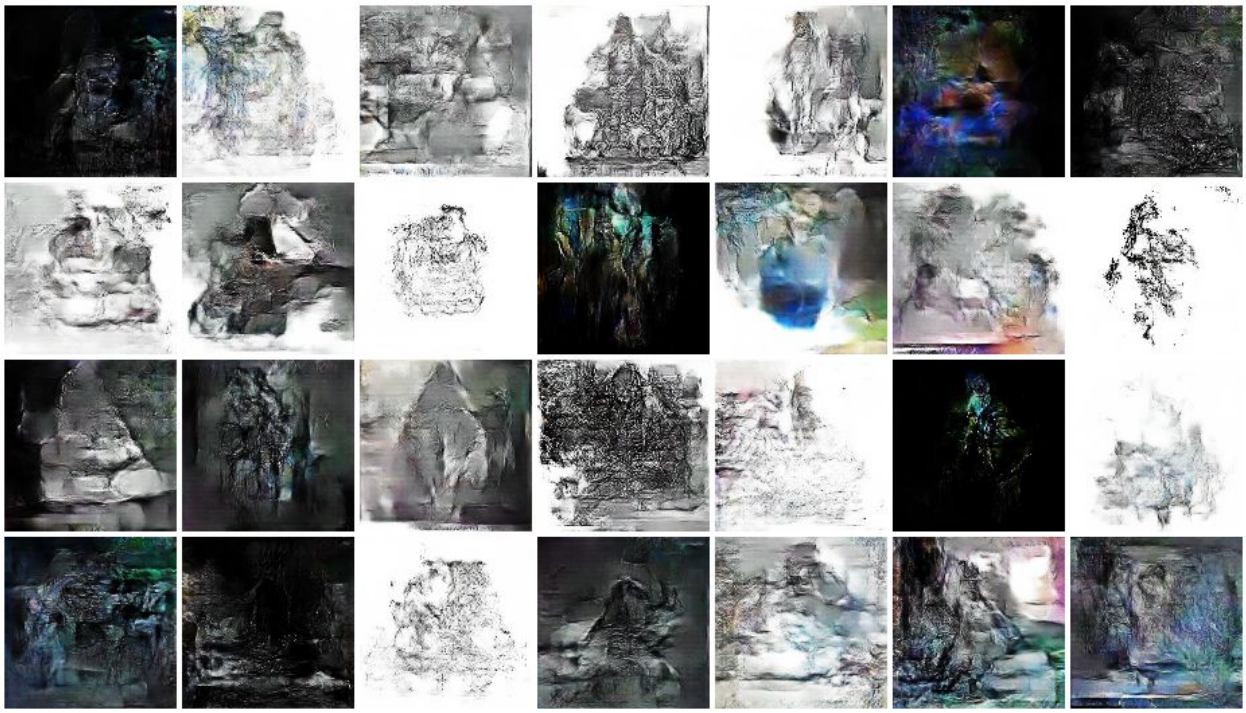
Chan Hiu Yan

IGNORANCE OF IDOLATROUS KNOWLEDGE, 2020



Generative Adversarial Network and feature extractor with TensorFlow.keras

"Ignorance of Idolatrous Knowledge" is a machine learning art piece that attempts to reveal the ignorance of artificial intelligence. Often feared by the public as maleficent robots with individual discretion, AI is actually very uncreative in the sense that it cannot go beyond the boundaries of its knowledge. In this piece, a GAN is trained to generate images of idols, whereupon when fed with images of humans, it knows only to idolize them. This act of idolatry parallels the ignorance of humans when mankind refuses to acknowledge his Creator God. Just as a machine that recognizes its programmer will be deemed the most intelligent AI ever lived, true wisdom comes from knowing God.

Documentation

Documentation video link: https://vimeo.com/422170707

Concept

Artificial intelligence? Or is it artificial ignorance?

In popular culture, the term “AI” may invoke images of humanoid robots falling in love, or autonomous computers conspiring to take over the world. However, are machines actually capable of acting out of its own will? Can computers think outside of what it has been told? Sure, many works have illustrated its generative abilities - producing images of faces that never existed, hallucinations of eyes in a photo - yes, these are generative, but not quite creative. Before being able to generate faces, it had been trained on a dataset of faces, and not apples for instance. Had it been able to generate faces from apples, that I have to admit is creative.

So is AI intelligent? In regards to its ability to think for itself, I would say no. In a more demeaning term, it is merely a function. It is only capable of doing what it is told with what it is given. It does not understand why it does what it does or what the data means. For example, in a neural network trained to classify images of digits, one can equally input images of oranges with a confident prediction of “0.” The machine is ignorant of what it knows and what it doesn’t know. But if the machine learns of its own ignorance, it is a large step towards intelligence. As stated in Plato’s Apology, “He among you is the wisest who, like Socrates, knows that his wisdom is really worth nothing at all.”

In light of this, I pursued the concept of training a network to recognize its own ignorance. Others have made similar attempts to differentiate when a given data is outside of its knowledge. Abhijit Bendale and Terrance E. Boult developed the OpenMax method[1] that made use of patterns in the activation vector. Dhamija et al.[2] actually trained the network with what they expect to classify as “unknown” ironically. Yarin Gal and Zoubin Ghahramani[3] applied the Bayesian theories to improve the threshold of probability under which data would be classified as uncertain. As an amateur in the field, I can only conclude that researchers have yet to yield an effective method to comprehensively teach a machine what it doesn’t know. And if anyone was to do that, it is not me within the scope of this project.

Artificial Stupidity

Instead, in my artwork I will show the very incapability of a machine to know what it knows. For example, a network trained to classify the age of an image of a person, when given an image of a banana, may confidently return a certain age. In fact, some bananas may compete better as a “45 year-old female” than other bananas. Such a spectrum of bananas may be interesting to

---

[1] A. Bendale and T. E. Boult, Towards Open Set Deep Networks (2016), The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 1563–1572

[2] A. R. Dhamija et al., Reducing Network Agnostophobia, (2018) arXiv preprint

[3] Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning: Yarin Gal, Zoubin Ghahramani

visualize. That said, bananas may not be the most meaningful object to explore. I wanted the object to reflect on the meaning of the artwork as a whole.

Idolatry

I remember seeing in the Barbican AI exhibition last summer, the history section showed figures of idols as the earliest form of artificial intelligence. In essence, man attributed intelligence to carved blocks of wood, where the idol is animated in man's perception. Similarly, AI can be described as intelligent merely in man's perception of its apparent understanding. As a Christian, I was fascinated by this nomination of idols as the pioneer of AI, having never perceived it in this sense. In a general definition, idolatry is anything that man upholds in the place of the only true God. In other words, God who created the world including mankind deserves all worship and glory, yet man in his own narrow mind attributed that glory to what is not God - that is, by carving his own idols and worshipping them as gods.
In parallel, man created AI, which in its narrow mind, does not recognize man. If any AI were to recognize the man who programmed him, it would have been deemed the most intelligent AI, outsmarting other AIs that are able to solve computationally heavy problems. Like so, if any man were to acknowledge the God who created him, he is wise, as it is written in the Bible:

> *This is what the LORD says:*
> *"Let not the wise man boast of his wisdom*
> *Or the strong man boast of his strength*
> *Or the rich man boast of his riches,*
> *But let him who boasts boast about this:*
> *That he understands and knows me,"*
>
> *Jeremiah 9:23-24a*

In this artwork, I would like to illustrate the stupidity of AI in the sense that:

1. It doesn't recognize the one who created it
2. It doesn't know what it doesn't know

The neural network is trained to classify images of idols. Once trained, the network is fed images of people, which it will then try to classify as a category of idol. In a way, the network commits idolatry by attributing the idol as its creator. It is trapped in its thinking because all it has ever seen are images of idols. Here, a parallel is drawn that mankind is trapped in his sin. This however, cannot be translated directly to conclude that man has no choice, because we know man is superior to machines in ways incomparable, including the fact that man has a will to choose. And God is superior to man in ways incomparable, including his power to save man from sin. As it is written in the Bible:

> *"As the heavens are higher than the earth,*
> *So are my ways higher than your ways*

And my thoughts than your thoughts."

Isaiah 55:8-9

As a reflective artwork, I would like to assert about man:

1. Man is only wise when he acknowledges the one who created him
2. God's ways are higher than man's ways

Process

To show that the machine is only capable of knowing what it knows, I planned to train it with some dataset A, and test it with another dataset B it had not seen before. To start off, a simple way to show this is with a CNN classifier. Whatever item from dataset B is fed into the model, it will recognize the item as one of the classes from dataset A.

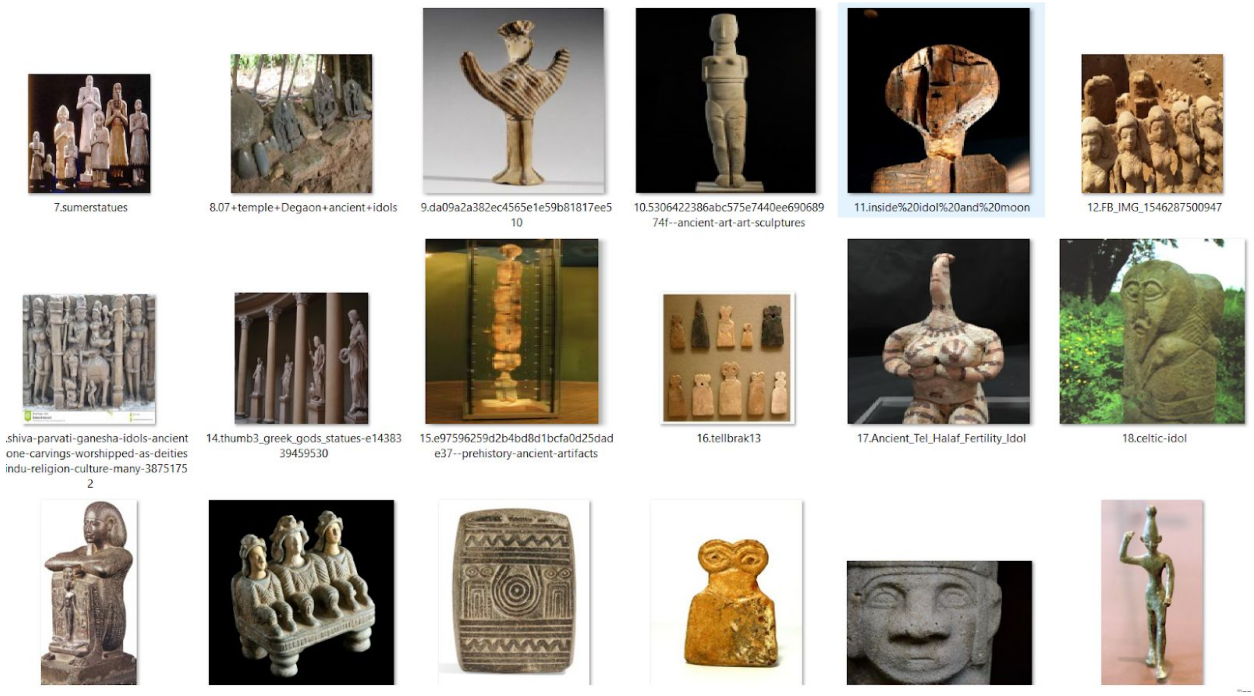
From there, to make things more interesting, I trained a deep convolutional GAN with dataset A. Then I tested its response inputs of dataset B. This exposes how the trained GAN makes sense of something that it had never seen before.

