGL with C++. I had some experience with the similar WebGL framework working on my web project WebDead (2017), but C++ and its quirks represented a particular challenge.

I created a basic OpenGL scene using some dynamic elements and textures sourced from photos I had taken:



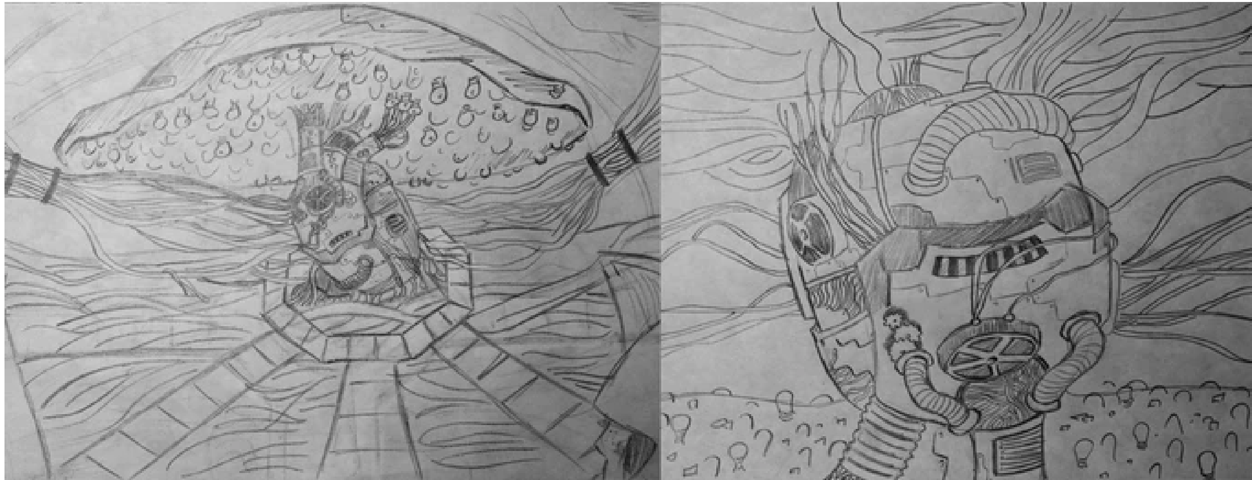
While I was working on this, I was working on building a campaign to raise funds for my film project 'Echo'. I commissioned concept art that involved detailed discussions of character and story to achieve, resulting in:

And a second piece (on the left) along with a poster I made using various effects for the background including running the image into Audacity as audio and doing the 'echo' effect on it:



I also created merchandise photos:

Did my own concept art drawings:



And created a rough animated storyboard to explain the story:

TravelGAN Hacking:
After I had finished my excursion into the ‘Echo’ campaign, I decided to switch my direction in the class, moving towards audio style transfer, again for the reasons stated in the abstract.

My technique of self-education was to read through the code of a pre-existing machine learning network, specifically the TraVelGAN style transfer network, and make sure that I understood what each line was doing.

I explored down every rabbit hole of misunderstanding - I read papers about convolutional neural networks and GANs, learned about loss/cost functions and optimizers and stochastic mathematics, and about strategies for overcoming the famous overfitting and underfitting problems in machine learning including randomcrops, batch norms, and layer skipping.

A huge technical hurdle was learning how Tensorflow and Keras implemented these techniques.

Knowledge Gained Summary - Tensorflow and Keras
Tensorflow works declaratively - that is, it works by specifying a 'graph' of data (tensors) and operations that are then run alltogether by 'sessions'. The purpose of such a workflow is that under the hood Tensorflow optimizes the graph to take the most advantage of available resources, instead of allowing the silly dumb-dumb-head data scientists to build their own algorithms for processing the operations and tensors.

Keras was just a collection of useful common machine learning operations such as 2D convolutions and batch norms.

Knowledge Gained Summary - Style Transfer From the Bottom Up
The Convolutional Neural Networks (CNNs) used in TraVelGAN were a flavor called U_net, originally designed for the task of object segmentaiton in biomedical imaging.

A U_net works by 'encoding' the features of a source domain of images into a 'latent feature space' through a 'down' set of convolutions, and then 'decoding' those features into 'new' images through an 'up' set of convolutions.

As the U_net encodes 'down' or decodes 'up', it recursively convolves the source domain images with kernels that have parameterized weights.

These kernel weights are what is being 'trained' or 'learned', and represent what is called the 'model' in machine learning.

