ELEC 6910T and COMP 6211D
Assignment 3

Qifeng Chen

Introduction

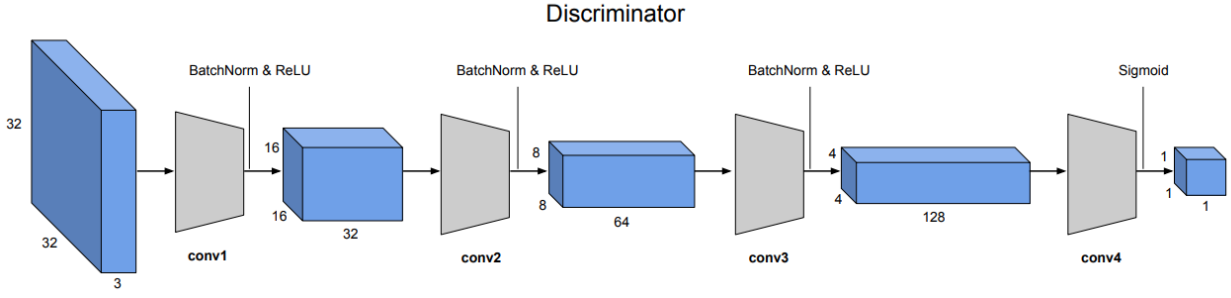
In this assignment, youll get hands-on experience coding and training GANs. This assignment is divided into two parts: in the first part, we will implement a specific type of GAN designed to process images, called a Deep Convolutional GAN (DCGAN). Well train the DCGAN to generate emojis from samples of random noise. In the second part, we will implement a more complex GAN architecture called CycleGAN, which was designed for the task of image-to-image translation (described in more detail in Part 2). Well train the CycleGAN to convert between Apple-style and Windows-style emojis. In both parts, youll gain experience implementing GANs by writing code for the generator, discriminator, and training loop, for each model.

We provide the skeleton code written with Pytorch for your convinience but feel free to use other types of deep learning frameworks (e.g., Tensorflow, MX-Net, Chainer and etc.).

Part 1: Deep Convolutional GAN (DCGAN)[30%]

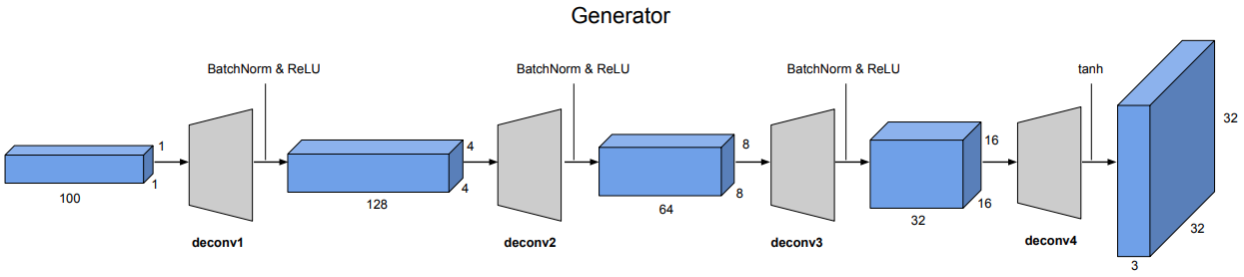
For the first part of this assignment, we will implement a Deep Convolutional GAN (DCGAN). A DCGAN is simply a GAN that uses a convolutional neural network as the discriminator, and a network composed of transposed convolutions as the generator. To implement the DCGAN, we need to specify three things: 1) the generator, 2) the discriminator, and 3) the training procedure. We will develop each of these three components in the following subsections.

Implement the Discriminator of the DCGAN [10%]



Implement this architecture by filling in the **__init__** method of the DCDiscriminator class in **models.py**. Note that the forward pass of DCDiscriminator is already provided for you.

Implement the Generator of the DCGAN [10%]



Implement this architecture by filling in the **__init__** method of the DCGenerator class in **models.py**. Note that the forward pass of DCGenerator is already provided for you.

Implement the Training Loop [10%]

Next, you will implement the training loop for the DCGAN. A DCGAN is simply a GAN with a specific type of generator and discriminator; thus, we train it in exactly the same way as a standard GAN. The pseudo-code for the training procedure is shown below. The actual implementation is simpler than it may seem from the pseudo-code: this will give you practice in translating math to code.