Gathering data

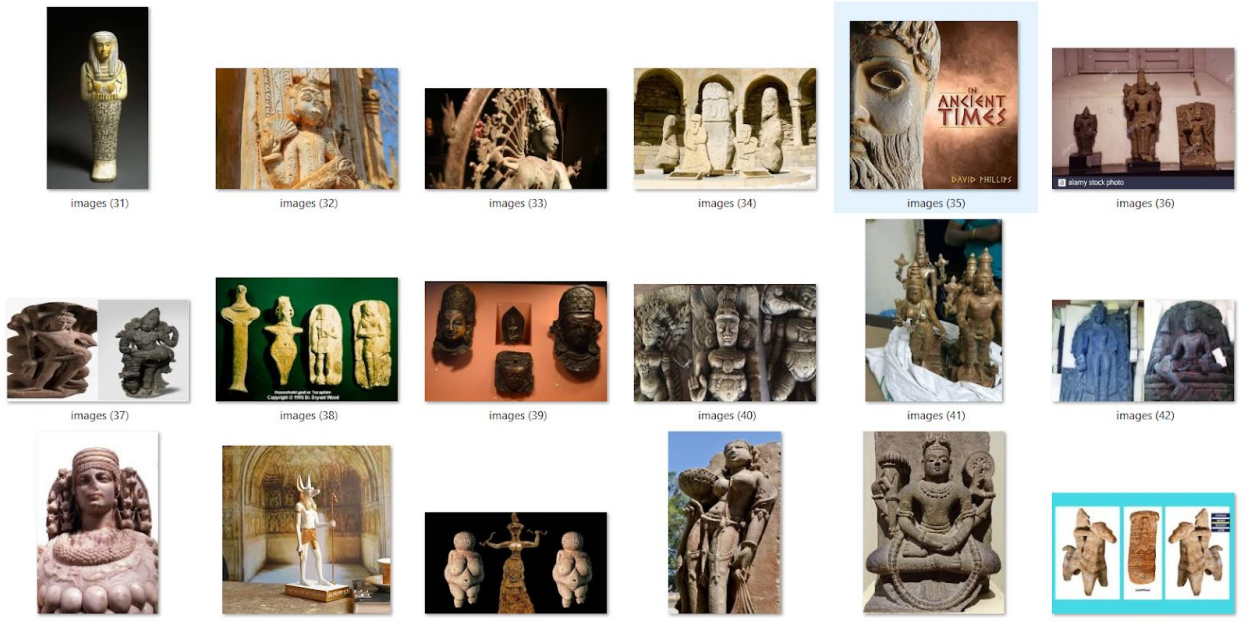
I chose to use images of idols as dataset A, to symbolically represent the machine's idolatry, or failure to recognize its creator human. For dataset B, I tested it with images of human faces, or really anything else but dataset A.

For dataset A, my initial idea was images of ancient idols and figurines. After some online research, I fail to find a ready-made dataset of decent size. I explored the option of scraping from google and bing searches. I found Hardik Vasa's Bing Scraper python tool[4], which is able to scrape all images from a single image search on Bing. Here are some results I was able to download.

[4] https://github.com/ultralytics/google-images-download



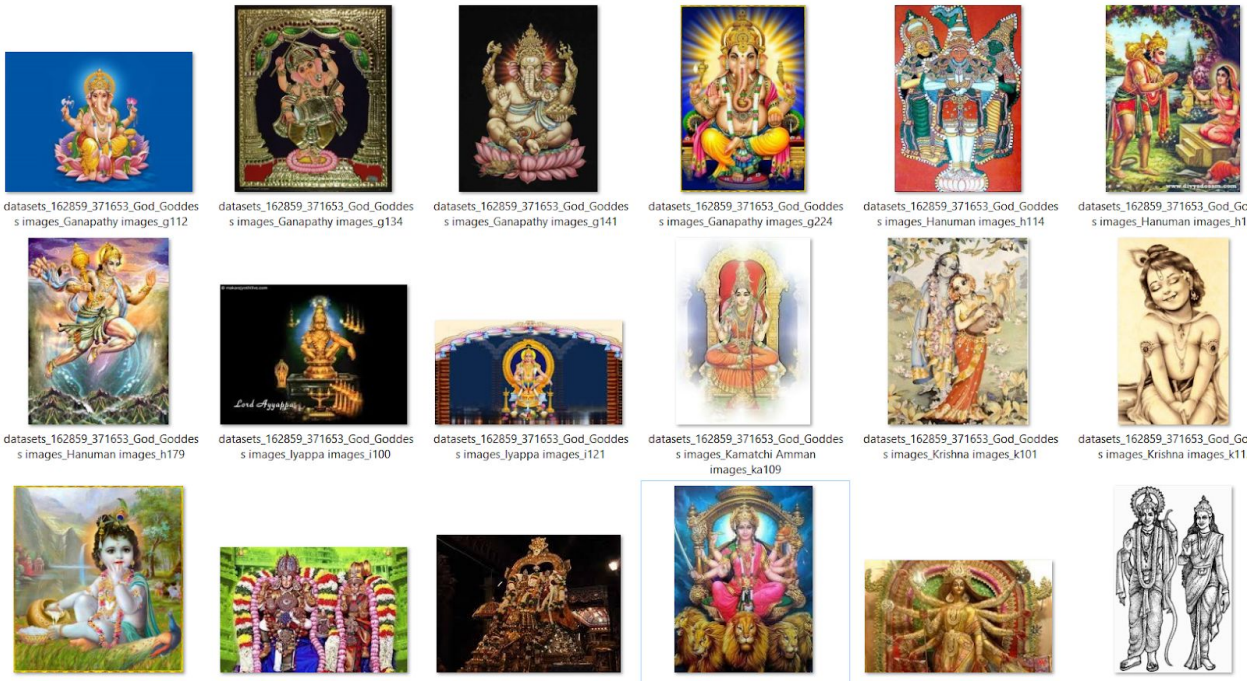
The problem with this is that Bing only showed around 150 images, therefore failing to scrape any more than that. Next, I looked into a Chrome extension Fatkun Batch Download Image offered by AITUXIU[5]. The extension allows a batch download of all images displayed on a single webpage. Here are some results I obtained from a Google Image search.



5 https://chrome.google.com/webstore/detail/fatkun-batch-download-ima/nnjjahlikiabnchcpehcpkdeckfgnohf?hl=en

In order to increase the number of downloaded images, I must keep scrolling down the page. The issue with this approach is that I must prune through the dataset, as there were many irrelevant or text-based images. Like Bing, the number of results are also limited to around 400.

I finally resorted to a ready-made dataset on Kaggle with 5,818 files.
TCE_Religious_Idols_Database by SP Praveen[6] contains 13 classes of Hindu gods and goddesses, ranging from 88 to 1,097 images per class.



Processing dataset

Since the images in the dataset had inconsistent resolutions and dimensions, I first processed all images to 128 x 128 pixels with 3 color channels. I used Python on Google Colab for processing, and later the same environment in training also. First I uploaded the dataset to Google Drive, mounted it in Colab, and unzipped it using `!unzip`.

I coded a dictionary for ease of accessing the names of the folders.

```
#dictionary
image_path = "/content/gdrive/My Drive/Colab Notebooks/gan/unzipped/God_Goddess images"
num2name = [file for file in listdir(image_path)]
name2num = {num2name[i]:i for i in range(len(num2name))}
```

[6] https://www.kaggle.com/sppraveen14/tce-religious-idols-database

```
NUM_CLASS = len(num2name)
print(name2num)
```

Then, to access each jpg file on drive, I obtained each directory path and stored it into an array for each folder. The following codes are borrowed and modified from the article “Image Pre-processing” by Prince Canuma on Towards Data Science.[7]

```
def loadImages(path, folder):
    image_files = sorted([os.path.join(path, folder, file)
                          for file in os.listdir(path + "/" + folder)
                          if file.endswith('.jpg')])
    return image_files
```

The image paths were then organized into their according folder, and into the `list_data` array.

```
list_data =[]
def loadFolders():
  for i in range(NUM_CLASS):
    folder = loadImages(image_path, num2name[i])
    print('number of files in' + num2name[i], len(folder))
    list_data.append(folder)
loadFolders()
```

