Instructor: Vishnu Boddeti
CSE 891-001: Deep Learning
Programming Homework 4
December, 15

Instructions:

- **Submission:** Only homeworks uploaded to Google Classroom will be graded. Submit solutions in PDF format.
- **Integrity and Collaboration:** You are expected to work on the homeworks by yourself. You are not permitted to discuss them with anyone except the instructor. The homework that you hand in should be entirely your own work. You may be asked to demonstrate how you got any results that you report.
- **Clarifications:** If you have any question, please look at Google Classroom first. Other students may have encountered the same problem, and is solved already. If not, post your question there. We will respond as soon as possible.

Introduction: In this assignment, we will get hands-on experience with coding and training GANs. This assignment consists of two parts: in the first part, we will implement a specific type of GAN designed to process images, called a Deep Convolutional GAN (DCGAN) [1]. We will 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 [2], which was designed for the task of image-to-image translation. We will train the CycleGAN to convert between Apple-style and Windows-style emojis. In both parts, you will gain experience implementing GANs by writing code for the generator, discriminator, and the training loop, for each model.

1 Deep Convolutional GAN (DCGAN)

For the first part of this assignment, you 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.

1.1 Implement the Discriminator of DCGAN [1pt]

The DCGAN discriminator is a convolutional neural network that has the following architecture:

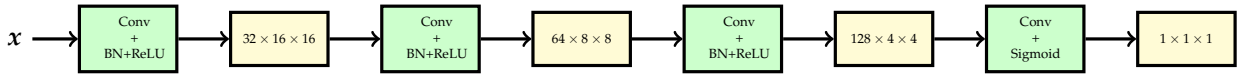

Figure 1: Discriminator Architecture for DCGAN

1. **Padding:** In each of the convolutional layers shown above, we down sample the spatial dimension of the input volume by a factor of 2. Given that we use kernel size $K = 5$ and stride $S = 2$, what should the padding be? Provide your answer in your writeup, and show your work (e.g., the formula you used to derive the padding).
2. **Implementation:** 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.

1.2 Implement the Generator of DCGAN [1pt]

Now, we will implement the generator of the DCGAN, which consists of a sequence of transpose convolutional layers that progressively up sample the input noise sample to generate a fake image. The generator we will use in this DCGAN has the following architecture:

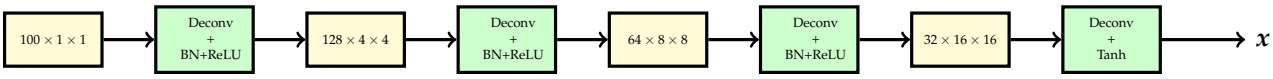

Figure 2: Generator Architecture for DCGAN

1. **Implementation:** 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.

   Note: The original DCGAN generator uses the `deconv` function to expand the spatial dimension. Odena et al. [3] later found that `deconv` creates checker board artifacts in the generated samples. In this assignment, we will use `upconv` that consists of an upsampling layer followed by conv2D to replace the `deconv` module (analogous to the conv function used for the discriminator above) in your generator implementation.

1.3 Implement the Training Loop [1.5pts]

Next, we will implement the training loop for DCGAN. DCGAN is simply a GAN with a specific type of generator and discriminator; so, we train it in exactly the same way as a standard GAN. We will use a variant, called, LS-GAN [4] where the discriminator uses a least squares loss function instead of the cross-entropy loss. The pseudo-code for the training procedure is shown in Algorithm 1. The actual implementation is simpler than it may seem from the pseudo-code: this will give you practice in translating math to code.

Algorithm 1 LS-GAN Training Loop Pseudocode

1: **procedure** TRAINGAN
2: Draw m training examples $\{x^{(1)}, \ldots, x^{(m)}\}$ from the data distribution p_{data}
3: Draw m noise examples $\{z^{(1)}, \ldots, z^{(m)}\}$ from the data distribution p_z
4: Generate fake images from the noise: $G(z^{(i)})$ for $i \in \{1, \ldots, m\}$
5: Compute the (least-squares) discriminator loss:

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

6: Update the parameters of the discriminator
7: Draw m new noise examples $\{z^{(1)}, \ldots, z^{(m)}\}$ from the data distribution p_z
8: Generate new fake images from the noise: $G(z^{(i)})$ for $i \in \{1, \ldots, m\}$
9: Compute the (least-squares) generator loss:

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

10: Update the parameters of the generator

1. **Implementation:** Open up the file `vanilla_gan.py` and fill in the indicated parts of the `training_loop` function. There are 5 numbered bullets in the code to fill in for the discriminator and 3 bullets for the generator. Each of these can be done in a single line of code.

