Project 4: Generative Adversarial Networks

CS231n, Deep Learning For Computer Vision
Aryaman Pandya

Section 1: Objective

The objective of this project is to gain a better understanding of:

1. The concepts involved with GANs- minimax game, sampling from a prior, generator, discriminator.
2. Architectural design of the networks used in GANs in the context of computer vision
3. Semi-supervised learning aided by GANs- transfer learning applied to the trained discriminator network
4. Libraries for GANs supported by PyTorch - what's out there, which ones work for which kinds of applications
5. Hyperparameter tuning best practices for GANs
6. The computational expense of training and testing when working with GANs

Section 2: Picking a network architecture

The scope of applications is limited to Computer Vision tasks, which limits the types of GAN architectures that can be used. Two of the main options (sticking to the theme described on https://github.com/aryamanpandya99/ComputerVision) are:

1. The architecture described by Goodfellow et. al. 2014, and run that on a low resolution dataset that's easy to train on
2. Architecture described by **DCGANS** Chintala et. al. 2016 and run that on a slightly higher resolution dataset
    a. This does not have to be limited by knowledge of 2016, and can be supplemented with newer learnings in the field

Item 1 works as a better option if the goal is to iterate quickly and learn things from a high-level, but option 2 would be more challenging and closer to SotA.
Going ahead with option **2**.

Section 3: Training procedure

Good to note that all training procedures are built off of the general algorithm specified in Goodfellow's paper:

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

end for

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Section 3: Picking a dataset

This one is a little bit more important as well as tricky to pick. Need to figure out technically:

1. What kind of resolution images can the architecture I pick handle?
    a. Original DCGAN paper uses 64x64x3 images, with a fairly large dataset, based on this, my 3 options are:
        i. Pick a dataset with somewhat similar dimensionality and resize if need be and obtain cool results
        ii. Pick a dataset with higher resolution/dimensionality and crop to a size that's feasible - this may make the generated images not look so great, but would be a more fun challenge to work on
        iii. Find a network architecture that can handle larger images, and thoroughly research methods to make sure RAM is under control and not overflowing
2. What's the size of the dataset? I'm going to be restricted by the storage restrictions on Colab (if I decide to progress with Colab, that is) and memory restrictions regardless of whether I decide to go local or remote. Need to figure out what those restrictions look like and find something that fits.
3. How applicable would the content being learned from the dataset be to the task of image classification via transfer learning? - think of the transfer learning dataset (rough idea) as well

Beyond these, there are some subjective questions to ask myself as well:

1. What is the type of quality I'm expecting my generated images to have?
    a. Considering it's my first GAN project, I think it's fair to have a focus on the technical concepts and not be tooo worried about the quality of generations- of course this doesn't mean I should be happy with a model that suffers from mode collapse, or one that spits out garbage. The output should at least look like there was some semantic understanding of the dataset but it's fine with some coarseness
2. What kinds of images do I want to work with? Continue the street/AV perspective theme? Or try something else that's more fun?
    a. Eh, doesn't matter. A street one would be fine but less diverse. Doesn't matter, figure it out lol.

Options:

1. BIRDS 450 SPECIES- IMAGE CLASSIFICATION
    a. 70,626 training images, 22500 test images(5 images per species) and 2250 validation images
    b. https://www.kaggle.com/datasets/gpiosenka/100-bird-species?resource=download
    c. Bird_dataset.zip downloaded and saved locally
    d. Good because the resolution isn't ridiculously high and the dataset provides 450 classes which will make for a good transfer learning spin-off
2. Anime Faces - someone already has a popular GAN notebook for this out there
    a. 21551 images

Section 4: Techniques to use

For this section, I think it's best to look at a bunch of different projects and the methods they use, then list them here and pick which ones I like best.

