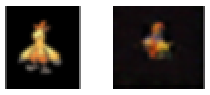
Drake Bennion

Machine Learning Engineer Nanodegree

February 2021

Using Generative Adversarial Networks to Generate Pokemon



A real Pokemon and an image generated in this project

Definition

Project Overview

Generative Adversarial Networks (GANs) are a machine learning approach to learning underlying distributions of input data in order to output data that convincingly imitates the input. In theory, GANs can learn from and imitate any kind of data, from text and images to audio and video. Most famously, GANs are the underlying algorithm generating Deepfakes, imitating video and audio of celebrities [1]. This project applies GANs to images of Pokemon, a popular Japanese video game franchise, in order to generate new sprite images that attempt to be convincing to any discerning Pokemon fan.

Some faces generated by a GAN

Problem Statement.

In the first Pokemon game released in Japan in 1996, Nintendo created 151 unique Pokemon. Every 3-5 years since then, Nintendo has released another “main” game introducing a new batch of roughly 100 Pokemon species, bringing the current total to 898. A generator for new Pokemon could help the designers and artists at Nintendo if they find themselves needing more ideas as new games are being developed. Every Pokemon has its own sprite image, and some even have different sprite images in different games. The goal of this project is to take those sprite images and train a GAN to generate new Pokemon that look similar to the original, while still hopefully being unique enough to clearly be new.

Metrics

The “convincingness” of a GANs output is obviously too subjective a measure to reliably base performance on, but being a relatively new machine learning technique, the measurement of GAN performance is still a somewhat open question. That said the Frechet Inception Distance, or FID score which measures both the quality and diversity of one dataset compared to another dataset, is a commonly accepted metric for GANs and has been shown to have some positive correlation with human judgement of visual quality. Because the FID score is a measure of distance a lower score indicates a better GAN, and it theoretically has no real upper limit. The FID score will be calculated using a Pytorch library [2].

Analysis

Data Exploration.

The input data for this project consists of images scraped from a Pokemon sprite gallery archive [3]. Originally, the idea was to take the 898 images from the main page and train on those, however initial tests and some research into the problem space highlighted the fact that 898 images is far too little data for a GAN to effectively train on.

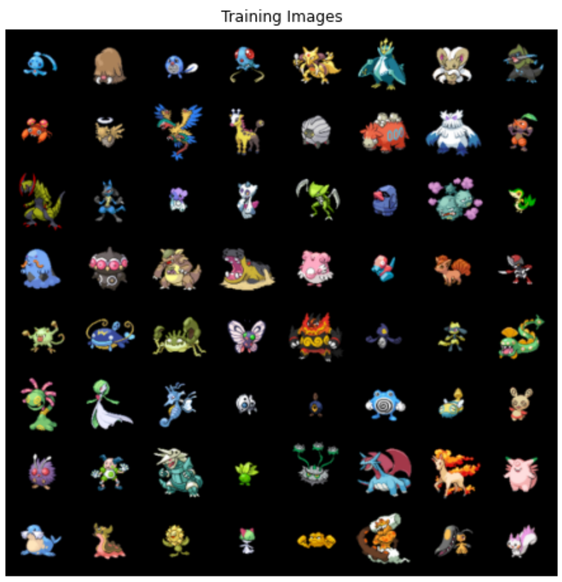
A first attempt to solve this problem was to include both the normal and "shiny" versions of the sprites which were really just different colorations of the same sprites, and to include all possible sprite versions from different generations. To clarify, Pokemon that were introduced in generation 1 were given updated sprites with every new generation.

Three different Bulbasaur sprites -- Gen1, Gen5 and Gen5 Shiny

Unfortunately, further experimentation showed these generational differences were great enough that the GAN could never come to a stable end-game, making this solution unviable. Another impediment was that some generations had random sprites with different colored backgrounds in their images, which also threw off the GAN. As a compromise, the dataset was reduced to just generation 5 sprites, both normal and shiny colorings. An idea for future improvement would be to do more pre-processing to some of the other generation images and remove those random backgrounds.

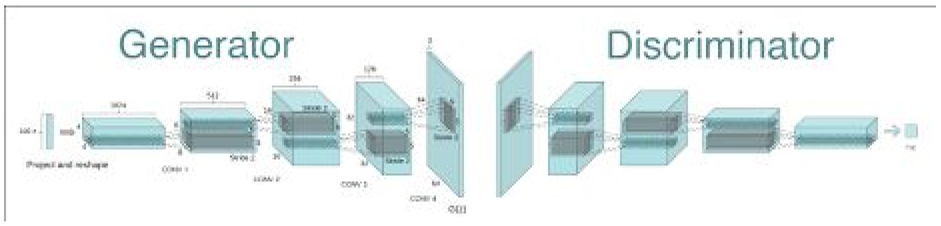
Exploratory Visualization.