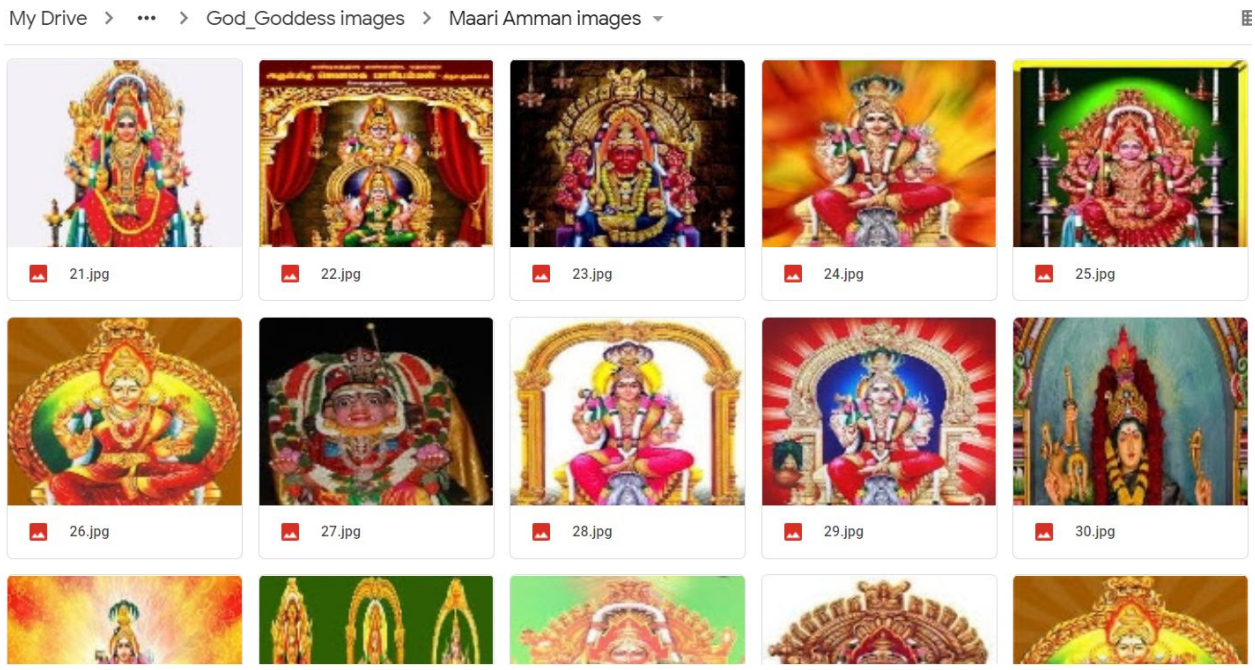
The `processing()` function will run over each jpg file in each of the folders and resize them to 128 x 128 pixels using the OpenCV library. It will then write the image into another directory in drive, keeping the same folder structure.

```
resized_path = "/content/gdrive/My Drive/Colab
Notebooks/gan/resized/God_Goddess images"
def processing(data):
  for i in range(len(data)):

    this_path = os.path.join(resized_path, num2name[i])
    if not os.path.exists(this_path):
           os.makedirs(this_path)

    for file in data[i]:
      try:
        img = cv2.imread(file)
        img_array = np.array(img)
        resized = cv2.resize(img_array, (128,128))
        print(type(resized))
        write_path = os.path.join(this_path, os.path.basename(file))

        cv2.imwrite(write_path, resized)
      except TypeError:
        print("TypeError")

processing(list_data)
```

[7] https://towardsdatascience.com/image-pre-processing-c1aec0be3edf

The images are now all resized, as shown below, pixelated and stretched when necessary. Although it is not ideal to stretch the image, I chose this way for simplicity and ease.

.



I stored each folder of images as a numpy array with the shape (num_of_images, 128, 128, 3).

```
results12 = []
def processing(data,i,arr):
    for file in data[i]:
      try:
        img = cv2.imread(file)
        img_array = np.array(img)
        arr.append(img_array)
        print(len(arr))
      except TypeError:
        print("TypeError")

processing(list_data,12,results12)
np12 = np.array(results12)
print(np12.shape)
np.save("/content/gdrive/My Drive/Colab Notebooks/gan/npy/np12.npy", np12)
```

Then I reorganized these numpy arrays in `x_train`, `y_train`, `x_test`, `y_test`, `x_dataset`, and `y_dataset`. 20 percent of the data is used to test. Here, I show the code that creates the training and testing numpy arrays.

```
def getNp(num):
  path = npy_path + "/" + "np" + str(num) + ".npy"
```

```
  arr = np.load(path)
  return arr
x_train = np.empty([0,128,128,3])
x_test = np.empty([0,128,128,3])
y_train = []
y_test = []

for i in range(NUM_CLASS):
  arr = getNp(i)
  x, _, _, _ = arr.shape
  upto = int(x*0.8)
  x_train = np.append(x_train, arr[0:upto], axis = 0)
  x_test = np.append(x_test, arr[upto:], axis = 0)
  for _ in range(upto):
    y_train.append(i)
  for _ in range(x-upto):
    y_test.append(i)

y_train = np.array(y_train)
y_test = np.array(y_test)
y_train = np.reshape(y_train, (4644,1))
y_test = np.reshape(y_test, (1167,1))
```

Now that is done, the data is ready to be used to train a classifier (with x_train, y_train, x_test, y_test) or a GAN (with x_dataset).

CNN Classifier

Since this is my first project with machine learning, I started with a simple classifier to test the waters. I modified the CNN code we used in class[8] using tensorflow.keras. The following shows the model summary whose parameters I chose quite arbitrarily.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 126, 126, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 63, 63, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 61, 61, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 30, 30, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 28, 28, 64)        36928
_________________________________________________________________
flatten (Flatten)            (None, 50176)             0
_________________________________________________________________
dense (Dense)                (None, 64)                3211328
```

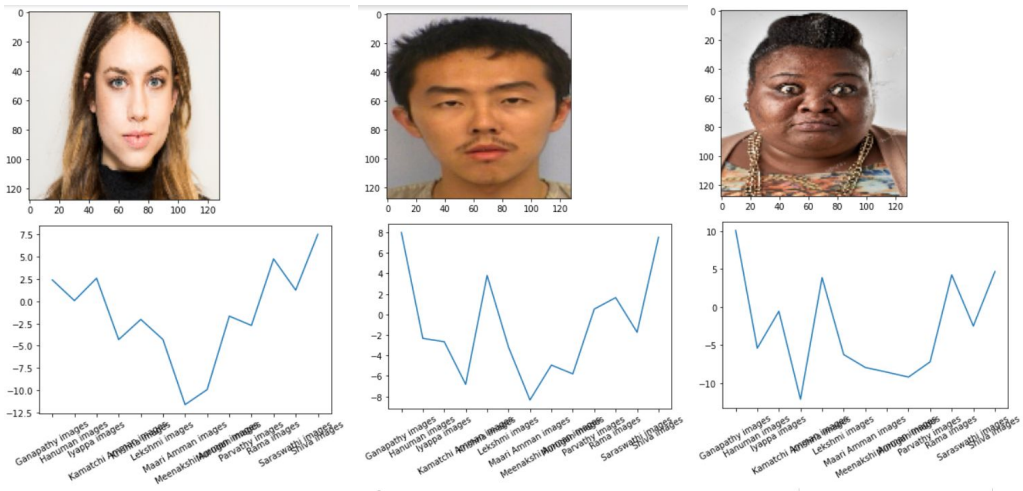
[8] https://colab.research.google.com/drive/1KlqNDOV2t_ehgWXRprmcrFTGiqAOS5nt

```
_________________________________________________________________
dense_1 (Dense)              (None, 13)                845
=================================================================
Total params: 3,268,493
Trainable params: 3,268,493
Non-trainable params: 0
_________________________________
```

The following shows the optimizer, loss of the model, number of epochs and batch size used in the training.

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
EPOCHS = 20
history = model.fit(x_train, y_train, epochs=EPOCHS, batch_size=200, shuffle=1)
```

After 20 epochs, the model can accurately classify the x_train data 99.55%. However, when evaluated using the x_test data, it returns an accuracy of 44.47%. That is not so important, since I wanted to move on to the GAN. Below shows the prediction results when the model is given data outside of its knowledge: it thinks the people belong to one of the idol classes.



DCGAN architecture

Moving on to the DCGAN, I based my code off of Tensorflow.org's DCGAN tutorial[9] with keras. The dataset values were normalized to a range between -1 and 1, shuffled and divided into batch size of 200. The generator will take a noise input with shape (100,). After each training epoch, the generator will save 28 image outputs from a preset array of random seed into a png file.

[9] https://www.tensorflow.org/tutorials/generative/dcgan

```
x_dataset = (x_dataset - 127.5) / 127.5 # Normalize the images to [-1, 1]
BUFFER_SIZE = 500
BATCH_SIZE = 200
NOISE_SIZE = 100
PREVIEW_ROWS = 4
PREVIEW_COLS = 7
seed = tf.random.normal([PREVIEW_COLS*PREVIEW_ROWS, NOISE_SIZE])
train_dataset =
tf.data.Dataset.from_tensor_slices(x_dataset).shuffle(BUFFER_SIZE).batch(BATCH_
SIZE)
```

The image saving code is adopted from the article "Generating Modern Art using Generative Adversarial Network(GAN) on Spell" by Anish Shrestha on Towards Data Science[10]. Initially I tried to adopt from this source the architecture as well. However, such architecture used so much memory that it returned an error. I decided to revert to my original code.

I had to adjust the architecture so that the generator output and discriminator output takes a shape of (None, 128, 128, 3), the size of the training images. The generator uses a filter size of (5, 5), a stride of (2, 2), and takes three Conv2DTranspose layers with 128, 64, 3 filters respectively. Below shows the generator summary.

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 65536)             6553600
_________________________________________________________________
batch_normalization_3 (Batch (None, 65536)             262144
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 65536)             0
_________________________________________________________________
reshape_1 (Reshape)          (None, 16, 16, 256)       0
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 32, 32, 128)       819200
_________________________________________________________________
batch_normalization_4 (Batch (None, 32, 32, 128)       512
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 32, 32, 128)       0
_________________________________________________________________
conv2d_transpose_4 (Conv2DTr (None, 64, 64, 64)        204800
_________________________________________________________________
batch_normalization_5 (Batch (None, 64, 64, 64)        256
_________________________________________________________________
leaky_re_lu_8 (LeakyReLU)    (None, 64, 64, 64)        0
_________________________________________________________________
conv2d_transpose_5 (Conv2DTr (None, 128, 128, 3)       4800
=================================================================
```

10 https://towardsdatascience.com/generating-modern-arts-using-generative-adversarial-network-gan-on-spell-39f67f83c7b4

```
Total params: 7,845,312
Trainable params: 7,713,856
Non-trainable params: 131,456
_________________________________________________________________
```

The discriminator uses the same filter and stride size as the generator, with three Conv2D layers and three Dropout layers of 30%. Below shows the discriminator summary.

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 64, 64, 64)        4864
_________________________________________________________________
leaky_re_lu_9 (LeakyReLU)    (None, 64, 64, 64)        0
_________________________________________________________________
dropout_3 (Dropout)          (None, 64, 64, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 32, 32, 128)       204928
_________________________________________________________________
leaky_re_lu_10 (LeakyReLU)   (None, 32, 32, 128)       0
_________________________________________________________________
dropout_4 (Dropout)          (None, 32, 32, 128)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 16, 16, 128)       409728
_________________________________________________________________
leaky_re_lu_11 (LeakyReLU)   (None, 16, 16, 128)       0
_________________________________________________________________
dropout_5 (Dropout)          (None, 16, 16, 128)       0
_________________________________________________________________
flatten_1 (Flatten)          (None, 32768)             0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 32769
=================================================================
Total params: 652,289
Trainable params: 652,289
Non-trainable params: 0
_________________________________________________________________
```

The following shows the loss calculation of both models. The discriminator decreases in loss when it outputs ones for real images and zeros for fake images. The generator decreases in loss when the discriminator gives ones to its fake images.

```
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

With this loss and Adam optimizers for both models, each training step uses the gradient tape to adjust its weights. The weights are stored on drive using a checkpoint manager, keeping the last three checkpoints.