Case Study 1: https://www.run.ai/guides/deep-learning-for-computer-vision/pytorch-gan

This one's a basic MLP v/s MLP, but the techniques could spill over:

1. Uses ADAM as an optimizer (like everything else, but good to know it works well for GANs too)
    a. lr : 1e-3
2. *Binary* Cross-Entropy Loss since discriminator's decision is real/fake

3. Generate noise prior, proceed to **train**:
    a. Zero_grad on optimizer for generator
    b. Generate noisy data by generator(noise)
    c. Run discriminator on generated data
    d. Calculate loss as if generated data was real
    e. gen_loss.backwards(), generator_optimizer.step() - Generator training iteration complete
    f. Zero_grad on optimizer for discriminator
    g. Run discriminator on training data (true data)
    h. Calculate loss v/s true labels (true across the board)
    i. dis_loss.backwards & dis_optimizer.step() - Discriminator training iteration complete

Case Study 2: https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

This one's a beginner level project implementing DCGANs on a small faces dataset, so more applicable to this project. Learnings:

1. Input size: 64x64x3 (good to keep in mind for dataset picking) from the celeba dataset (202,599 images) Zipped at ~1.13 gigs (consider for storage/memory constraints)
2. ConvNet (Discriminator) v/s Backwards strided ConvNet (generator)
3. Discriminator
    a. 5 2D Convolutional layers
    b. Leaky ReLU activation
    c. Batch normalization layers
    d. Output is a binary 1 (input is real) or 0 (input is generated) so the output from the last conv layer is fed through a sigmoid function
2. Generator
    a. 5 Back strided convs
    b. Batch normalization layers post convs
    c. Vanilla ReLU
    d. Input - latent vector z from a gaussian prior
    e. Tanh activation on the output layer
    f. Output: 64x64x3 image
3. batch_size = 128, lr = 2e-3
4. Random weight initialization

5. Binary Cross Entropy Loss - used for both the discriminator and the generator, adjusted using the formula specified in the paper
6. ADAM for both networks
7. `fixed_noise = torch.randn(64, nz, 1, 1, device=device)`
8. Training Procedure:
    a. Incorrect Hyperparameter selection can lead to mode collapse (got to be careful)
    b. Construct different mini-batches for real and fake images, and adjust the generator's objective function to maximize $logD(G(z))$
    c. Training the Discriminator (start with zero grad for netD not netGo see how goo):
        i. Goal is to maximize probability of correct classification we want to maximize log(D(x)) + log(1-D(G(z)))
        ii. Since we have separate mini-batches,
        iii. First forward pass real samples through D, calculate loss log(D(x)) and calculate grad in back pass
        iv. Next, construct batch of fake images using G, forward pass through D, and compute loss ($log(1-D(G(z)))$) this time however, we accumulate gradients during the backward pass
        v. Step through discriminator's optimizer
    d. Training the Generator (start with zero grad):
        i. Training the Generator is a little bit like training the inverse of the discriminator
        ii. We want to maximize log(D(z)) - classify output from G as D(G(z)), compute G's loss using 1 (true label) for all samples
        iii. Compute gradients in a backward pass
        iv. Optimizer step

Tips & tricks for smoother training by Soumith Chintala: https://github.com/soumith/ganhacks

1. Normalize image inputs to be between -1 and 1
2. Tanh as last layer of generator output
3. Flip labels when training generator: real = fake, fake = real
4. Can use stability tricks from RL- experience replay
5. Use ADAM
    a. SGD for discriminator and ADAM for generator

6. If labels are available, train the discriminator to also classify the samples- look at auxiliary GANs
7. Add some artificial noise to inputs to D
8. Add some Gaussian noise to every layer of the generator
9. Use dropouts in G in both train and test phases

DEBUGGING GANs: https://developers.google.com/machine-learning/gan/problems

Section 5: Virtual Machine GPU (Colab) v/s host GPU

This decision isn't project specific but one I need to confront anyway for applicability to future work. So far, all projects (P1-3) have been implemented using GPUs available through colab pro. This is convenient because I don't need to worry too much about maintaining module versions, configuring drivers etc. However, I've been a little agnostic to what the GPU's actual capabilities are:

1. What's the RAM available on each machine?
    a. Host: 8 gigs
    b. Remote: 12gigs (regular), 25 gigs (High-RAM)
2. What kind of NViDIA GPUs am I working with on each machine?
    a. Host: GeoForce RTX 3070 Graphics clock frequency 2100 MHz, memory transfer rate 11000 MHz
    b. Remote: NViDIA Tesla 4 Graphics Clock Frequency 1590 MHz, memory transfer rate 1250 MHz
3. What's more important for my purposes, RAM or compute speed?
4. Do I want to commit to continuing to write notebooks? Or do I want to venture out and start building more holistic code?

– The unfortunate conclusion of this section (for now) is that the NVIDIA driver on my laptop is messed up (thanks to bad treatment by me) For P4, I'm going to resort to using the Colab provided GPUs (slower, sad)