As of generation 5 of Pokemon introduced in the games Black and White, 1298 normal and shiny sprite images were available. Below is a sampling of 64 of these 64x64 pixel images used in training the GAN. The overall diversity of these sprite images combined with the relatively puny size of the dataset may explain some of the resulting performance issues discussed later. One experiment in this project included converting the images to black and white to see if the GAN was able to imitate those more successfully, but this effectively reduced the dataset down to 649 unique images because the shiny sprites are truly just different colorations, and this led to GAN instability.



Algorithms and Techniques.

A GAN is actually a composition of two neural networks: one network called the generator that learns to generate data, and another network called the discriminator that learns how to determine if data is from the "real" dataset or was assembled by the generator. For this project the input to the discriminator is image data transformed into a 64x64x3 tensor (64x64 pixels and 3 RGB values), and the discriminator's output is its estimated probability that the input image is from the original dataset. The input to the generator is a random noise vector which for this project is from a random Gaussian distribution, and its output is a 64x64x3 tensor which when displayed as an image will ideally look similar to a Pokemon. The two networks improve by learning from each other in a sort of zero sum game. To train a GAN the discriminator is given a batch of entirely real data to learn on, then a batch of data created entirely by the generator which is initially just completely random, and the generator is given the discriminator's output to learn from. The generator learns what the discriminator is using to discern the real images from the fake, and the discriminator is constantly being "reminded" what to look for in the real data batches. Below is an illustration of the neural network's architectures.



GAN architecture [4]

Benchmark.

In this Paper With Code, *Prescribed Generative Adversarial Networks* the authors developed the PresGAN to solve a problem other GAN approaches tend to have called "mode collapse" [5]. Mode collapse occurs when a generator in a GAN repeatedly generates certain small sets of outputs because they once scored well against the discriminator which causes the final results of the generator to be very similar instead of reflective of the diversity of the input. This PresGAN was used on a handful of datasets including the CIFAR-10 dataset and performed well both in terms of generating realistic imitations and achieving a relatively low FID score. This implementation is currently globally ranked #45 on CIFAR-10 for its FID score of 52.202, and this gives us a ballpark estimate of what kind of FID score to aim for in this project. An FID score of under 100 for this project could be considered a success.

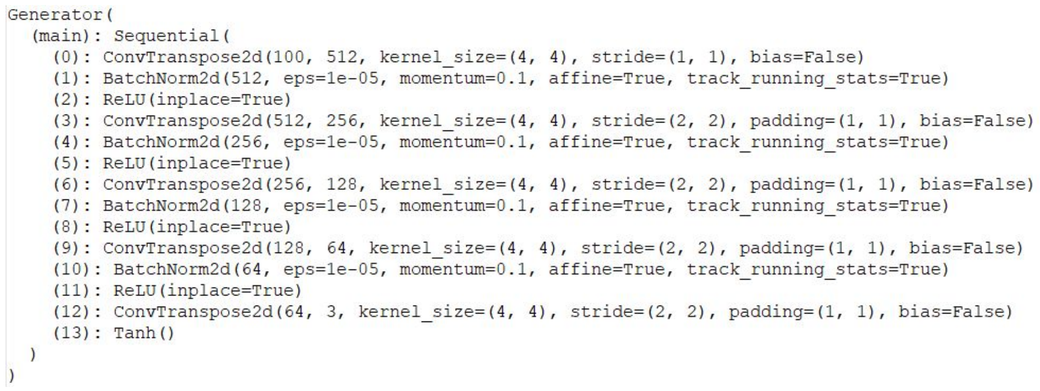
Methodology

Data Preprocessing.

Thanks to some built-in Pytorch functionality for handling data transforms and composing transformations, not much pre-processing had to be done on the images ahead of time. But those transforms are a sort of pre-processing step, so it's worth mentioning that prior to being presented to the discriminator the images were resized to the desired 64x64 pixels, center-cropped based on that image size, converted to tensors, and those tensors were normalized to a mean of 0.5 and standard deviation of 0.5. As mentioned earlier, an attempt to improve results was added here in the form of a "grayscale" transformation step, but the results were undesirable and this step was removed. It's possible as a future improvement that some image processing could be done to remove the earlier mentioned random background colors from some images, but there was no discernable pattern to which images had black, blue, white or no backgrounds so this would have to be done on all of the images to be safe.

Implementation.