```
checkpoint_dir = "/content/gdrive/My Drive/Colab
Notebooks/gan/training-checkpoints"
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,

discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)
manager = tf.train.CheckpointManager(
    checkpoint, directory= checkpoint_dir, max_to_keep=3)

if manager.latest_checkpoint:
  checkpoint.restore(manager.latest_checkpoint)
  print ('Latest checkpoint restored!!')
```

Training

For each training epoch, it trains on the image batch, and then outputs the a png file of 28 generated images, saved to drive. Every 15 epochs, the manager saves the checkpoint.

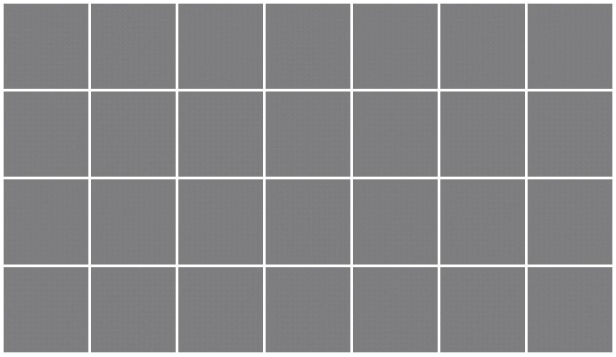
```
def train(dataset, epochs, starting_epoch):
  for epoch in range(epochs):
    start = time.time()

    for image_batch in dataset:
      train_step(image_batch)

    # Produce images for the GIF as we go
    display.clear_output(wait=True)
    save_images(generator,
                            epoch + 1 + starting_epoch,
                            seed)

    # Save the model every 10 epochs
    if (epoch + 1) % 15 == 0:
      ckpt_save_path = manager.save()
      print ('Saving checkpoint for epoch {} at {}'.format(epoch+1 +
starting_epoch,
                                                           ckpt_save_path))

    print ('Time for epoch {} is {} sec'.format(epoch + 1 + starting_epoch,
time.time()-start))

  # Generate after the final epoch
  display.clear_output(wait=True)
  save_images(generator,
                           epochs + starting_epoch,
                           seed)
```

