DCGAN

A Preprint

Abstract

We propose and evaluate a set of constraints on the architectural topology of Convolutional GANs that make them stable to train in most settings. We name this class of architectures Deep Convolutional GANs (DCGAN) • We use the trained discriminators for image classification tasks, showing competitive performance with other unsupervised algorithms. • We visualize the filters learnt by GANs and empirically show that specific filters have learned to draw specific objects.

1 Main

Generative models fall in two categories: parametric and non-parametric.

Example of non parametric: do matching from a database of existing images, often matching patches of images, and have been used in texture synthesis (Efros et al., 1999), super-resolution (Freeman et al., 2002) and in-painting (Hays Efros, 2007).

Coming to parametric there wasnt much success until 2015. Laplacian Pyramid: In image processing, a Laplacian pyramid is a sequence of images. The lowest level is a blurred version of the original image. Each subsequent level stores the difference (like an edge map) between the image at the previous level and a further blurred version of that image. This allows the image to be reconstructed by starting with the most blurred version and progressively adding the details from each level of the pyramid. It essentially decomposes the image into different bands of image frequencies.

Laplacian Pyramid Extension (LAPGANs): The approach by Denton et al. (2015) uses a cascade of GANs, each corresponding to a different level of a Laplacian pyramid.

It starts by generating a very low-resolution image at the coarsest level of the pyramid using a GAN. Then, at each subsequent level, another GAN is trained to generate the high-frequency details (the difference image) for that level, conditioned on the upscaled output from the previous, coarser level. By adding these generated details at each step, the model progressively refines the image, building it up from a coarse outline to a higher-resolution version.

Architecture guidelines for stable Deep Convolutional GANs • Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator). • Use batchnorm in both the generator and the discriminator. • Remove fully connected hidden layers for deeper architectures. • Use ReLU activation in generator for all layers except for the output, which uses Tanh. • Use LeakyReLU activation in the discriminator for all layers

2 Architecture

The first is the all convolutional net (Springenberg et al., 2014) which replaces deterministic spatial pooling functions (such as maxpooling) with strided convolutions, allowing the network to learn its own spatial downsampling. We use this approach in our generator, allowing it to learn its own spatial upsampling, and discriminator. Second is the trend towards eliminating fully connected layers on top of convolutional features. The strongest example of this is global average pooling which has been utilized in state of the art image classification models (Mordvintsev et al.). We found global average pooling increased model stability but hurt convergence speed. A middle ground of directly connecting the highest convolutional features to the input and output respectively of the generator and discriminator worked well. The first layer of the GAN, which takes a uniform noise distribution Z as input, could be called fully connected as it is just a matrix multiplication, but the result is reshaped into a 4-dimensional tensor and used as the start of the convolution stack. For the discriminator, the last convolution layer is flattened and then fed into a single sigmoid output. See Fig. 1 for a visualization of an example model architecture. Third is Batch Normalization (Ioffe Szegedy, 2015) which stabilizes learning by normalizing the input to each unit to have zero mean and unit variance. This helps deal with training problems that arise due to poor initialization and helps gradient flow in deeper models. This proved critical to get deep generators to begin learning, preventing the generator from collapsing all samples to a single point which is a common failure mode observed in GANs. Directly applying batchnorm to all layers however, resulted in sample oscillation and model instability. This was avoided by not applying batchnorm to the generator output layer and the discriminator input layer. The ReLU activation (Nair Hinton, 2010) is used in the generator with the exception of the output layer which uses the Tanh function. We observed that using a bounded activation allowed the model to learn more quickly to saturate and cover the color space of the training distribution. Within the discriminator we found the leaky rectified activation (Maas et al., 2013) (Xu et al., 2015) to work well, especially for higher resolution modeling. This is in contrast to the original GAN paper, which used the maxout activation (Goodfellow et al., 2013).

3 Details

Training was completed on 3 different datasets:

- Imagenet-1k
- Faces
- LSUN

. No pre-processing was applied to training images besides scaling to the range of the tanh activation function [-1, 1]. All models were trained with mini-batch stochastic gradient descent (SGD) with a mini-batch size of 128. Weights were initialized from $N(0; 0.02)$. While previous GAN work has used momentum to accelerate training, we used the Adam optimizer (Kingma Ba, 2014) with tuned hyperparameters. We found the suggested learning rate of 0.001, to be too high, using 0.0002 instead.

Let s simply describe how it works step by step. First of all Input noise generator: 100D vector z is sampled from U. Think of z as a random seed or a point in a "latent space"(a hidden space of representations).

Then Project and Reshape. 100D first projected into a higher-dimensional space. The paper mentions this is a matrix multiplication that results in a representation with many feature maps. This flat representation is then reshaped into a 4-dimensional tensor that can be processed by convolutional layers. According to the diagram, this initial tensor has: Spatial dimensions: 4x4 pixels. Number of feature maps (channels): 1024. So, the 100D vector z is transformed into a 4x4x1024 tensor.

Then CONV1. Fractionally-Strided Convolution with input 4x4x1024 tensor and output 8x8x512

CONV 2 (Fractionally-Strided Convolution):

This layer takes the output from CONV 1 and performs another fractionally-strided convolution. Input: The 8x8x512 tensor (approximately). Output: The number of feature maps is reduced to 256, and the spatial dimensions are again doubled. This would result in a 16x16x256 tensor. CONV 3 (Fractionally-Strided Convolution):

The process continues. Input: The 16x16x256 tensor (approximately). Output: The number of feature maps is reduced to 128, and spatial dimensions are doubled, yielding a 32x32x128 tensor. CONV 4 (Fractionally-Strided Convolution):

This is the final convolutional layer in the generator. Input: The 32x32x128 tensor (approximately). Output: This layer produces the final output image. The number of feature maps is reduced to 3, which corresponds to the three color channels (Red, Green, Blue) of a color image. The spatial dimensions are doubled one last time to 64x64. So, the output is a 64x64x3 tensor, representing a 64x64 pixel color image. Output Image G(z):

The 64x64x3 tensor from CONV 4 is the generated image, denoted as G(z).