Implementing Creative Adversarial Networks

Daniel Ahn
University of California, Los Angeles
Department of Computer Science
dahn@g.ucla.edu

Kevin Lee
University of California, Los Angeles
Department of Electrical Engineering
kevin.lee@engineering.ucla.edu

Dawei Huang
University of California, Los Angeles
Department of Computer Science
daweihuang@g.ucla.edu

Abstract

We implement a Creative Adversarial Network (CAN). The CAN architecture synthesizes 'creative' art by maximizing its outputs' deviation from established styles and minimizing its deviation from general art distribution. We validate our architecture on the art dataset WikiArt.

1. Introduction

CAN is based off of Generative Adversarial Networks, introduced by Goodfellow et. all in 2014. [2] GAN is a framework that relies on an adversarial process to estimate data distributions. GAN is composed a generative model G and a discriminative model D. G learns to generate distributions that are so similar to the data distribution that the discriminator can not differentiate G's outputs from the actual training dataset. On the other hand, the discriminator D attempts to keep up with G's outputs and optimize itself to be better at discriminating between the generated distribution and the data distribution. This adversarial process results in the following minimax function that defines the loss function of the Generative Adversarial Network:

$$\min_G \max_D V(D,G) = \mathbf{E}_{\boldsymbol{x}\sim p_{data}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbf{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(z)}[\log(1 - D(G(\boldsymbol{z})))]. \quad (1)$$

There has been many other works utilizing GANs in innovative ways. CycleGAN performs image-to-image translation from two unordered image collections X and Y i.e. changing a zebra into a horse and vice versa. [3] StyleGAN is another recent exciting innovation enabling the addition of features to an image at different scales i.e. changing just the nose or hairstyle of an image of a face. [5]

We implement the CAN architecture designed by Elgammal et al from scratch and train it on different genres. [1] In addition to the vanilla GAN loss function presented in (1) the CAN loss function uses terms to represent the style ambiguity as well as the real or fake quality of an image. The CAN loss function can be seen below with the added style ambiguity term.

$$\begin{aligned}\min_G \max_D V(D,G) = \mathbf{E}_{x,\hat{c}\sim p_{data}(x,\hat{c})}[&\log D_r(x) \\ &+ \log D_c(c = \hat{c}|x)] \\ &+ \mathbf{E}_{z\sim p_z(z)}\Big[\log(1 - D_r(G(z))) \\ &- \Big(\sum_{\hat{c}\in C}\frac{1}{|C|}\log(D_c(\hat{c}|G(z))) \\ &+ (1 - \frac{1}{|C|})\log(1 - D_c(\hat{c}|G(z)))\Big)\Big]\end{aligned} \quad (2)$$

An important feature of the CAN loss function is the addition of $D_c(c|x)$ to the expected value of the real distribution. This helps to regularize the generated image to be as stylistically ambiguous as possible from the artworks in the dataset. The ambiguity is achieved by having the generator maximize (3).

$$\sum_{\hat{c}\in C}\frac{1}{|C|}\log(D_c(\hat{c}|G(z))) + (1 - \frac{1}{|C|})\log(1 - D_c(\hat{c}|G(z))) \quad (3)$$

$$\sum_{x\in X} p(x)\log q(x) \quad (4)$$

(3) is based off of cross entropy (4) in which it measures the information needed to express the ideal distribution p with the estimated distribution q. In our context, we attempt to maximize the loss across all classes by maximizing the number of bits needed to differentiate the output of the generator distribution from the discriminator's learned distributions. This forces the generator to tend toward distributions dissimilar to distributions the discriminator has already learned while constraining it to characteristics learned on real images. By doing so, we were hoping that our GAN model would generate images whose distribution is dissimilar to each art genre's distribution but is similar to the amalgamation of all genre's distribution.

2. Results

We display our results in 4x4 grids in the below figures. After adjusting the learning rates and the discriminator-generator training ratio to 1:2 and sending the output of the hidden layers through a sigmoid function, we obtain tenable results.