The implementation of this GAN was done using Pytorch, and was heavily based on a tutorial provided by Pytorch for writing a face-generation GAN [6]. The generator network was composed of 5 convolutional layers making use of ConvTranspose2d, BatchNorm2d, ReLu, and Tanh layers. The discriminator was likewise composed of 5 convolutional layers, but using Conv2d, LeakyReLu, BatchNorm2d, and Sigmoid layers. Below is shown the Pytorch output of printing the two networks which gives a more detailed view of their architecture. During training, the loss function used was BCELoss, and both networks used Adam optimizers with betas (0.5, 0.999).

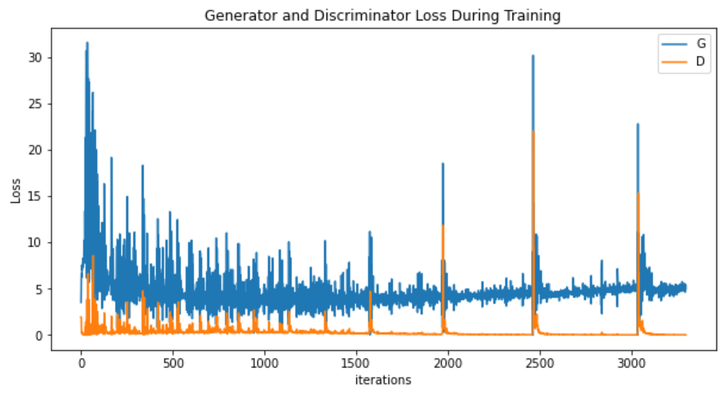


```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2,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

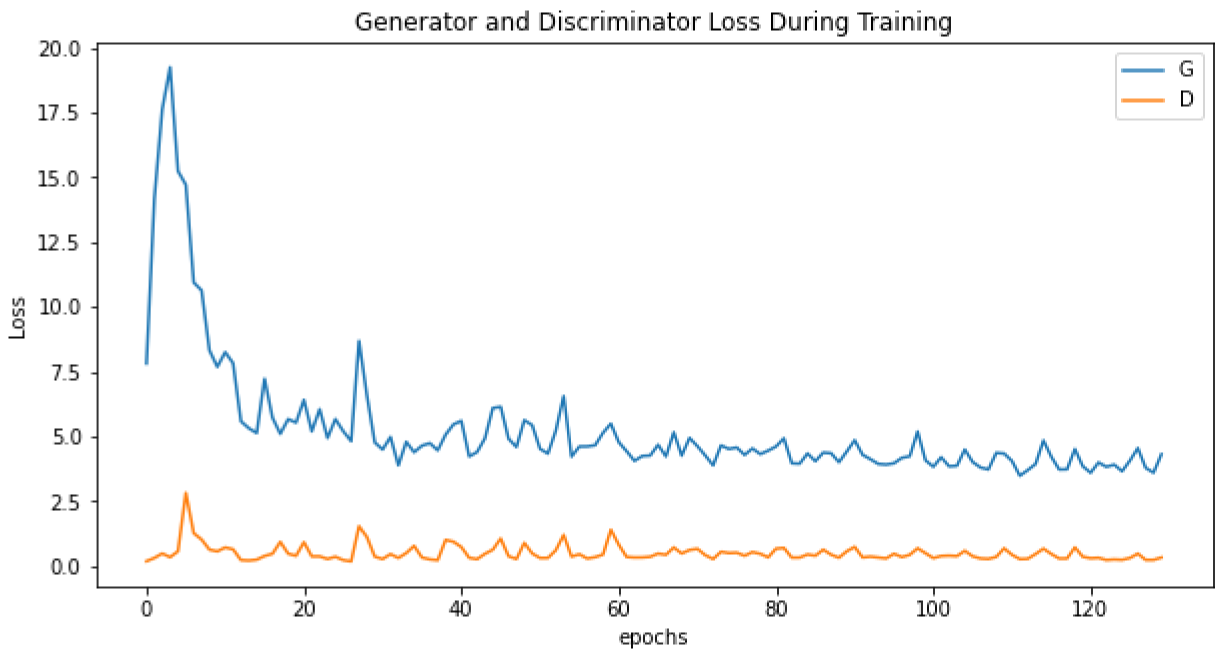
```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05,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

Refinement.

One upside to having a relatively small dataset is that it becomes more feasible to train for more epochs in a reasonable amount of time. Below is a visualization of the losses of the generator and discriminator during training. Notice that the x-axis in this image depicts the number of *iterations* in training, **not** the number of epochs. These visualizations were instrumental along the way in deciding a good number of epochs to train.



During the longest run, the loss of the generator was generally decreasing until somewhere in the 1400 range, or about 127 epochs. After that, the loss began to increase on average, and soon after 1500 iterations we see a very clear illustration of the “mode collapse” problem highlighted earlier in the spikes of the losses of both the generator and the discriminator. This is why the final value for the number of epochs was decided to be 130. Below is a visualization of the network losses on the final run of 130 epochs.