In a typical GAN, two U_nets are 'trained' with opposing objectives.

The Generator U_net is attempting to encode images from a source domain into a vector within the feature space that can be decoded into a new image in some target domain.

And the Discriminator U_net is attempting to encode the images generated by the Generator into a vector within the feature space that can represent the probability that the generated image is in the target domain.

In a style transfer GAN, the target domain will be different than the source domain. Ordinarily this would mean the generator would generate new examples of the target domain and ignore anything about the source domain.

Therefore additional constraints are added to incentivize the Generator U_net to retain the 'content' from the source domain, but still create images that are realistically from the target domain.

TraVelGAN incentivizes the Generator to do this by adding a third CNN called a Siamese network. A Siamese network encodes the source image and the generated image each into a 1000-dimensional vector in a 'content space'.

They adjust the incentive for the Generator to also minimize the difference between the content space vector for the generated image and the content space vector for the source image. Therefore, the generated images must still plausibly exist in the target domain (in a different style) AND maintain the same 'content' of the source domain.

What I Changed With The TraVelGAN

The addition that I put myself to hacking into the TraVelGAN was an additional constraint on the generator that would incentivize it to be stable across images concatenated horizontally - that is, spectrograms of arbitrary width stitched together from spectrograms of a specific width.

Within the Tensorflow graph I was to split the images in half, then generate two images that are then concatenated together before put through the discriminator.

Before entering the graph, I would have to convert each audio clip into spectrograms of standard dimensions, mapped in a way I could unmap from, and after the graph I would have to concatenate the generated images into their full length versions and convert backwards from Mel Spectrograms to audio files.

In order to do that I had to put my knowledge to the test - adjust hyperparameters, disable certain quirks about the original TraVelGAN implementation, play ball with the tensor dimensions they had set, and...

Most importantly, I had to get it working on my laptop. In the later stages, I had already successfully written the changes to the graph, and the preprocessing and postprocessing steps were integrated into the code, but then I started getting the OOM errors (out of memory), 'memory errors', and 'unable to allocate array' errors.

I had to re-roll my list comprehensions into loops, do some VERY 'I dont care just let me do it' error handling, learn about file-like objects, streams, and buffers, and compromise on some hyperparameters like batch size. The current network works in batches of 4 images at a time, which is tiny for these sorts of things.