Open up the file **vanilla_gan.py** and fill in the indicated parts of the train function following the pseudo-code shown below. The provided skeleton code basically follows the DCGAN [2] but used least-squares loss proposed in LSGAN [1].

Algorithm 1 GAN Training Loop Pseudocode

procedure TRAINGAN

Draw m training examples $\{x^{(1)}, \ldots, x^{(m)}\}$ from the data distribution p_{data}

Draw m noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from the noise distribution p_z

Generate fake images from the noise: $G(z^{(i)})$ for $i \in \{1, \ldots m\}$

Compute the (least-squares) discriminator loss:

$$J^{(D)} = \frac{1}{2m}\sum_{i=1}^{m}\left[\left(D(x^{(i)}) - 1\right)^2\right] + \frac{1}{2m}\sum_{i=1}^{m}\left[\left(D(G(z^{(i)}))\right)^2\right]$$

Update the parameters of the discriminator

Draw m new noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from the noise distribution p_z

Generate fake images from the noise: $G(z^{(i)})$ for $i \in \{1, \ldots m\}$

Compute the (least-squares) generator loss:

$$J^{(G)} = \frac{1}{m}\sum_{i=1}^{m}\left[\left(D(G(z^{(i)})) - 1\right)^2\right]$$

Update the parameters of the generator

Part 2: CycleGAN [30%]

Generative Adversarial Networks have been successfully applied to image translation, and have sparked a resurgence of interest in the topic. The basic idea behind the GAN-based approaches is to use a conditional GAN to learn a mapping from input to output images. The loss functions of these approaches generally include extra terms (in addition to the standard GAN loss), to express constraints on the types of images that are generated.

A recently-introduced method for image-to-image translation called CycleGAN is particularly interesting because it allows us to use un-paired training data. This means that in order to train it to translate images from domain X to domain Y , we do not have to have exact correspondences between individual images in those domains. For example, in the paper that introduced CycleGANs [3], the authors are able to translate between images of horses and zebras, even though there are no images of a zebra in exactly the same position as a horse, and with exactly the same background, etc.

Thus, CycleGANs enable learning a mapping from one domain X (say, images of horses) to another domain Y (images of zebras) without having to find perfectly matched training pairs.

Emoji CycleGAN

Now well build a CycleGAN and use it to translate emojis between two different styles, in particular, Windows $\Longleftrightarrow$ Apple emojis.

Implement the Generator of the CycleGAN [15%]

The generator in the CycleGAN has layers that implement three stages of computation: 1) the first stage encodes the input via a series of convolutional layers that extract the image features; 2) the second stage then transforms the features by passing them through one or more residual blocks; and 3) the third stage decodes the transformed features using a series of transpose convolutional layers, to build an output image of the same size as the input.

The residual block used in the transformation stage consists of a convolutional layer, where the input is added to the output of the convolution. This is done so that the characteristics of the output image (e.g., the shapes of objects) do not differ too much from the input. Implement the following generator architecture by completing the **__init__** method of the CycleGenerator class in **models.py**.

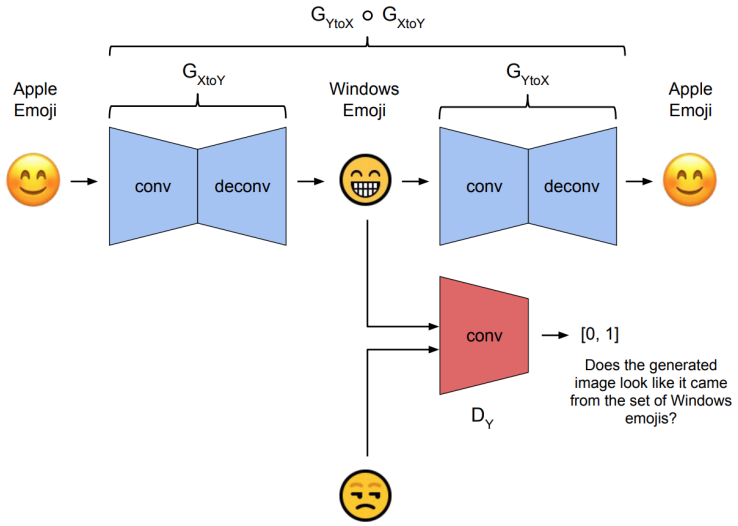
To do this, you will need to use the conv and deconv functions, as well as the ResnetBlock class, all provided in **models.py**.