1.4 DCGAN Experiments [1pt]

1. Train the DCGAN with the following command:

   ```
   python vanilla_gan.py
   ```

   By default, the script runs for 20000 iterations, and should take approximately 30 minutes to run on a regular desktop. The script saves the output of the generator for a fixed noise sample every 200 iterations throughout training; this allows you to see the improvement, over time, in the images generated by the generator. Include in your write-up one of the samples from early in training (e.g., iteration 10000) and one of the samples from later in training, and give the iteration number for those samples. Briefly comment on the quality of the samples, and in what way they improve through training.

2. Multiple techniques can be used to stabilize GAN training. We have provided code for `gradient_penalty` [5]. Try turn on the `gradient_penalty` flag in the args parser and train the model again. Are you able to stabilize the training? Briefly explain why the gradient penalty can help. You are welcome to check out the related literature above for gradient penalty.

3. Play with some other hyperparameters such as `spectral_norm`. You can also try lowering `lr` (learning rate), and increasing `d_train_iters` (number of discriminator updates per generator update) to 5. Are you able to stabilize the training? Can you explain why the above measures help?

2 CycleGAN

In this problem you will implement the CycleGAN architecture.

2.1 Motivation: Image-to-Image Translation

Say you have a picture of a sunny landscape, and you wonder what it would look like in the rain. Or perhaps you wonder what a painter like Monet or van Gogh would see in it? These questions can be addressed through image-to-image translation, wherein an input image is automatically converted into a new image with some desired appearance.

Recently, 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 need exact correspondences between individual images in those domains. For example, in the paper that introduced CycleGANs, 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. To summarize the differences between paired and un-paired data, we have:

- Paired training data: $\{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$
- Un-paired training data:
  - Source set: $\{x^{(i)}\}$ with each $x^(i) \in X$
  - Target set: $\{y^{(i)}\}$ with each $y^(i) \in Y$
  - For example, X is the set of horse pictures, and Y is the set of zebra pictures, where there are no direct correspondences between images in X and Y.

2.2 Emoji CycleGAN

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

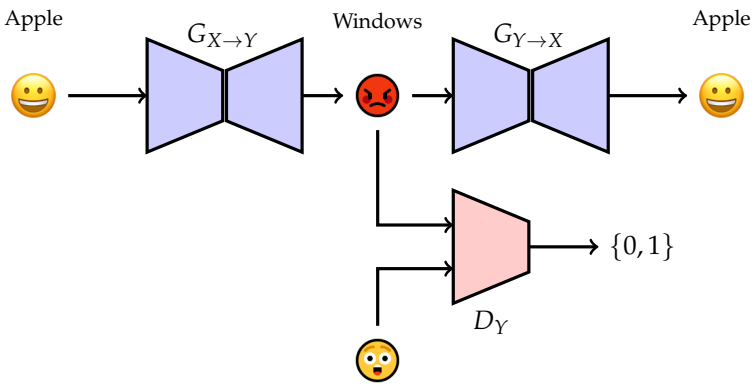

Figure 3: CycleGAN architecture includes two generators $G_{X\to Y}$, $G_{Y\to X}$ and a discriminator D_Y.

2.3 Implement the Generator of the CycleGAN [2pts]

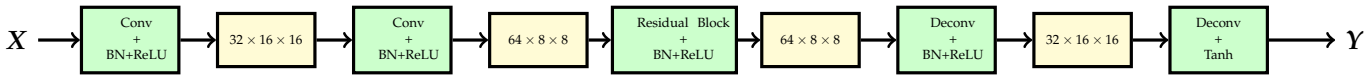

Figure 4: Generator Architecture $G_{X\to Y}$ for CycleGAN

The generator in 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) does 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 `upconv` 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.