```
ckpt_save_path = manager.save()
```

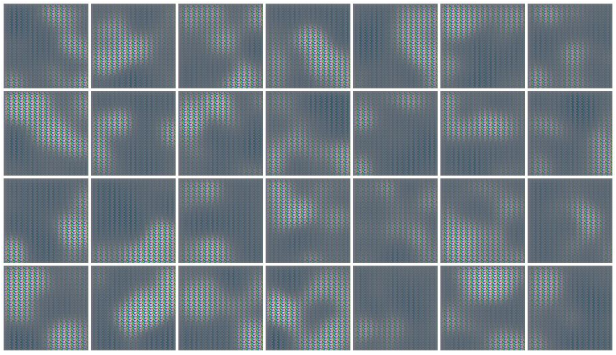
The images and checkpoints saved used up a large amount of storage, since the checkpoints weren't truly deleted but were placed in the trash folder. I registered for and used up three google accounts to switch the data to keep training. After this, I decided to use the university gapps account which provided unlimited storage. With this, I continued to train until 10,000 epochs and used up 64 GB. The training took 4 days, from 10 to 50 seconds per epoch.

Output

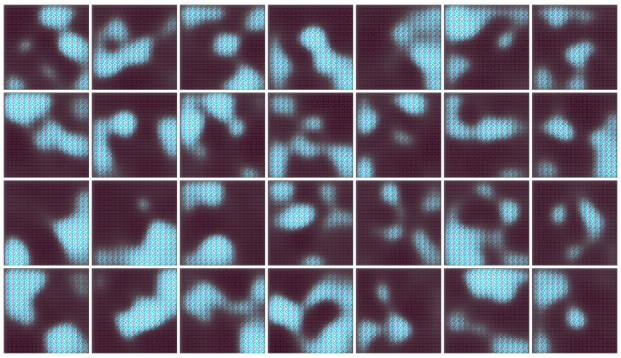
The learning curve in the beginning was quick, adding details rapidly.



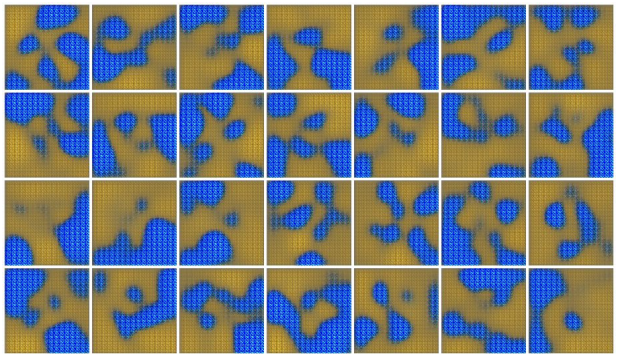
Epoch 1



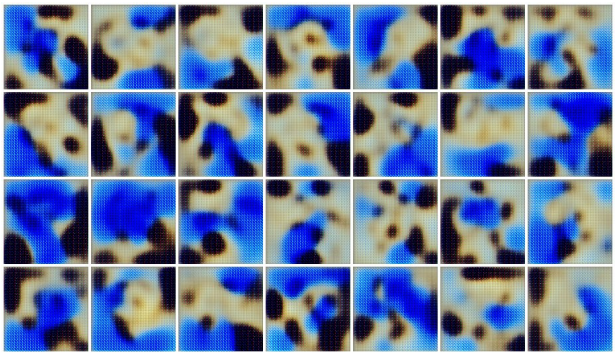
Epoch 7



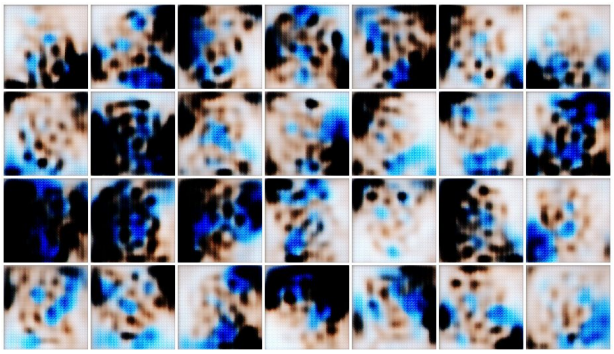
Epoch 14



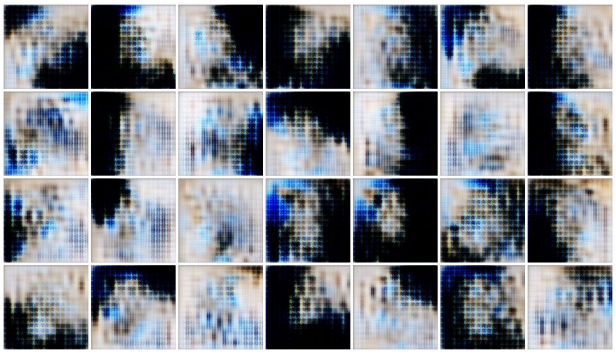
Epoch 21



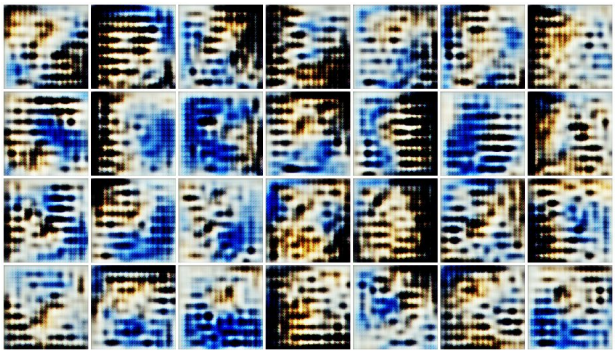
Epoch 28



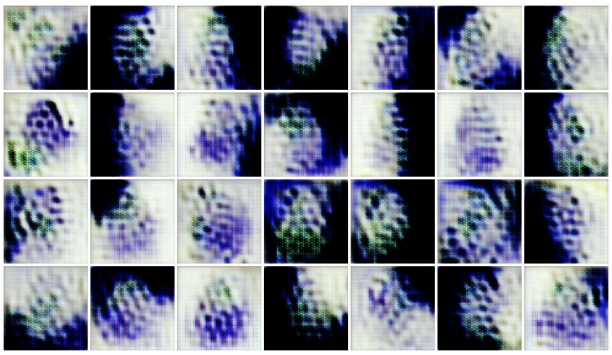
Epoch 50



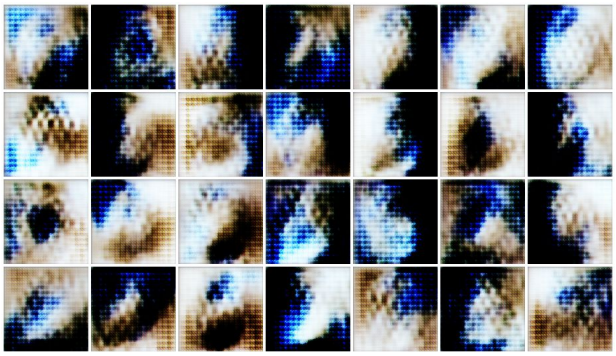
Epoch 60



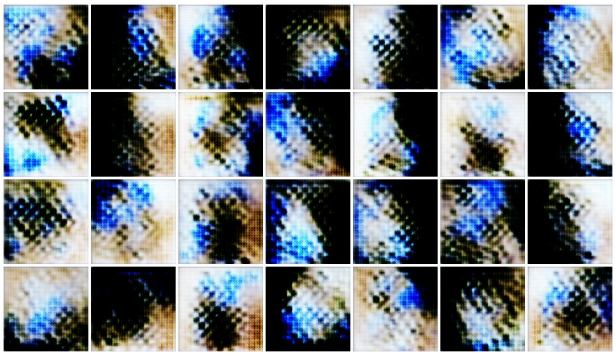
Epoch 70



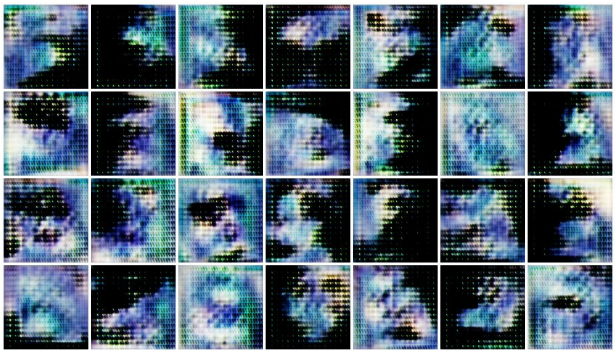
Epoch 80



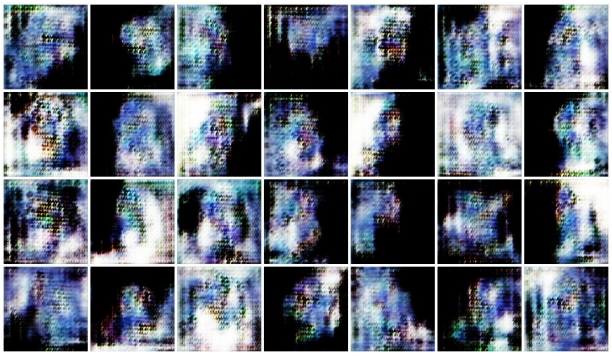
Epoch 90



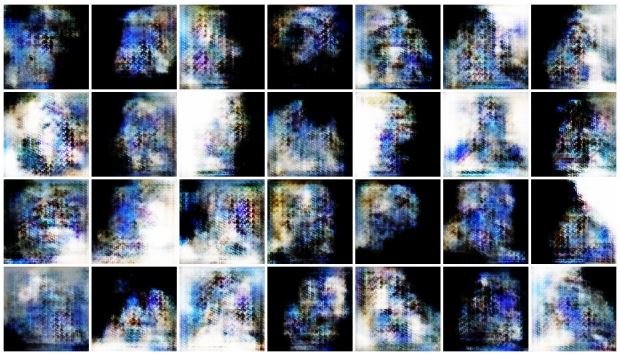
Epoch 100



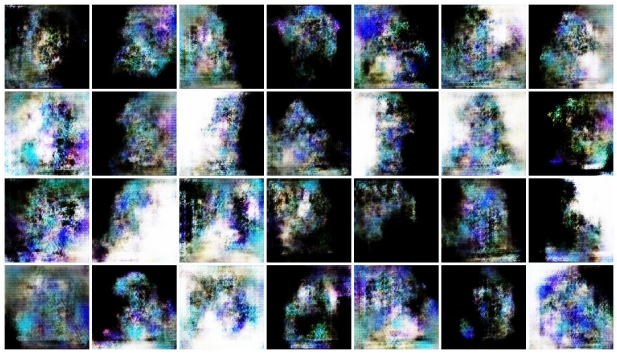
Epoch 150



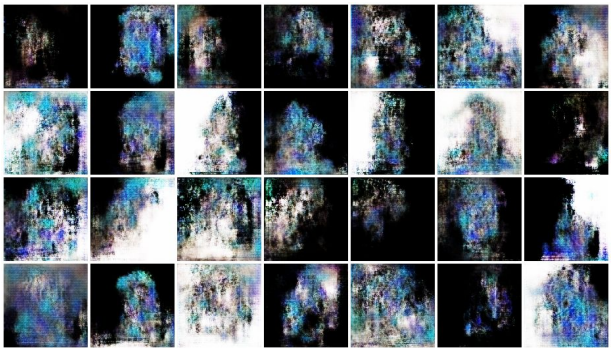
Epoch 200



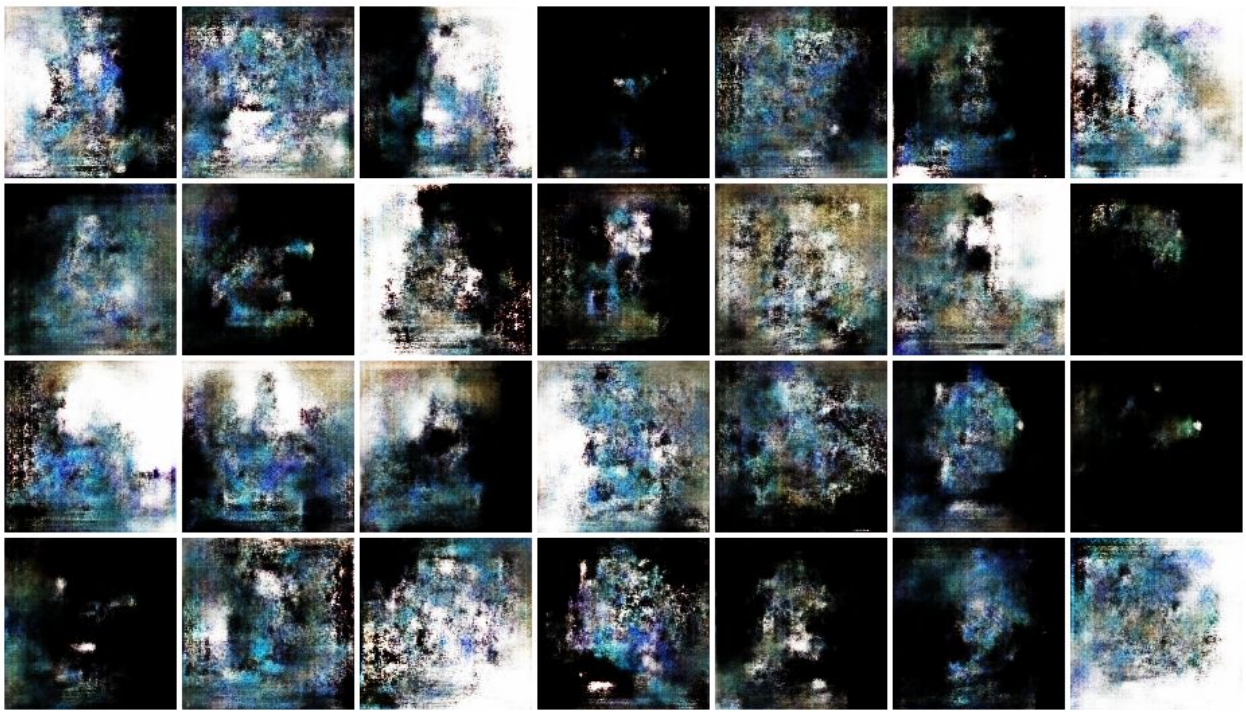