Note: There are two generators in the CycleGAN model, $G_{X \to Y}$ and $G_{Y \to X}$, but their implementations are identical. Thus, in the code, $G_{X \to Y}$ and $G_{Y \to X}$ are simply different instantiations of the same class.

Implement the Training Loop [15%]

Finally, we will implement the CycleGAN training procedure, which is more involved than the procedure in Part 1.

Similarly to Part 1, this training loop is not as difficult to implement as it may seem. There is a lot of symmetry in the training procedure, because all operations are done for both $X \to Y$ and $Y \to X$ directions. Complete the train function in **cycle_gan.py**.



Cycle Consistency

The most interesting idea behind CycleGANs (and the one from which they get their name) is the idea of introducing a *cycle consistency* loss to constrain the model. The idea is that when we translate an image from domain X to domain Y , and then translate the generated image back to domain X, the result should look like the original image that we started with.

The cycle consistency component of the loss is the mean squared error between the input images and their reconstructions obtained by passing through both generators in sequence (i.e., from domain X to Y via the $X \to Y$ generator, and then from domain Y back to X via the $Y \to X$ generator). The cycle consistency loss for the $Y \to X \to Y$ cycle is expressed as follows:

$$\mathcal{J}_{cycle}^{(Y \to X \to Y)} = \frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} - G_{X \to Y}(G_{Y \to X}(y^{(i)})) \right)^2$$

Implement the cycle consistency loss by filling in the following section in **cycle_gan.py**. Note that there are two such sections, and their implementations are identical except for swapping X and Y . You must implement both of them.

Report [40%]

You should run experiments with completed codes of DCGAN and CycleGAN and report it. For DCGAN, you should run at least 30 epochs. For CycleGAn, you are required to run at least 5,000 itertations. If you have powerful gpus, then you can run extra steps. You are required to report following contents :

- Report the training loss graph of DCGAN. [10%]
- Report the generated image samples of DCGAN. [10%]
- Report the training loss graph of CycleGAN. [10%]
- Report the generated image samples of CycleGAN. [10%]

Algorithm 2 CycleGAN Training Loop Pseudocode

procedure TRAINCYCLEGAN

Draw a minibatch of samples $\{x^{(1)}, \ldots, x^{(m)}\}$ from domain X

Draw a minibatch of samples $\{y^{(1)}, \ldots, y^{(m)}\}$ from domain Y

Compute the discriminator loss on real images:

$$\mathcal{J}_{real}^{(D)} = \frac{1}{m}\sum_{i=1}^{m}(D_X(x^{(i)}) - 1)^2 + \frac{1}{n}\sum_{j=1}^{n}(D_Y(y^{(j)} - 1)^2$$

Compute the discriminator loss on fake images:

$$\mathcal{J}_{fake}^{(D)} = \frac{1}{m}\sum_{i=1}^{m}(D_Y(G_{X\to Y}(x^{(i)})))^2 + \frac{1}{n}\sum_{j=1}^{n}(D_X(G_{Y\to X}(y^{(j)})))^2$$

Update the discriminators

Compute the $Y \to X$ generator loss:

$$\mathcal{J}^{(G_{Y\to X})} = \frac{1}{n}\sum_{j=1}^{n}(D_X(G_{Y\to X}(y^{(j)})) - 1)^2 + \mathcal{J}_{cycle}^{(Y\to X\to Y)}$$

Compute the $X \to Y$ generator loss:

$$\mathcal{J}^{(G_{X\to Y})} = \frac{1}{m}\sum_{i=1}^{m}(D_Y(G_{X\to Y}(x^{(i)})) - 1)^2 + \mathcal{J}_{cycle}^{(X\to Y\to X)}$$

Update the generators

Submission

You should submit your assignemnt in the format of a zip file. The name of a submission file should be like 'PA3_{your name}_{your student id}.zip'. You must contain four files in your submission.

- models.py
- cycle_gan.py
- vanilla_gan.py
- report.pdf

Deadline : April 30, 2019. 11:59pm
Submit to : hr.yang@connect.ust.hk

References

[1] Xudong Mao, Qing Li, Haoran Xie, Raymond YK Lau, Zhen Wang, and Stephen Paul Smolley. Least squares generative adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2794–2802, 2017.

[2] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[3] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232, 2017.