DCGAN Tutorial

References

Original tutorial from PyTorch
Author of the original tutorial: Nathan Inkawhich

Note: This tutorial has been adapted to work with images of size `32x32` instead of `64x64` for faster training.

Dataset

This experiment uses the flowers dataset from the competition. A downscaled version of the images are provided in folder `dataset_flowers_64px`. We provide `64x64` images in case you want to modify the training to create larger images or want to modify the dataloader in some way. By default, the images will be downsampled to `32x32` during training.

The original tutorial uses the Celeb-A Faces Dataset.

Introduction

This tutorial will give an introduction to DCGANs through an example. We will train a generative adversarial network (GAN) to generate new celebrities after showing it pictures of many real celebrities. Most of the code here is from the DCGan implementation in pytorch/examples and this document will give a thorough explanation of the implementation and shed light on how and why this model works. But don't worry, no prior knowledge of GANs is required, but it may require a first-timer to spend some time reasoning about what is actually happening under the hood. Also, for the sake of time it will help to have a GPU. Lets start from the beginning.

What is a GAN?

GANs are a framework for teaching a DL model to capture the training data's distribution so we can generate new data from that same distribution. GANs were invented by Ian Goodfellow in 2014 and first described in the paper Generative Adversarial Nets. They are made of two distinct models, a *generator* and a *discriminator*. The job of the generator is to spawn 'fake' images that look like the training images. The job of the discriminator is to look at an image and output whether or not it is a real training image or a fake image from the generator. During training, the generator is constantly trying to outsmart the discriminator by generating better and better fakes, while the discriminator is working to become a better detective and correctly classify the real and fake images. The equilibrium of this game is when the generator is generating perfect fakes that look as if they came directly from the training data, and the discriminator is left to always guess at 50% confidence that the generator output is real or fake.

Now, lets define some notation to be used throughout tutorial starting with the discriminator. Let x be data representing an image. $D(x)$ is the discriminator network which outputs the (scalar) probability that x came from training data rather than the generator. Here, since we are dealing with images the input to $D(x)$ is an image of CHW size 3x32x32. Intuitively, $D(x)$ should be HIGH when x comes from training data and LOW when x comes from the generator. $D(x)$ can also be thought of as a traditional binary classifier.

For the generator's notation, let z be a latent space vector sampled from a standard normal distribution. $G(z)$ represents the generator function which maps the latent vector z to data-space. The goal of G is to estimate the distribution that the training data comes from (p_{data}) so it can generate fake samples from that estimated distribution (p_g).

So, $D(G(z))$ is the probability (scalar) that the output of the generator G is a real image. As described in Goodfellow's paper D and G play a minimax game in which D tries to maximize the probability it correctly classifies reals and fakes ($logD(x)$), and G tries to minimize the probability that D will predict its outputs are fake ($\log(1 - D(G(x)))$). From the paper, the GAN loss function is

$$\underset{G}{\min}\underset{D}{\max} V(D, G) = \mathbb{E}_{x\sim p_{data}(x)}\big[logD(x)\big] + \mathbb{E}_{z\sim p_z(z)}\big[log(1 - D(G(z)))\big] \tag{1}$$

In theory, the solution to this minimax game is where $p_g = p_{data}$, and the discriminator guesses randomly if the inputs are real or fake. However, the convergence theory of GANs is still being actively researched and in reality models do not always train to this point.

What is a DCGAN?

A DCGAN is a direct extension of the GAN described above, except that it explicitly uses convolutional and convolutional-transpose layers in the discriminator and generator, respectively. It was first described by Radford et. al. in the paper Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks The discriminator is made up of strided convolution layers, batch norm, LeakyReLU activations. The input is a 3x64x64 input image and the output is a scalar probability that the input is from the real data distribution. The generator is comprised of convolutional-transpose layers, batch norm layers, and ReLU activations. The input is a latent vector, z, that is drawn from a standard normal distribution and the output is a 3x64x64 RGB image. The strided conv-transpose layers allow the latent vector to be transformed into a volume with the same shape as an image. In the paper, the authors also give some tips about how to setup the optimizers, how to calculate the loss functions, and how to initialize the model weights, all of which will be explained in the coming sections.