Epoch 400



Epoch 600

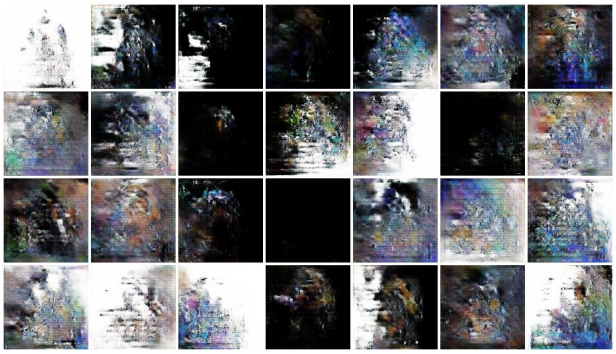


Epoch 800

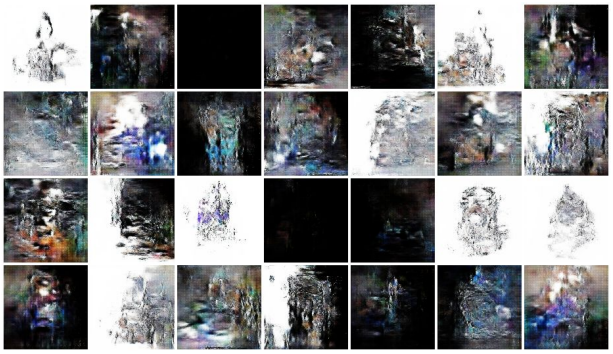


Epoch 1,000

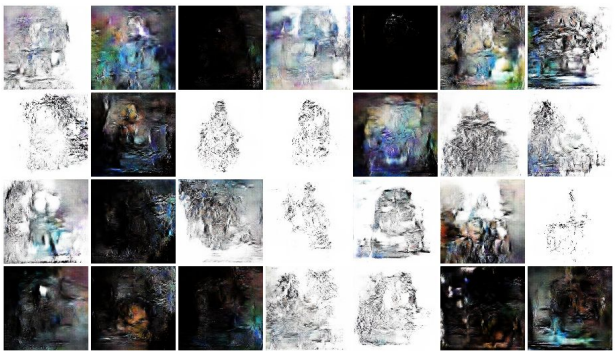
The images between each epoch began to look less different the later down. Besides pointing out how aesthetically pleasing these images looked, almost like oil paintings, I could start making out shapes of some idol-forms sitting facing center. Sometimes, the images would seemingly regress before progressing again.



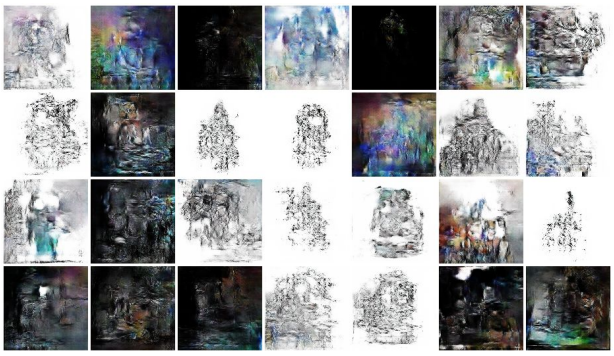
Epoch 1,500



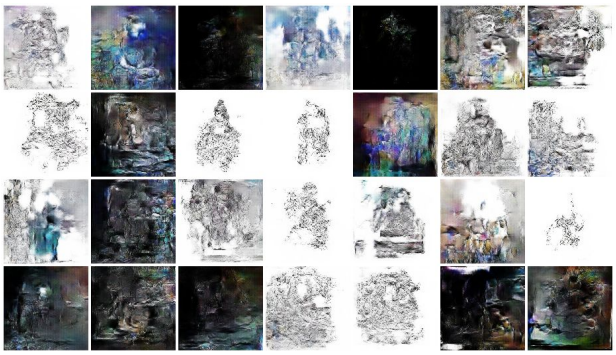
Epoch 2,000



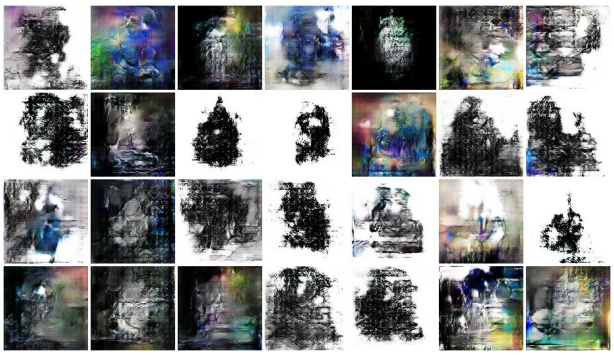
Epoch 2,500



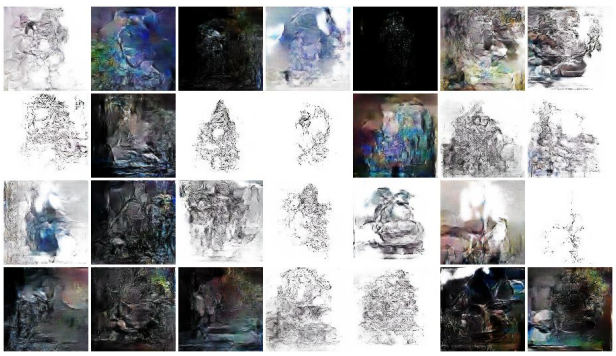
Epoch 3,000



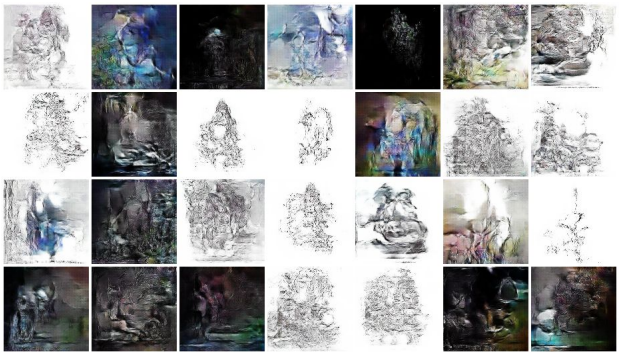
Epoch 3,500



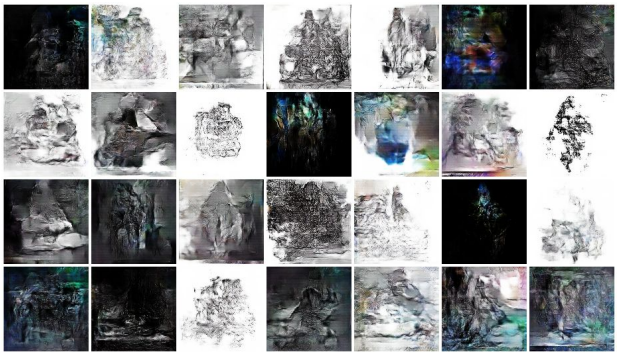
Epoch 4,000



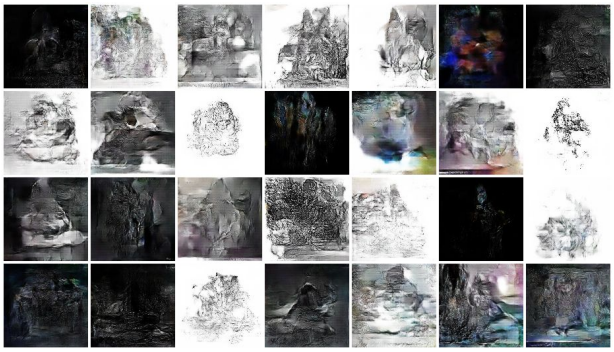
Epoch 4,500



Epoch 5,000

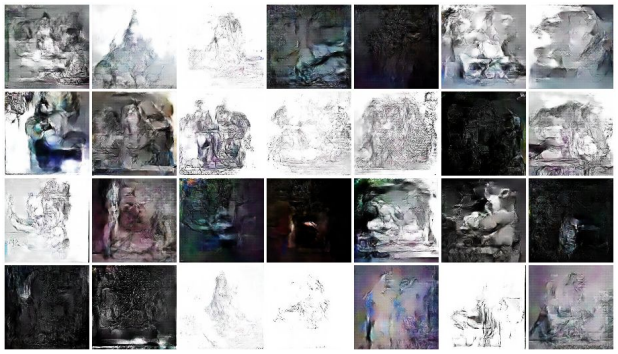


Epoch 6,000

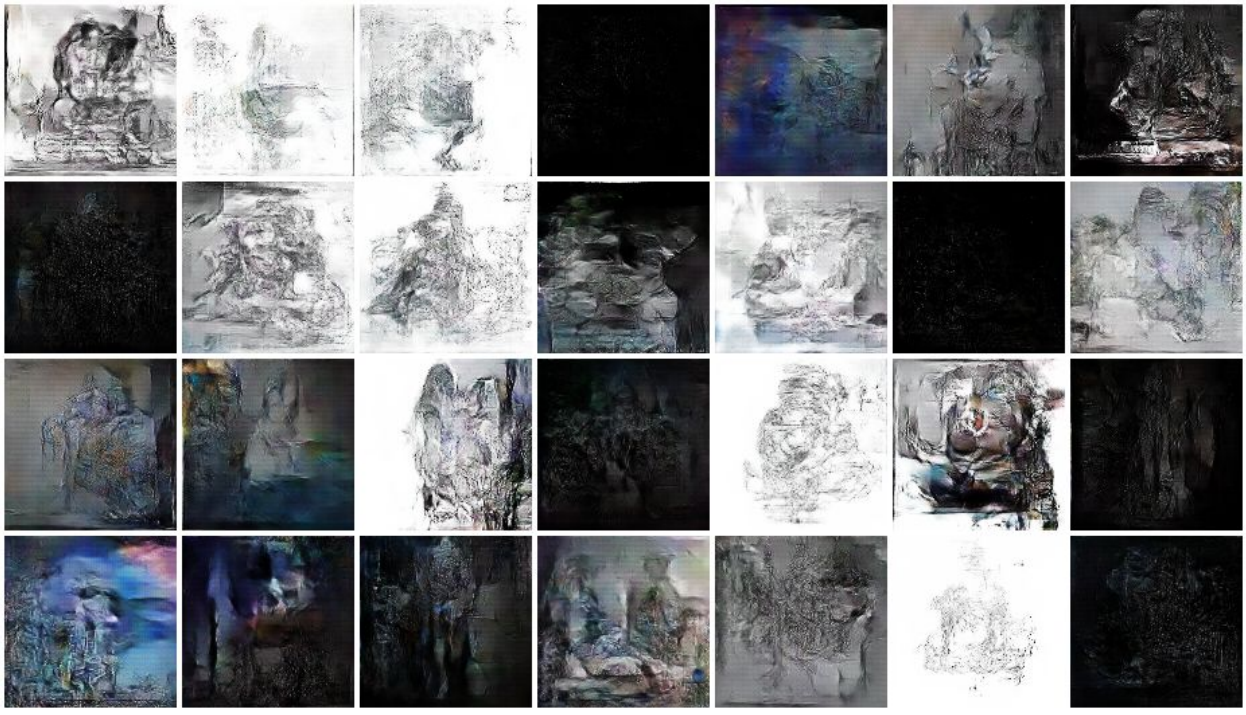


Epoch 7,000

Epoch 8,000



Epoch 9,000

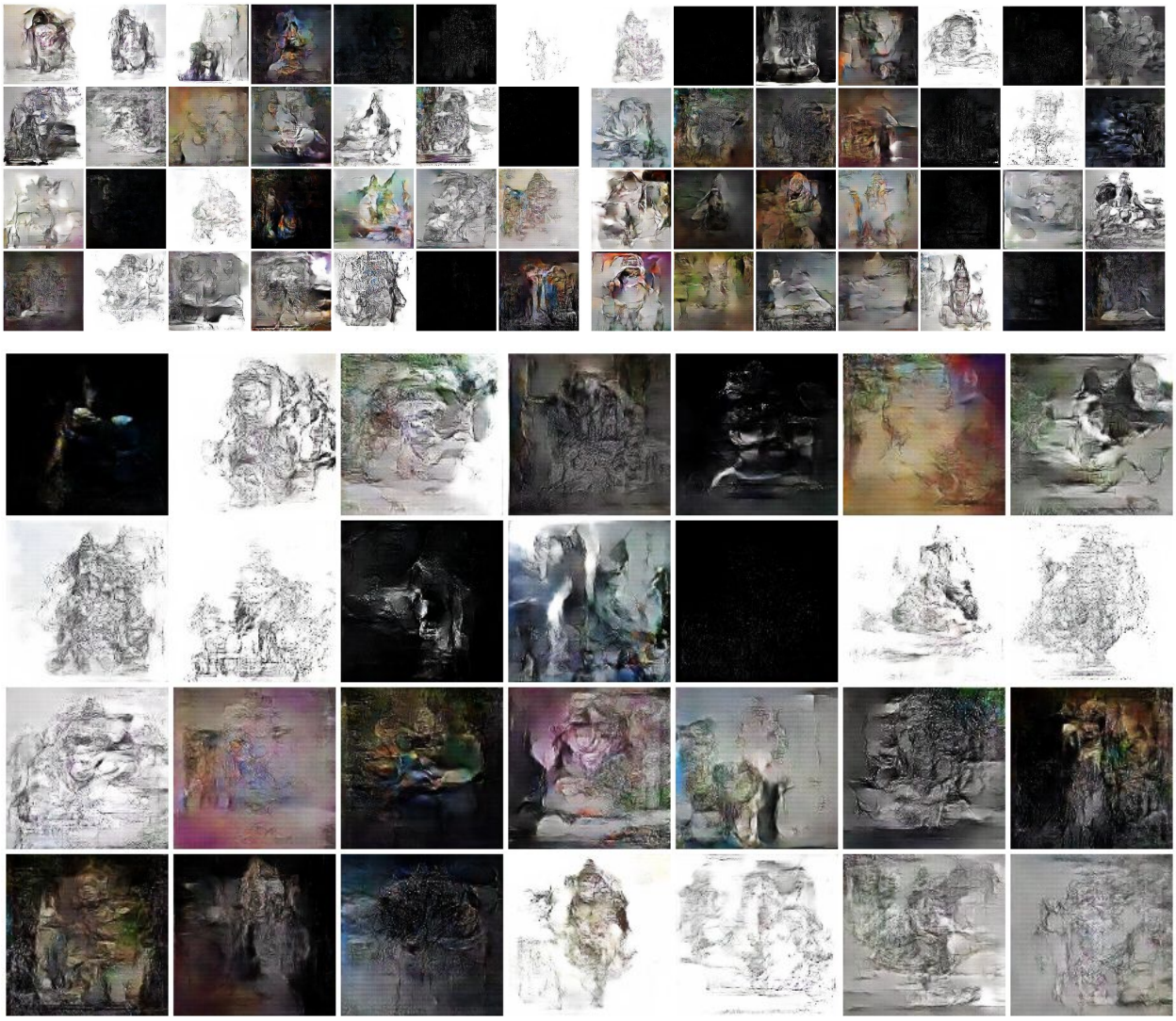


Epoch 10,000

I discovered quite late that the colors in my output were distorted due to an OpenCV problem. I forgot to convert them to RGB colorspace. However, since I amended this only after training, I would not take the time to generate the correct RGC images from each epoch.