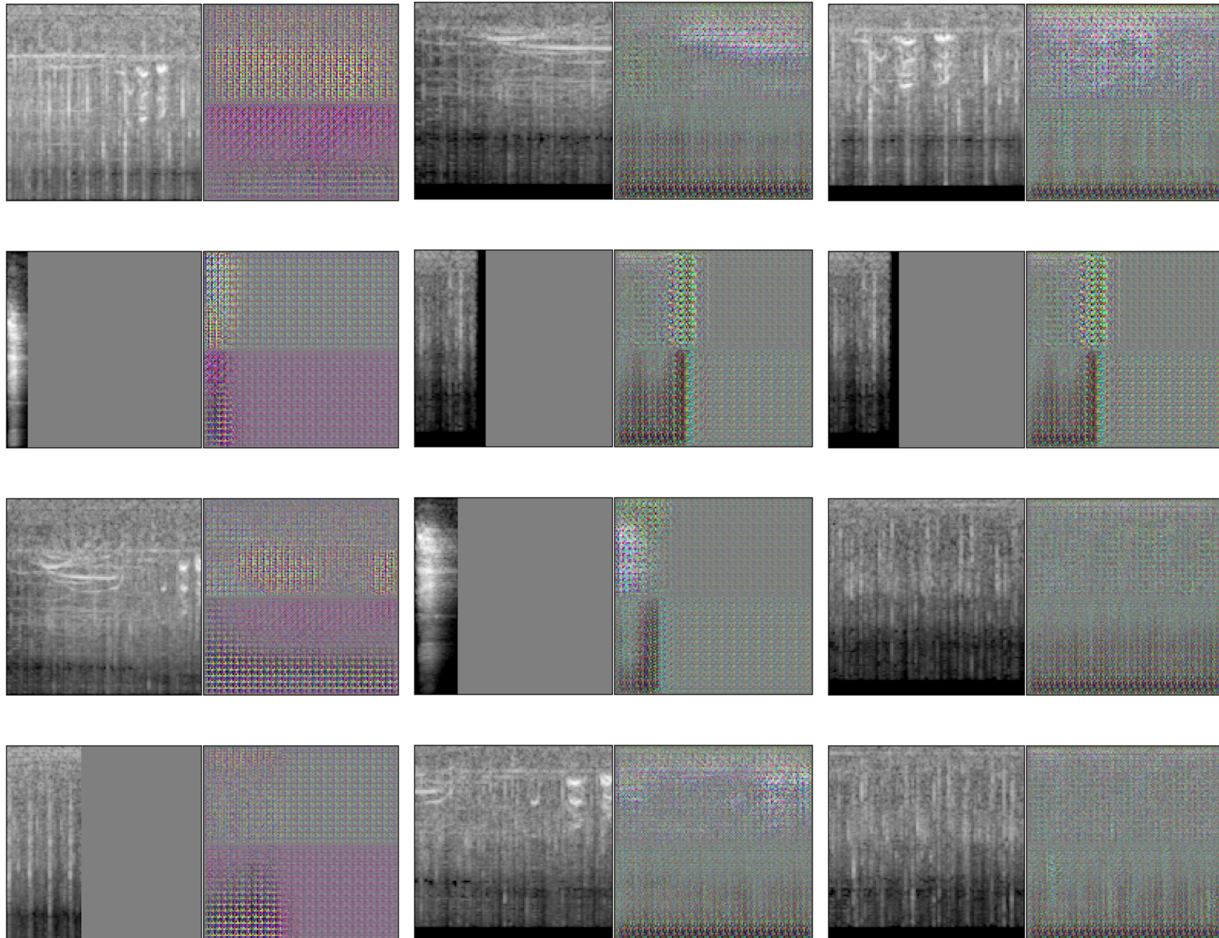
Here's a code snippet with some explanations of its function in the '#' comments:

```
inputDir = args.datadirb1
D = 128

b1 = []
for filename in os.listdir(inputDir):
    inputPath = inputDir + filename
    outputPath = outputDir + filename.split('.')[0] + '.jpg' # change file ending to '.jpg'
    b1.append(inputPath)

b1 = [fn for fn in b1 if any(['mp3' in fn.lower(), 'wav' in fn.lower(), 'aif' in fn.lower(), 'flac' in fn.lower()])]

out = []
for path in b1[:20]:
    try: # this try-except block just skips audio files that won't load properly
        with open(path, 'rb') as audioFile:
            # read the file
            tmp = io.BytesIO(audioFile.read())
            rawAud, sr = sf.read(tmp)
            # transpose and resample it
            rawAud = rawAud.T
            rawAud = librosa.resample(rawAud, sr, 44100)
            # average the stereo channels into a single mono channel
            rawAud = (rawAud[0,:] + rawAud[1,:]) / 2
            # convert the loaded audio into a mel spectrogram - map to -1:1 range - cast to three channels
each with the same data ( using np.repeat() and np.newaxis)
            melSpec =
np.repeat((librosa.power_to_db(librosa.feature.melspectrogram(y=rawAud,sr=sr),ref=np.max)/40+1)[:,:,np
.newaxis],3,axis=2)
            # calculate how many images of width 128 we can squeeze out of melSpec
            nimgs = int(np.ceil(melSpec.shape[1]/D))
            # grab those images from melSpec and save them to out. In case we're looking at an image that
doesn't fit into 128 width, then pad the rest of the values before saving it
            for j in range(nimgs):
                res = melSpec[:,j*D:j*D+D,:]
                if res.shape[1] < D:
                    res = np.pad(res, ((0,0),(0,D-res.shape[1]),(0,0)), constant_values=(0,0))
                out.append(res)
    except Exception:
        pass
```

Experiments

The final result produced the following images when trained on dog barks -> car honks. Images with cutoff portions are the ends of certain files being padded:

At iteration 10, 500, 1000 (source images on left, generated images on right):



You can see that as they progress in iteration count, the information in the outputs starts to match the details in their sources more closely. It also seems as though the lower halves of the images show a distinct boundary in some of the generated images.

As I made the code translate the spectrograms back to audio, I have those as well, though I do not embed them here. The first thing I recognized in the audio is that the source domain audio is distorted due to the conversion to and from a spectrogram. The second thing is that the target audio has a slightly more tinny sound to it - lower frequency information has been converted to higher frequency sound. This squares with the domain shift from dog barks to car horns.

Future Work:

More extravagant examples showing the kind of magic this network can do will be soon to come.

Working with AWS cloud computing will allow me to escape the resource boundaries of my laptop, and reduce the amount of time spent to reach the high iteration count (50,000 and above) required to reach that kind of magic.