Inputs

Let's define some inputs for the run:

- `batch_size`: The batch size used in training. The DCGAN paper uses a batch size of 128
- `image_size`: The spatial size of the images used for training. This implementation defaults to `32x32`. If another size is desired, the structures of D and G must be changed. See here for more details.
- `nc`: The number of color channels in the input images. For color images this is 3.
- `nz`: The length of latent vector.
- `ngf`: This sets the depth of feature maps in the generator.
- `ndf`: This sets the depth of feature maps in the discriminator.
- `num_epochs`: The number of training epochs to run. Training for longer will probably lead to better results but will also take much longer.
- `lr`: The learning rate for training. As described in the DCGAN paper, this number should be `0.0002`.
- `beta1`: The beta1 hyperparameter for the two Adam optimizers. As described in paper, this number should be `0.5`.

Implementation

With our input parameters set and the dataset prepared, we can now get into the implementation. We will start with the weigth initialization strategy, then talk about the generator, discriminator, loss functions, and training loop in detail.

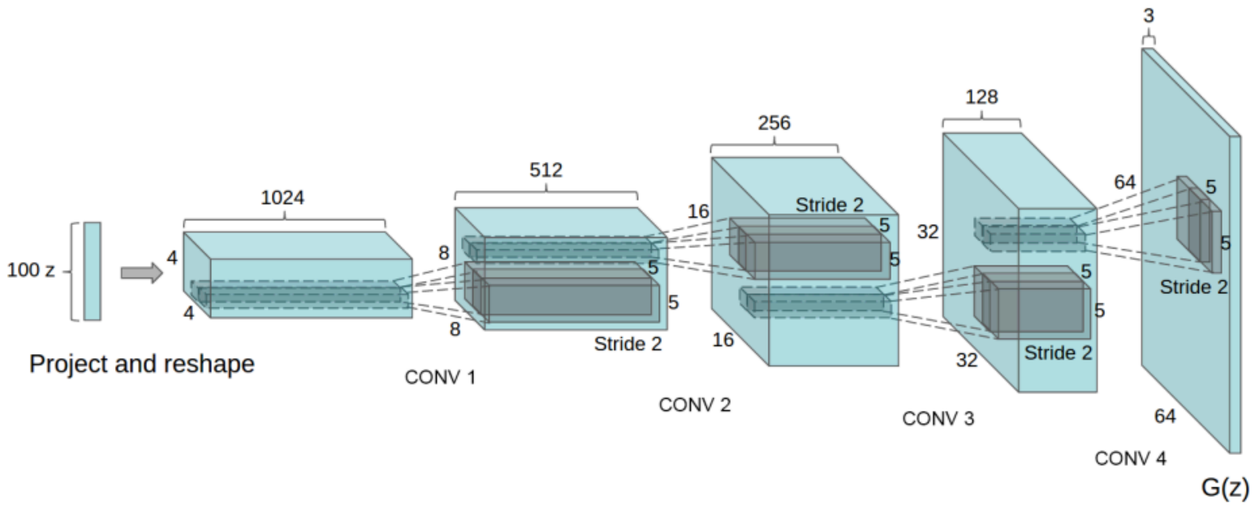
Weight Initialization

From the DCGAN paper, the authors specify that all model weights shall be randomly initialized from a Normal distribution with `mean=0, stdev=0.02`. The `weights_init` function takes an initialized model as input and reinitializes all convolutional, convolutional-transpose, and batch normalization layers to meet this criteria. This function is applied to the models immediately after initialization.

Generator

The generator, G, is designed to map the latent space vector (z) to data-space. Since our data are images, converting z to data-space means ultimately creating a RGB image with the same size as the training images (i.e. `3x32x32`). In practice, this is accomplished through a series of strided two dimensional convolutional transpose layers, each paired with a 2d batch norm layer and a ReLU activation. The output of the generator is fed through a tanh function to return it to the input data range of $[-1, 1]$. It is worth noting the existence of the batch norm functions after the conv-transpose layers, as this is a critical contribution of the DCGAN paper. These layers help with the flow of gradients during training. An image of the generator from the DCGAN paper is shown below.

Note: As we reduced the image size from `64x64` to `32x32`, we change the last layer and use a regular convolution instead of a transposed convolution.



Notice, the how the inputs we set in the input section (`nz`, `ngf` and `nc`) influence the generator architecture in code. `nz` is the length of the input vector z, `ngf` relates to the size of the feature maps that are propagated through the generator, and `nc` is the number of channels in the output image (set to `3` for RGB images). Below is the code for the generator.

Now, we can instantiate the generator and apply the `weights_init` function. Check out the printed model to see how the generator object is structured.