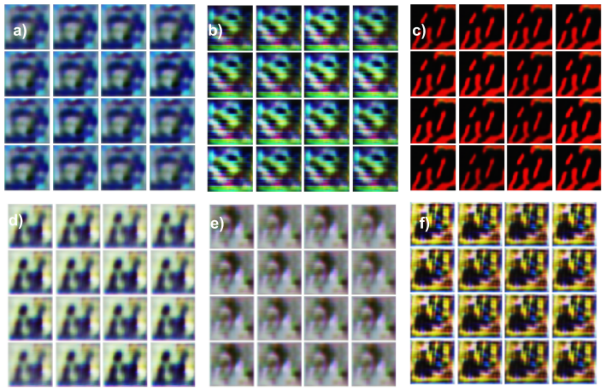

Figure 1. Good generated images. Images were generated at 64x64 resolution and selected for display. Full generated image set here: https://imgur.com/a/eZ4Kmci

We also show some of our failure progressions, marked by their inability to change the output image over many iterations, or sparse output. Notice that some of these failure cases also display over-saturated colors.

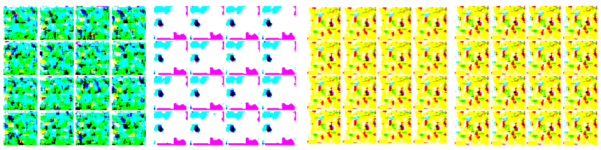

Figure 2. Bad generated images. Images were generated at 64x64 resolution at iteration 0, 300, 4900, 31000, from left to right. In this case the network plateaus, from iteration 4900 to 31000 there are extremely few changes in the generated image.

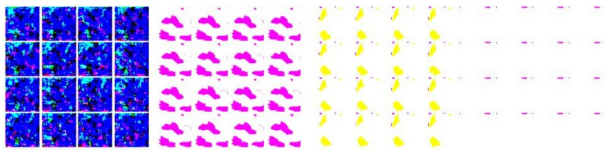

Figure 3. Bad generated images. Images were generated at 64x64 resolution at iteration 0, 300, 1800, 7700, from left to right. Notice that the network learns to generate only sparse images.

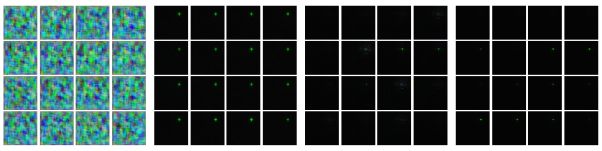

Figure 4. Bad generated images. Images were generated at 64x64 resolution at iteration 0, 10, 2000, 3460, from left to right. In this attempt we scaled by a scalar the output of the hidden layer to the generator's sigmoid function to prevent color saturation. Notice that the network's output is over-saturated, characterized by black surroundings with a small circular impulse.

3. Discussion

Due to Google Colab memory restrictions, we were only able to run our network up to 13,310 iterations. Given a more powerful computing environment, we would be able to train for more iterations at a larger resolution and generate more expressive images.

We initially tried a naive two layer fully connected network for our generator and discriminator model. This model proved to be ineffective and lead to high discriminator loss. Upon inspection, we realized that two fully connected layers lacked the capacity to accurately learn and classify features of our images. This of course lead to poor training for our generator since it was always able to fool the discriminator. This phenomenon is associated with mode collapse as our generator only needs to generate a small diversity of samples to minimize its losses.

To remedy the issue, we upgraded the discriminator's architecture to that of a 5 layer convolutional neural network. This solved the problem of low discriminator capacity and we observed that our discriminator loss decreased. However, we were then met with the new problem of poor generator performance. After the generator had been upgraded to an inverse convolutional neural network, the two layer fully connected generator no longer had the ability to fool the discriminator. This lead to images that seemingly consisted of noise similar to the distribution the initial input to the generator was drawn from. By observing these outputs, we realized we would need to implement some sort of an inverse convolutional generator to match the capacity of the discriminator.