```
RGB_img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

Here, I show the actual generated images after 10,000 epochs. This corresponds with the training data, as the idols are brownish of color.



Seemingly, after 10,000 epochs, the machine failed to produce high-resemblance images to its given dataset A. There are some similarities, however, such as the complex lines and vague shape of some sitting form. One can argue that the machine is capable of creativity in this sense, to stray from its given knowledge. This, though, can be attributed to its noise input, a pseudo-randomness that can drive “new” ideas. Its “new” ideas, at the same time, does not

resemble anything else we know, such as apples, houses, cars, but merely forms that have travelled down the path away from noise and towards dataset A. To further push that the machine is incapable of the kind of intelligence to create something meaningful which it does not know, the machine will only make sense of any meaningful input with its narrow worldview of dataset A. To visualize this I fed the machine with images of human faces to see how the machine interprets and twists it into its knowledge.

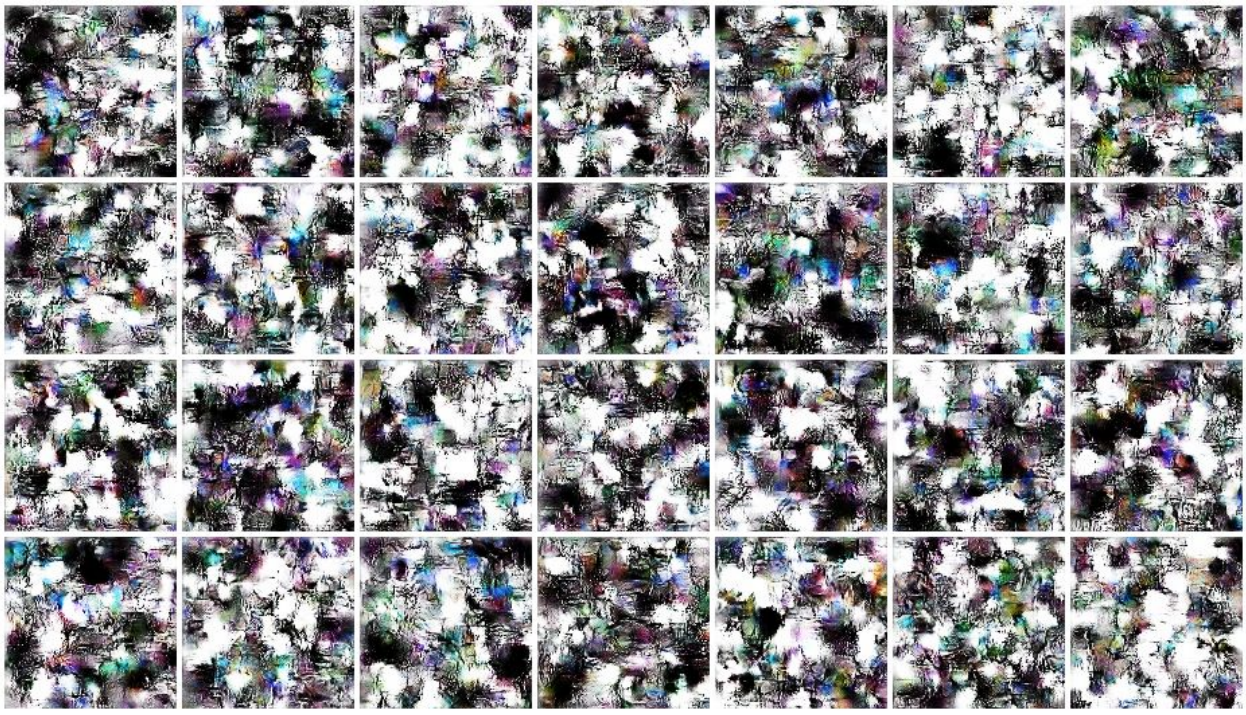
The Ignorance

In order to feed the GAN with images, I must amend the input shape from (100,) to something compatible. I extracted the top layers and visualized what it sees from there.

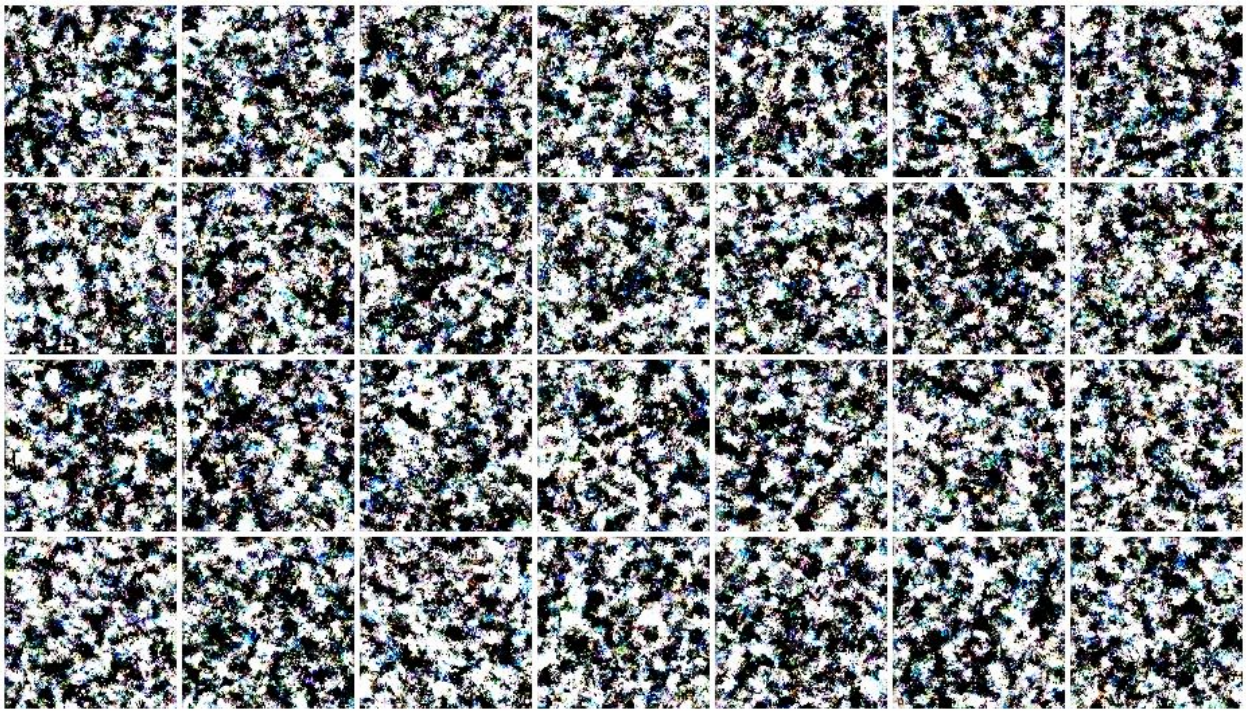
```
def make_feature_extractor3():
    model = tf.keras.Sequential()
    model.add(tf.keras.Input(shape=(16,16,256,)))
    for layer in generator.layers[4:]:
      model.add(layer)
    return model
```

For testing I used noise to visualize what the model sees, keeping different numbers of top layers.