Discriminator

As mentioned, the discriminator, D, is a binary classification network that takes an image as input and outputs a scalar probability that the input image is real (as opposed to fake). Here, D takes a `3x32x32` input image, processes it through a series of `Conv2d`, `BatchNorm2d`, and `LeakyReLU` layers, and outputs the final probability through a `Sigmoid` activation function. This architecture can be extended with more layers if necessary for the problem, but there is significance to the use of the strided convolution, `BatchNorm`, and `LeakyReLU`s. The DCGAN paper mentions it is a good practice to use strided convolution rather than pooling to downsample because it lets the network learn its own pooling function. Also batch norm and leaky relu functions promote healthy gradient flow which is critical for the learning process of both G and D.

Now, as with the generator, we can create the discriminator, apply the `weights_init` function, and print the model's structure.

Loss Functions and Optimizers

With D and G setup, we can specify how they learn through the loss functions and optimizers. We will use the Binary Cross Entropy loss function which is defined in PyTorch as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -\left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)\right] \tag{2}$$

Notice how this function provides the calculation of both log components in the objective function (i.e. $\log(D(x))$ and $\log(1 - D(G(z)))$). We can specify what part of the BCE equation to use with the y input. This is accomplished in the training loop which is coming up soon, but it is important to understand how we can choose which component we wish to calculate just by changing y (i.e. GT labels).

Next, we define our real label as `1` and the fake label as `0`. These labels will be used when calculating the losses of D and G, and this is also the convention used in the original GAN paper. Finally, we set up two separate optimizers, one for D and one for G. As specified in the DCGAN paper, both are Adam optimizers with learning rate `0.0002` and Beta1 `0.5`. For keeping track of the generator's learning progression, we will generate a fixed batch of latent vectors that are drawn from a Gaussian distribution (i.e. `fixed_noise`) . In the training loop, we will periodically input this `fixed_noise` into G, and over the iterations we will see images form out of the noise.

Training

Finally, now that we have all of the parts of the GAN framework defined, we can train it. Be mindful that training GANs is somewhat of an art form, as incorrect hyperparameter settings lead to mode collapse with little explanation of what went wrong. Here, we will closely follow Algorithm 1 from Goodfellow's paper, while abiding by some of the best practices shown in ganhacks.

Namely, we will ”construct different mini-batches for real and fake” images, and also adjust G's objective function to maximize $\log\ D(G(z))$. Training is split up into two main parts. Part 1 updates the Discriminator and Part 2 updates the Generator.

Part 1 - Train the Discriminator

Recall, the goal of training the discriminator is to maximize the probability of correctly classifying a given input as real or fake. In terms of Goodfellow, we wish to ”update the discriminator by ascending its stochastic gradient”. Practically, we want to maximize $\log(D(x)) + \log(1 - D(G(z)))$. Due to the separate mini-batch suggestion from ganhacks, we will calculate this in two steps. First, we will construct a batch of real samples from the training set, forward pass through D, calculate the loss ($\log(D(x))$), then calculate the gradients in a backward pass. Secondly, we will construct a batch of fake samples with the current generator, forward pass this batch through D, calculate the loss ($\log(1 - D(G(z)))$), and *accumulate* the gradients with a backward pass. Now, with the gradients accumulated from both the all-real and all-fake batches, we call a step of the Discriminator's optimizer.

Part 2 - Train the Generator

As stated in the original paper, we want to train the Generator by minimizing $\log(1 - D(G(z)))$ in an effort to generate better fakes. As mentioned, this was shown by Goodfellow not to provide sufficient gradients, especially early in the learning process. As a fix, we instead wish to maximize $\log(D(G(z)))$. In the code we accomplish this by: classifying the Generator output from Part 1 with the Discriminator, computing G's loss *using real labels as GT*, computing G's gradients in a backward pass, and finally updating G's parameters with an optimizer step. It may seem counter-intuitive to use the real labels as GT labels for the loss function, but this allows us to use the $\log(x)$ part of the BCELoss (rather than the $\log(1 - x)$ part) which is exactly what we want.

Finally, we will do some statistic reporting and at the end of each epoch we will push our `fixed_noise` batch through the generator to visually track the progress of G's training. The training statistics reported are:

- `Loss_D`: Discriminator loss calculated as the sum of losses for the all real and all fake batches $(\log(D(x)) + \log(D(G(z))))$.
- `Loss_G`: Generator loss calculated as $\log(D(G(z)))$
- `D(x)`: The average output (across the batch) of the discriminator for the all real batch. This should start close to `1` then theoretically converge to `0.5` when G gets better. Think about why this is.
- `D(G(z))`: Average discriminator outputs for the all fake batch. The first number is before D is updated and the second number is after D is updated. These numbers should start near `0` and converge to `0.5` as G gets better. Think about why this is. Note: This step might take a while, depending on how many epochs you run and if you removed some data from the dataset.