An inverse convolutional generator greatly increased the ability of the network to create coherent images from

random inputs. After adding a 5 layer inverse convolutional neural network to our generator our generated images started to have characteristics closer to that of actual artwork. In the starting phases of training, we could easily see that generated images learned some of the key traits of art. There were shapes resembling faces and bodies in locations typical of a portrait painting which indicated that the network learned to not only draw shapes but also learned their relative placement.

In addition to tuning the architecture, we explored the use of different optimizers and their effects on the generator and discriminator loss. We ran multiple sessions with optimizers such as Adam and RMSProp and compared their results to see which gave the best performance. Ultimately, we decided on the Adam optimizer since it allowed for faster training times with nearly identical results.

Despite the extensive tuning and testing we employed, we found that our implementation still suffered from some serious problems. One major problem of any GAN-based architecture is balancing the discriminator and the generator loss. To ensure proper learning, it is vital that the discriminator and generator remain in an equilibrium, without either gaining dominance. In our case, the discriminator loss quickly converged to zero, preventing the discriminator from properly learning, as in Figure 2. To solve this, we train the generator twice for every discriminator training iteration and increase the generator learning rate relative to the discriminator, allowing the generator to keep pace. Increasing the training ratio above two resulted in the generation of sparse images as in Figure 3.

Additionally, the generator produced images that were over-saturated, with RGB values of either 0 or 255, indicating that the sigmoid input in the generator was saturated. We map the sigmoid input to [-4,4] using a linear mapping to increase the density of points within the center region and adjust the learning rates. This approach works well because it drastically reduces the range of inputs to the sigmoid that get saturated. We use this approach in Figure 1. Incorrect learning rates resulted in dark images as in Figure 4.

Our network suffers from mode collapse, as our generator often generates a only limited diversity of samples. Within the displayed 4x4 image grids, each image is almost identical to the others. One possible cause of mode collapse is that to compensate for a weak generator, we train our generator multiple times, pushing it towards a local minimum with low variance with respect to a noisy input. This corresponds to strengthening the generator disproportionally. One possible solution is to instead weaken the discriminator, allowing a training ratio of 1:1 and hopefully mitigating mode collapse. A few other approaches to solve this are directly encouraging diversity through minibatch discrimination and feature mapping, or using unrolled GANs. [6] [4] We intend to address this issue in future works.

We believe that the approach we have taken to be creative since it required extensive tuning of multiple of generator - discriminator pairs in order to see which offered the highest performance. We implemented several different generator - discriminator architectures ranging from simply connected layers to experimenting with deep convolutional networks.

References

[1] M. E. M. M. Ahmed Elgammal1, Bingchen Liu. Can: Creative adversarial networks generating art by learning about styles and deviating from style norms. *arXiv:1706.07068*, 2017. 1

[2] M. M.-B. X. D. W.-F. S. O. A. C. Y. B. Ian J. Goodfellow, Jean Pouget-Abadie. Generative adversarial networks. *arXiv:1406.2661*, 2014. 1

[3] P. I. A. A. E. Jun-Yan Zhu, Taesung Park. Unpaired image-to-image translation using cycle-consistent adversarial networks. *arXiv:1703.10593*, 2017. 1

[4] D. P. J. S.-D. Luke Metz, Ben Poole. Unrolled generative adversarial networks. *arXiv:1611.02163*, 2017. 3

[5] T. A. Tero Karras, Samuli Laine. A style-based generator architecture for generative adversarial networks. *arXiv:1812.04948*, 2018. 1

[6] W. Z. V. C.-A. R. X. C. Tim Salimans, Ian Goodfellow. Improved techniques for training gans. *arXiv:1606.03498*, 2016. 3

Daniel Ahn - 504817439
Dawei Huang - 304792166
Kevin Lee - 604815473

GAN architecture:

Discriminator:

Our GAN architecture is a type of neural networks composed of a generator and a discriminator. The discriminator is a CNN. This means that the CNN takes in an image and each convolutional layer of the CNN is composed of a convolution layer, a leaky relu layer, and a average pooling layer. The convolution layer has multiple filters that perform convolutions onto the image, reducing its height and width and increasing its depth. Then the resulting output is activated under a leaky relu activation function and placed under an average pooling layer. The average pooling layer partitions the output of the leaky relu into 2x2x1 slices and average out the slices' elements in the final output.