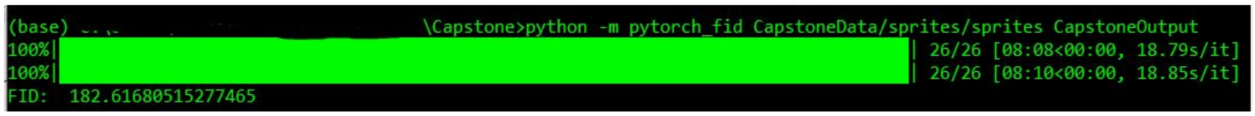


Other unsuccessful attempts at improving results were taken from a slightly outdated github repo [7], although a lot of these “hacks” were already incorporated into the design originally. Interestingly, tip 17: “Use Dropouts in G” did result in lower losses overall for the generator, however the final images were less defined than without, so the dropout layers were removed.

Results

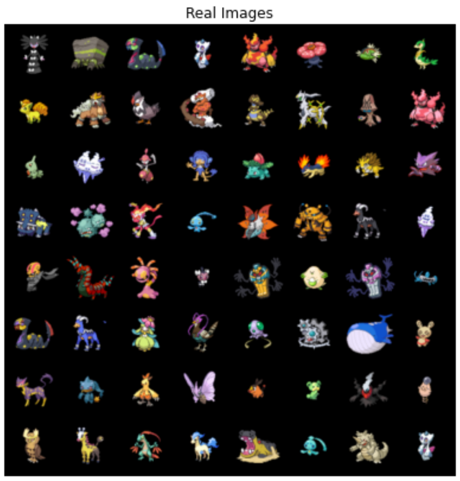
Model Evaluation and Validation.

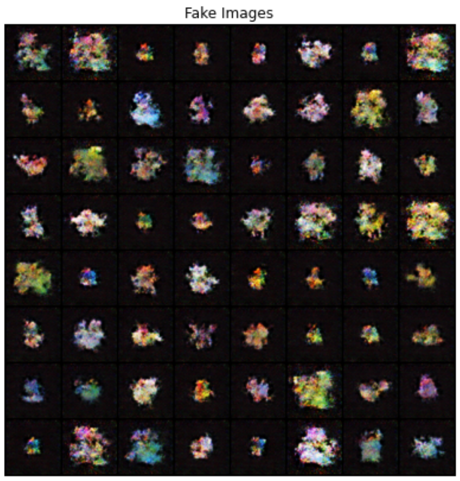
The GAN was used to generate 1298 fake Pokemon in order to create a second dataset to be used in the Pytorch library that measures the FID of two datasets. The resulting FID of this GAN against the original dataset was 182.62.



```
(base) [redacted]\Capstone>python -m pytorch_fid CapstoneData/sprites/sprites CapstoneOutput
100%|██████████████████████████████████████████████████████████| 26/26 [08:08<00:00, 18.79s/it]
100%|██████████████████████████████████████████████████████████| 26/26 [08:10<00:00, 18.85s/it]
FID:  182.61680515277465
```

Shown below is a sampling of 64 real Pokemon compared to images generated by the GAN.





Justification.

Considering the goal of an FID score under 100, the score this GAN achieved can't really qualify this experiment as a "success". Understandably from this score, the overall quality of the generated "Pokemon" is a bit lackluster as well, but the GAN clearly made progress over just generating random noise, and with some refinement and a lot more data, along with the incredible progress being made in the space of generative modeling, GAN's just might produce some full-fledged Pokemon some day. In the meantime, maybe the blurry blobs produced could be used as a jumping-off point for the artists and designers at Nintendo, or at the very least give them a good laugh.

Conclusion

While it's disappointing the generated "Pokemon" probably won't be fooling anyone, it's encouraging to see the GAN clearly learned to imitate the variety of shapes and colors based on such a small dataset. The dataset could be expanded or even changed entirely by scouring the internet for independent artist's renditions of Pokemon, which are plentiful. Or perhaps in the future an ensemble of GAN's could tackle the various components that make up the input, namely the faces, the body shapes, and the colors, and assemble them to achieve more convincing results.

References

[1] Deepfake. https://en.wikipedia.org/wiki/Deepfake.

[2] Pytorch-fid. https://github.com/mseitzer/pytorch-fid.

[3] Pokemon sprite gallery. https://pokemondb.net/sprites.

[4] GAN architecture. https://newtraell.cs.uchicago.edu/files/ms_paper/xksteven.pdf.

[5] PresGAN. https://paperswithcode.com/paper/prescribed-generative-adversarial-networks.

[6] Pytorch DCGAN tutorial. https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html.

[7] GANhacks. https://github.com/soumith/ganhacks.