2.4 Implement the CycleGAN Training Loop [2pts]

Finally, we will implement the CycleGAN training procedure, this is more involved than the training procedure for DCGAN. Just as in the DCGAN case, the training loop is not as difficult to implement as it may

Algorithm 2 CycleGAN Training Loop Pseudocode

1: **procedure** TRAINCYCLEGAN
2: Draw a minibatch of samples $\{x^{(1)}, \ldots, x^{(m)}\}$ from domain X
3: Draw a minibatch of samples $\{y^{(1)}, \ldots, y^{(m)}\}$ from domain Y
4: Compute the discriminator loss on real images:

$$L_{real}^{D} = \frac{1}{m}\sum_{i=1}^{m}\left[\left(D_X(x^{(i)}) - 1\right)^2\right] + \frac{1}{n}\sum_{i=1}^{n}\left[\left(D_Y(y^{(i)}) - 1\right)^2\right] \tag{1}$$

5: Compute the discriminator loss on fake images:

$$L_{fake}^{D} = \frac{1}{m}\sum_{i=1}^{m}\left[\left(D_Y(G_{X\to Y}(x^{(i)}))\right)^2\right] + \frac{1}{n}\sum_{i=1}^{n}\left[\left(D_X(G_{Y\to X}(y^{(i)}))\right)^2\right]$$

6: Update the parameters of the discriminators
7: Compute the $Y \to X$ generator loss:

$$L^{(G_{Y\to X})} = \frac{1}{m}\sum_{i=1}^{m}\left[\left(D_X(G_{Y\to X}(y^{(i)})) - 1\right)^2\right] + \lambda_{cycle} L_{cycle}^{Y\to X\to Y}$$

8: Compute the $Y \to X$ generator loss:

$$L^{(G_{X\to Y})} = \frac{1}{m}\sum_{i=1}^{m}\left[\left(D_Y(G_{X\to Y}(x^{(i)})) - 1\right)^2\right] + \lambda_{cycle} L_{cycle}^{X\to Y\to X}$$

9: Update the parameters of the generators

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 `training_loop` function in `cycle_gan.py`. There are 5 bullet points in the code for training the discriminators, and 6 bullet points in total for training the generators. Due to the symmetry between domains, several parts of the code you fill in will be identical except for swapping X and Y; this is normal and expected.

Cycle Consistency: The most interesting idea behind CyclyGANs 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 the 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 $L_{cycle}^{Y\to X\to Y}$ for the $Y \to X \to Y$ cycle is expressed as follows:

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

The *cycle consistency loss* for the $X \to Y \to X$ cycle is defined analogously.

Implement the cycle consistency loss by filling in the corresponding sections 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.

2.5 CycleGAN Experiments [1.5pts]

Training the CycleGAN from scratch can be time-consuming if you do not have a GPU. In this part, you will train your models from scratch for just 600 iterations, to check the results. To reduce training time, we provided the weights of pre-trained models that you can load into your implementation. In order to load the weights, your implementation must be correct.

1. Train the CycleGAN from scratch using the command:

   ```
   python cycle_gan.py
   ```

   This runs for 5000 iterations, and saves the generated samples in the `results/samples_cyclegan` folder. In each sample, images from the source domain are shown with their translations to the right. Include in your writeup the samples from both generators at either iteration 4000 or 5000, e.g., `sample-004000-X-Y.png` and `sample-004000-Y-X.png`.

2. Change the random seed and train the CycleGAN again. What are the most noticeable differences between the *similar* quality samples from the different random seeds? Explain why there is such a difference?

3. Changing the default `lambda_cycle` hyperparameters and train the CycleGAN again. Try a couple of different values including without the cycle-consistency loss. (i.e. `lambda_cycle = 0`).

   For different values of `lambda_cycle`, include in your writeup some samples from both generators at either iteration 400 and samples from a later iteration. Do you notice a difference between the results with and without the cycle consistency loss? Write down your observations (positive or negative) in your writeup. Can you explain these results, i.e., why there is or isn't a difference among the experiments?

Submission: Here is everything you need to submit.

- Three code files: `models.py`, `vanilla_gan.py` and `cycle_gan.py`
- A PDF document titled `programming-assignment-4-msunetid.pdf` containing samples generated by your DCGAN and CycleGAN models, and your answers to the written questions.

References

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

[2] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. *arXiv preprint*, 2017.

[3] Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and checkerboard artifacts. *Distill*, 2016.

[4] Xudong Mao, Qing Li, Haoran Xie, Raymond YK Lau, Zhen Wang, and Stephen Paul Smolley. Least squares generative adversarial networks. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, pages 2813–2821. IEEE, 2017.

[5] Hoang Thanh-Tung, Truyen Tran, and Svetha Venkatesh. Improving generalization and stability of generative adversarial networks. *arXiv preprint arXiv:1902.03984*, 2019.