The reason we wanted to do a leaky relu layer instead of a relu layer is that we faced a problem when training that we later realized is a dying relu problem. Due to the nature of a relu function, the dying relu problem is most of the weights of the output neurons goes into zeroes, preventing any effective gradients and hence training. So we improvise by using a leaky relu function instead so that there would still be some gradients when the input value is less than zero. In addition, we use an average pooling layer instead of a max pooling layer common in traditional CNN is that average pooling preserve information of all the elements covered by the average pooling filter.

The discriminator is composed of 3 convolutional layers. The shapes of each outputs end up like this:

(64, 64, 3) → (32, 32, 32) → (16, 16, 64) → (8, 8, 128)

At the last layer, in which the image has been convoluted into a very small, very deep layer of shape (8, 8, 128), the CNN flattens the output and put the output as an input into a fully connected layer. But what we did differently than traditional CNN discriminators common in GAN is that instead of having one fully connected layer, we ended up having 2 fully connected layer going in parallel like this:

(8, 8, 128) → (8*8*128,) → (1024,) → (7,)
→ (1024,) → (1,)

The first fully connected layer is a 7-dimensional vector that represents the logits of the art genres the discriminator thinks the image is supposed to be classified as. The second fc layer represents the logits of the distribution the discriminator thinks the image belongs to (real distribution or generated/fake distribution). These two fully-connected layers could represent the heads of a classification discriminator and a real-art discriminator, respectively. The sigmoid of the first logits represents the probability $D_c (c \mid x)$; this is the probability the discriminator will classify the input (64, 64, 3) tensor as class c or art genre c. The sigmoid of the second logits represents the probability $D_r (x)$; this is the probability the discriminator classify the input (64, 64, 3) tensor as the dataset distribution rather than a generative distribution. The equilibrium will be reached when $D_r(x) = 0.5$ for all x because that means that the discriminator won't be able to differentiate the difference between the generative distribution and the dataset distribution. In colloquial terms, this is when the discriminator won't be able to tell a generated image apart from an image from the dataset.

Generator:

The generator is a neural net that takes in z, a normally-sampled 100-dimensional vector, and send the vector z through two fully connected layers in order so that the features can be propagated to locations in the vector that will allow for the creation of a coherent image. Then the output vector is reshaped into a small, very deep tensor. The tensor goes through a leaky relu and a transposed convolution layer. A transposed convolution layer can be thought of as a convolution with fractional slides. The transposed convolution layer is helpful because it upsample our small, deep input layer into a wider, more shallow output layer. The tensor goes through the leaky relu and transposed convolution layer three times total, ending up in the expected dimensions of an input image. The following graph elucidate the generative process:

Z.shape = (100,) → (1024,) → (8*8*128,) → (8, 8, 128) → (16, 16, 64) → (32, 32, 32) → (64, 64, 3)

In the end, we linearly map the output of weights of generative image to [-4, 4] range and sigmoid the logit output. Furthermore discussion about this portion is mentioned in the report

Training

We train on Google Colab using GPUs. To train our architecture we use 18,458 images from the WikiArt dataset collected from 7 genres: cubism, early renaissance, fauvism, pointillism, pop art, realism, and ukiyo-e. We do not perform data augmentation because our dataset size is sufficient. To speed up computation, we first resize our images to 64x64 and normalize to [-1,1]. We use an Adam optimizer and generator learning rate of 1e-3 and discriminator learning rate of 1e-4. For each training iteration, we take a uniform random minibatch of 200 images from the dataset to train on. We train the style discriminator on a minibatch of artworks we want to train away from and train the real/fake discriminator on a minibatch that characterizes the “realness” of an image. Following the discriminator training, we train the generator twice and save the outputs every 10 iterations