CS 763/CS 764: Lab 08

Generative Adversarial Networks

- Announced Apr 13. Due Apr 21 9AM

1 Introduction

A big problem with deep learning is labeled data. Generative Adversarial Networks (GANs) constitute an important toolbox for the computer vision student since GANs are able to produce output with a specified distribution, at least in theory. GANs, as a side effect, can end up generating amazing pictures for many other purposes unrelated to supervised learning. In this lab, we will be training generative models using GANs.
In brief, GANs are made up of two components: generators and discriminators. We can think of the generator as a forger, trying to generate fake coins, and the discriminator as law enforcers, trying to stop the circulation of fake coins. In order to generate realistic images, our goal (in the end) is to let the forger succeed in fooling the law enforcers. That is, in the ideal situation, the accuracy of discriminator will be 50% on both the fake and real images (or in our analogy, coins). See this page for a tutorial.

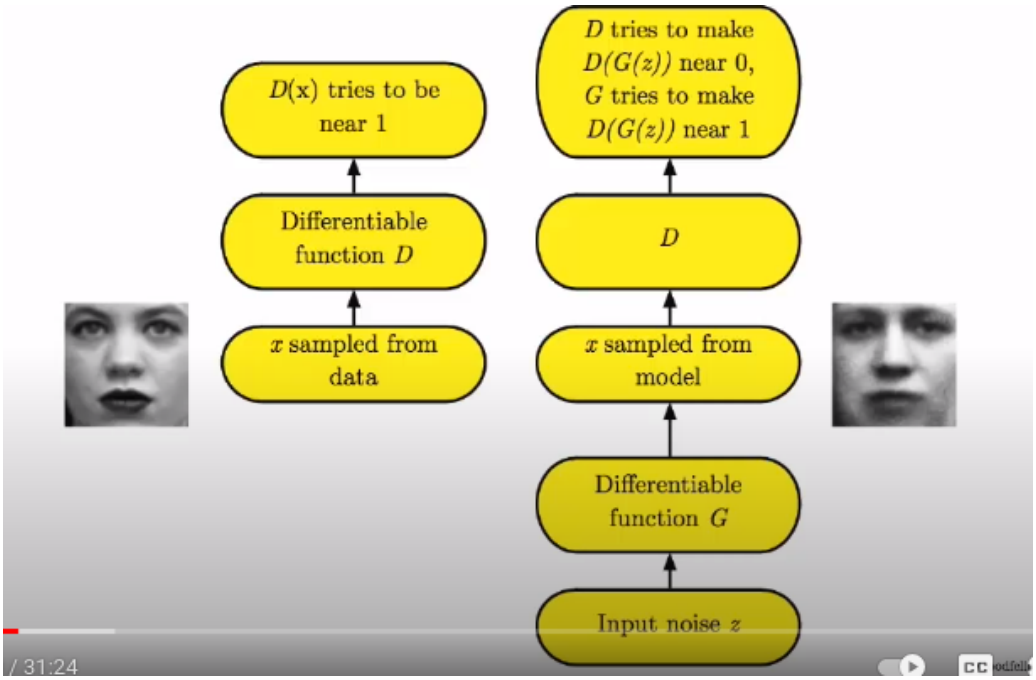

Figure 1: The Discriminator (pictured on the left) has access to samples, and thus a distribution but no labels. The neural network (aka differentiable function D) will output 1 if the input provided to it is a "real" image from the desired distribution. (Right): The generator takes noise as input and conjures up (using a neural network G) an image, and passes it to the D function for criticism. Over time, the generator gets better with the loss function (shown on the top) simulating the expert teaching the novice.

1.1 About the lab

.

Training on a CPU for GANs is not practical. You can use a GPU runtime on Google Colab for the same. Refer to this tutorial on using Google Colab.
We have provided template .ipynb notebooks with TODOs that you need to complete here. You will upload the .ipynb files to google colab and start editing the code. Please do not edit the code outside the TODOs. For each task you need to submit your final notebook with all the TODOs filled, and also the google drive link to the trained model checkpoints (however, don't forget the usual submission guidelines including the reflection essay).
Data and the generator for the assignment is built into the code.
For ease of working with your partners, you can create a shared drive on your IITB LDAP Google Drive account and share this with your teammates. This drive can be linked to the google colab runtime (Linking your LDAP account to drive automatically links all shared drives; refer to the tutorial for linking a google drive account to colab) and you may save your checkpoints directly to drive (instead of downloading/uploading them). Some code to make this happen has been provided.
Submit the following files: **lab8_vanilla.ipynb**, **lab8_cwgan.ipynb**, **checkpoint_links.txt**. The **checkpoint_links.txt** file should contain shareable links (two links – one for each of the questions) of your checkpoints on google drive (not onedrive or dropbox). Please do not make any changes in your code outside the `"##TODO##"`s

