1 Generative Adversarial Network Background

Generative Adversarial Networks are composed of two parts; a discriminator and a generator. These can be any sort of feedforward neural network, but in the simplest case they are just Multilayer Perceptrons. StyleGAN uses Convolutional Neural Networks instead. These two networks have opposing aims. In the most general sense, the goal of the discriminator is to accurately distinguish between random variables generated by the generator and random variables from some real distribution (that is likely not parametrizable). The goal of the generator is to produce random variables that the discriminator can not distinguish from a real probability distribution. This is guaranteed to be possible by the Universal Approximation Theorem.

In the case of square images of size n, our probability distribution would be in dimension $n \times n$, and the specific distribution corresponding to say 'dog' images would be just one distribution in this massive vector space $\mathbb{R}^{n \times n}$. The generator is tasked with generating random variables (which can be reshaped into images) that are so close to this 'dog' distribution that the discriminator cannot tell them apart.

2 Model Parameters

To actually go about training such a network, we need to define 2 feedforward networks that can be trained with standard backpropagation. We will call them

$$G : \mathcal{N}(0,1) \to X_g \text{ and } D : X_t \cup X_g \to [0,1]$$

Here $\mathcal{N}(0,1)$ is the normal distribution with zero mean and unit variance, and X_t and X_g are the true and generated distributions respectively. The generator turns a random variable into some RV that follows the more complex distribution X_g, and the discriminator can take some RV from either the true or generated distributions and decide if it is real with some probability in $[0,1]$. We also need to define a loss function in such a way that both networks can optimize the same loss, since the networks will be trained together. Our objective is best defined as follows (where p_t and p_g are the densities of X_t and X_g respectively)

$$\begin{aligned}\min_G \max_D V(G,D) &= \mathbb{E}_{x \sim p_t(x)}[\log D(x)] + \mathbb{E}_{z \sim \mathcal{N}(0,1)}[\log(1 - D(G(z)))] \\ &= \mathbb{E}_{x \sim p_t(x)}[\log D(x)] + \mathbb{E}_{x \sim p_g(x)}[\log(1 - D(x))]\end{aligned}$$

The second way of writing holds because of the definition of G. This is a very concise way of writing what we want from G and D mathematically. The second expression shows that the D that will maximize this loss will output something close to 0 for $x \sim p_g(x)$ (since this is generated data) but something close to 1 for $x \sim p_t(x)$ (since this is real data). This is exactly what we want from D. Similarly the first expression shows that the G that will minimize the loss will output something that D classifies as an image, since 1 is the maximal value for the only term that involves G. This is exactly what we want from our G. Thus the loss is measuring what we want it to. We can even go so far as to say that the optimal loss occurs when $p_g = p_t$ since that is when V is at its minimum for a given D. This loss function coupled with the weights and biases of G and D are all we need to train the network.

The actual training algorithm can take any form, though of course backpropagation would be the most familiar. We need to optimize the parameters of two networks at the same time, and this can be done by performing gradient ascent on the discriminator many times using both variables sampled from p_t and generated variables sampled from p_g within each epoch while only performing gradient descent on the generator once in each epoch by computing the gradient using RV's from the generated distribution. Of course techniques like Stochastic Gradient Descent, Momentum or Adam could also improve convergence while also improving efficiency depending on the size of the architecture.

3 A Real Example

A practical example can be found at this github repo.

G and D are a very simple networks with only 1 layer and 4 nodes, but they can still learn to generate 2×2 images. With the practical example it also becomes clear that the loss criterion as given, while theoretically correct is easier to optimize against when it is split up. Thus the loss function for G only involves 1 term whereas the loss function for D is split up into a function that takes inputs from p_t and one that takes inputs from p_g. The weights are also updated manually in each step, with each getting updated once per epoch. Running through the notebook can give some practical insight into how Generative Adversarial Networks work.

4 References

[1] Bengio et al. *Generative Adversarial Networks.* arXiv:1406.2661

[2] Joseph Rocca *Understanding Generative Adversarial Networks (GANs)* Article

[3] Luis Serrano `Generative Adversarial Networks` Github