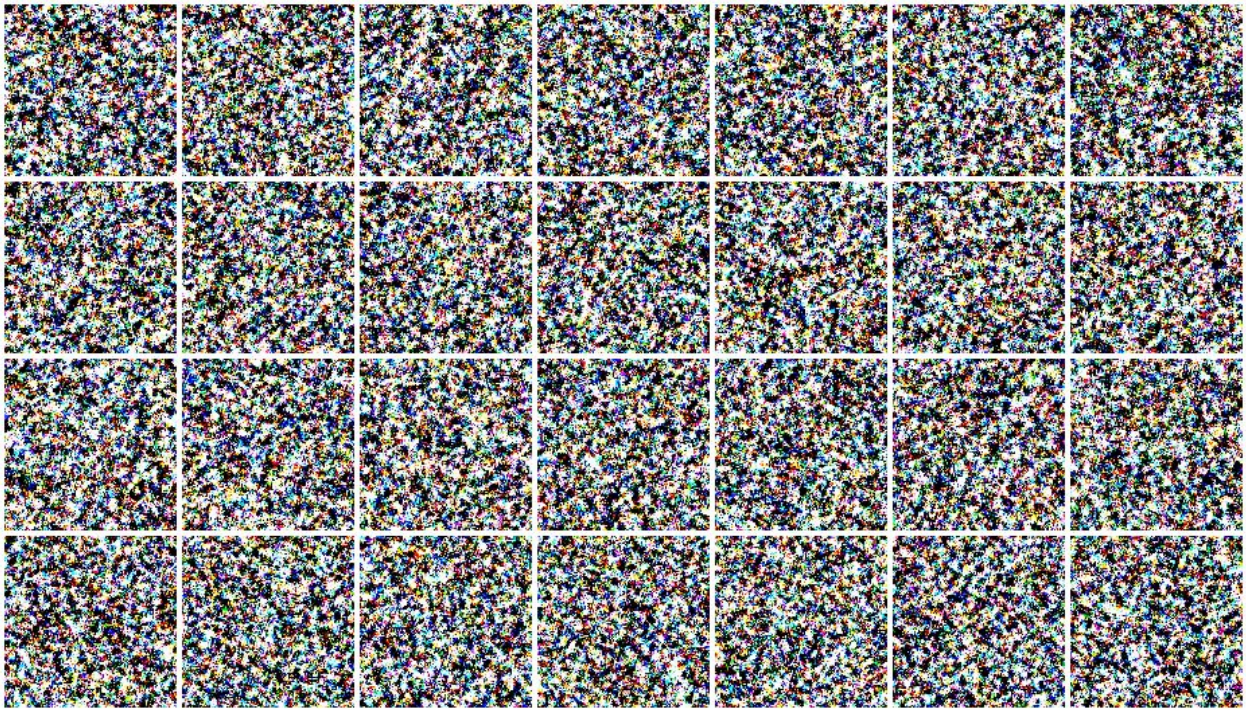
Last 3 Conv2DTranspose layers kept:



Last 2 Conv2DTranspose layers kept:



Last 1 Conv2DTranspose layers kept:



I used the Flickr-Faces-HQ dataset from NVIDIA Research Projects[11], the 128 x 128 thumbnails for the input. I first resized the images so they fit the three feature extractors above, 16 x16, 32 x 32, 64 x 64, respectively.

```
def processing(res):

  this_path = os.path.join(faces_path, str(res))
  if not os.path.exists(this_path):
           os.makedirs(this_path)

  for i in range(50):
      try:
        filename = str(i) + ".png"
        file = os.path.join(faces_path, filename)
        img = cv2.imread(file)
        img_array = np.array(img)
        resized = cv2.resize(img_array, (res, res))
        write_path = os.path.join(this_path, filename)

        cv2.imwrite(write_path, resized)
      except TypeError:
        print("TypeError")
```

After resizing and saving them into drive, I arranged them into an array to feed into the feature extractor. I first put 28 images together into a numpy array.

```
def get_faces_array(res):
  this_list = []
  for image_count in range(28):
      get_path = os.path.join(faces_path, str(res))
      filename = str(image_count) + ".png"
      file = os.path.join(get_path, filename)
      img = cv2.imread(file)
      this_list.append(img)
  arr = np.array(this_list)
  return arr
```

Then from there, I doubled that array until the channels scaled up from 3 to the desired shape. Whatever leftover that is not filled up is filled up by noise. The seed is normalized between -1 and 1.

```
import math
seed16 = get_faces_array(16)
def loop_channels(seed, channel, res):
  for i in range(int(math.log(channel/3,2))):
    seed = np.append(seed, seed, axis = 3)
  _, _, _, c = seed.shape
  ran = tf.random.normal([28, res, res, channel - c])
```

[11] https://github.com/NVlabs/ffhq-dataset

```
  seed = np.append(seed, ran, axis = 3)
  seed = (seed-127.5)/127.5
  return(seed)
seed16 = loop_channels(seed16, 256, 16)
```

The seed is ready to be fed into the feature extractor. Since there were three extractors containing different numbers of top layers, I created three human face seeds to feed into them. Here, I only showed the code for the feature extractor that contains three Conv2DTranspose layers with an input made from 16 x 16 face images.

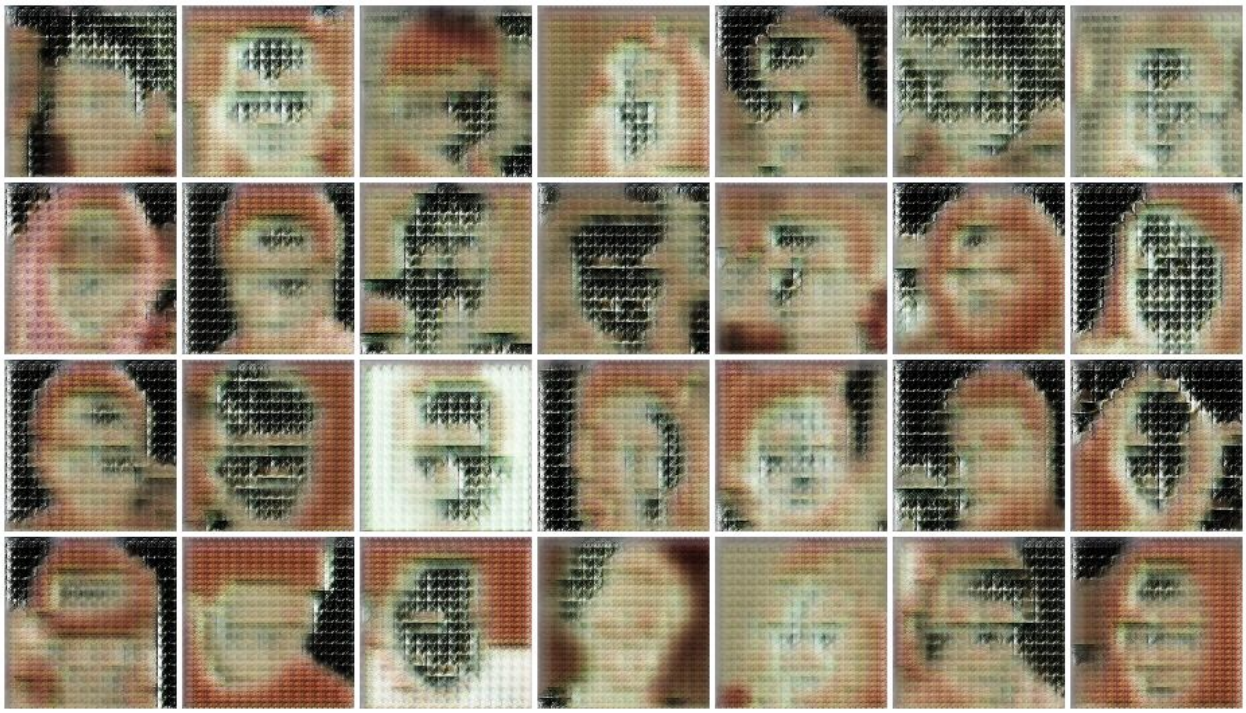
```
save_images(feature_extractor3, "faces3-001", seed16)
```

Original seed input:

Feature extractor1: top 1 Conv2DTranspose layer:

Feature extractor2: top 2 Conv2DTranspose layer:

Feature extractor3: top 3 Conv2DTranspose layer:



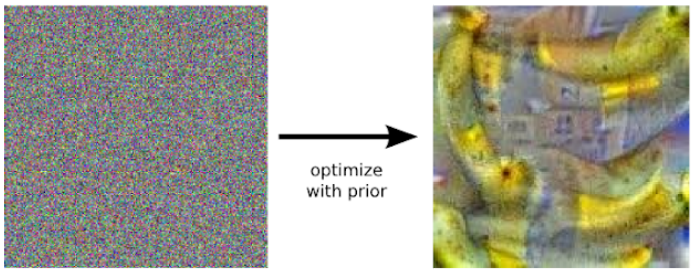
Evidently, extracting only the top 1 layer would maintain more details yet encourage color distortions, since there were less filter changes yet a larger stray from the input normalization. The result from feature extractor3 shows more interpretation of the machine as it is trained to mold the input with its previously acquired knowledge.

Conclusion

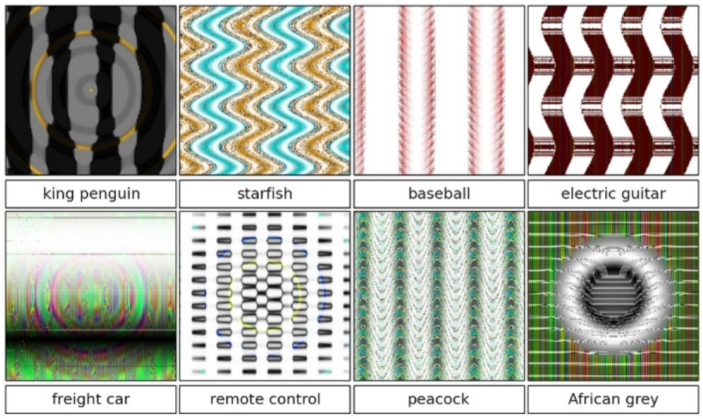
Although artificial intelligence has often been depicted as self-determined minds, machines are nothing but complex trained functions. It is limited by its realm of knowledge, which is prescribed by its creator. Ignorance of its own ignorance is hereby illustrated by the model's inability to adapt to new inputs. However, another aspect was revealed in this process, a "creativity" that is unearthed in training - randomness, error, exploration. A child may be trained to walk by his parents, who never taught him how to fall, yet a child in his attempt to walk will fall in the process. In this way, the child has lost his ignorance of falling, although the only knowledge instructed him was walking. All in all, it spurs on the question of whether one can learn beyond oneself in further exploration concerning the ignorance of artificial intelligence.

Similar Works

DeepDream by Google engineer Alexander Mordvintsev[12] creates dream-like images that "hallucinate" certain enhanced patterns from its trained dataset. The neurons are trained to see things a certain way regardless of whether the new input actually consists of that feature. The image is tweaked to excite that feature more and more, exposing what the network "sees." This is an interesting work in that it provides a peephole into what the black box is seeing. My artwork also aimed to reveal this hidden bias, but focused on its inability to cope with new data. In my work, the model failed to handle the unknown data properly and thus can only "see" the data it was trained on.



In a research "Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images", Nguyen, Yosinski, and Clune[13] explore images that are unrecognizable to human eyes but that DNNs classify into a certain class with high confidence. The results demonstrated that neural networks can be easily fooled, since their understanding is limited to classifying certain kinds of data only. This work explores the fixed mindset of machines in recognizing objects while my work reveals the fixed mindset in generation and creativity. NNs, and raise questions about the generality of DNN computer vision.



[12] https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html
[13] Nguyen A, Yosinski J, Clune J. Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images. In Computer Vision and Pattern Recognition (CVPR '15), IEEE, 2015.