2 The Original GAN

In this task, we will be training a simple Deep Convolution GAN (DCGAN)[1] with the MNIST dataset. MNIST is the most commonly used dataset in machine learning (ML) applications with 32x32 grayscale images of 0-9 digits (images are used since it is easier to visualize). Here, the generator and discriminator will be CNNs[2] with the Generator having to upsample[3] to create images. The goal is to generate images as shown in figure 2.
Q: Explain the nomenclature DC

2.1 Tasks

Implement the following:

1. **Generate function:** Define a generate function that takes in the number of images as an argument. Upon calling generate(n), the function must return an nx28x28 numpy array where each of the n 2-D matrices are realistic images that may have been drawn from the same distribution as the training data.

2. **Generator and Discriminator Architectures:**

   The generator takes a noise vector of size (B,100), where B is the batch size and produces a 28x28 image. Design a suitable generator with 2-3 Conv Layers

   Q: What does the number 100 signify? Use the word `dimension` and speculate on the significance.

   The discriminator takes a 28x28 image and produces a probability value as the prediction score. The discriminator uses 3-4 convolution layers preceded by a single linear layer. It also outputs a probability score.

3. **Loss Function:** Define a Binary Cross Entropy loss function (you may use the library function). This will serve as the loss to process the discriminator output. Don't forget to move this to the GPU.

[1] Since the goal is to work images, we work with DCGAN as opposed to GAN
[2] convolve again!
[3] See ConvTranspose2d

Figure 2: The generated images by a trained vanilla GAN on the MNIST dataset

4. **Optimizers:** Define two optimizers of your choice from the pytorch library — one for the generator and the other for the discriminator. You will have to use an appropriate learning rate (tune the learning rate).

5. **Training:** Complete the `train()` function as per the comments in the code

6. **Other parameters and variables:** Recall that we have linked a google drive to store our checkpoints. Store the absolute/relative path to the corresponding folder in the variable `path_to_checkpoint`.

   You also need to tune the number of epochs used for training.

   Q What is the probability distribution of the output images generated by the GAN? Are all digits generated with equal probability? Speculate.

3 Conditional Wasserstein GAN

We introduce two changes to the original GAN.

In the end, the original GAN tries to minimize the distance between the actual (desired) and predicted probability distribution. In doing so, the so-called Kullback-Leibler divergence is used, but the process of reaching the end goal may be tricky. There are other ways in trying to minimize the distance. The so-called Wasserstein GAN, WGAN uses the earth mover distance (also used in image segmentation) to reduce the distance.

Q What is tricky about the original GAN? Write in your own words making sure you understand the buzz words. Try to draw a picture if that helps the explanation.

A conditional GAN is a relaxation and allows the generator to have additional inputs to influence the possible output. In the context of MNIST, we may be interested in generating only the digit 5, for instance. More sophisticated input can be from a completely different domain to enable style transfer. The discriminator is also suitably conditioned. Usually the inputs are concatenated.

We will use the Fashion MNIST dataset for this task. In the previous task, we had no control over the classes of the generated images – all 10 digits could be produced. Now we will introduce a conditional behaviour along with the WGAN to generate images in a controlled "fashion" :). Given a class label, your trained model should be able to generate only the images belonging to that class. **Q** Assume your assignment is successful. How would you use this in a business, real world setting?

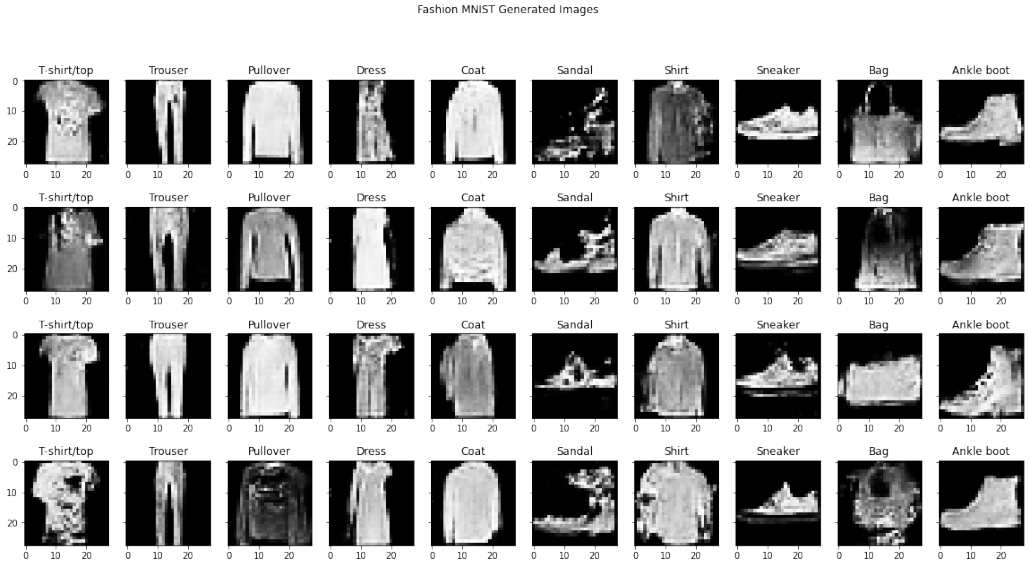

Figure 3: The generated images by a trained cWGAN on the fashion MNIST dataset. Each of the 10 columns represents a class in the dataset.

3.1 Tasks

Implement the following:

1. **Generate function:** Define a generate function that takes in two arguments - the number of images and the class label. Upon calling generate(n, c_i), the function must return an nx28x28 numpy array where each of the n 2-D matrices are realistic images belonging to the class c_i of the dataset.

2. **Generator and Discriminator Architectures:** We introduce conditional behaviour by training both the generator-discriminator architectures while considering the image labels. In other words, we concatenate a vector (or any other data-structure) to the input of both the models that contains the label data, preferably in a one-hot encoding.

   Take the example of the architecture used in the previous task. The Generator takes in a 100-D vector as input. We can extend this vector by appending a one hot representation of the class label (digit). The Discriminator takes in a 28x28 Image. We can expand this image along the channels to a 11x28x28 image by appending a 10x28x28 matrix. Can you think of how the class information will be stored in this 10x28x28 matrix?

   You may suitably modify the architecture used in the original GAN earlier, or you can explore other architectures.

   Q: Describe your process of conditioning.

3. **Loss Function:** WGAN uses the Earth Mover's Distance (or simply Wasserstein Distance) as its loss. Read this tutorial for a basic idea of the EM distance. Our discriminator will now be producing a real number for an input image, unlike the vanilla GAN, which had produced a

probability. Let these scores be:

$$\begin{aligned} score_{real} &= D(x_{real}) \\ score_{fake} &= D(G(z)) \end{aligned}$$

Here z is the latent space for image generation (that is the noise vector used to generate new data). Hence, our training losses will be given by

$$\begin{aligned} L_{generator} &= -score_{fake} \\ L_{discriminator} &= score_{fake} - score_{real} \end{aligned}$$

There yet remains the Lipschitz Continuity condition to be applied on the discriminator (what? why? Go read the tutorial!). The WGAN paper gave a workaround to this by clipping the weights of the model in some range. But even the authors mentioned that "Weight clipping is a clearly terrible way to enforce a Lipschitz constraint". But thankfully, a substitute to this clipping was suggested by Gulrajani et. al. in the form of gradient penalty (conveniently known as WGAN-GP). WGAN-GP is based on the fact that a differentiable function f is 1-Lipschitz if and only if it has gradients with norm at most 1 everywhere. The paper suggests a proxy to WGAN condition by penalising the training if the norm of the gradients moves away from 1. So finally, the discriminator loss is given by

$$L_{discriminator} = score_{fake} - score_{real} + \lambda * (||gradient||_2 - 1)^2$$

Here λ is a regularisation constant (the WGAN-GP paper uses $\lambda = 10$, but you are welcome to tune this further).

Implement the `get_WLoss_generator`, `get_WLoss_discriminator` and

`get_gradient_regularisation` functions.

We have provided you with a `get_grad_function` which returns the gradient at a stage of training.

4. **Optimizers:** Define two optimizers of your choice from the pytorch library - one for the generator and the other for the discriminator. You will have to use an appropriate learning rate (tune the learning rate).

5. **Training:** Complete the `train()` function as per the comments in the code

6. **Other parameters and variables:** Recall that we have linked a google drive to store our checkpoints. Store the absolute/relative path to the corresponding folder in the variable `path_to_checkpoint`. You also need to tune the number of epochs used for training.

4 Submission Guidelines

As in previous labs, and note Sec. 1.1