Programming Assignment 4: CycleGAN

Deadline: March 29, 2019 at 11:59pm
Based on an assignment by Paul Vicol

Submission: You must submit three files through MarkUs[1]: a PDF file containing your writeup, titled `a4-writeup.pdf`, and your code `cycle_gan.ipynb`. Your writeup must be typeset using LaTeX.

The programming assignments are individual work. See the Course Information handout[2] for detailed policies.

You should attempt all questions for this assignment. Most of them can be answered at least partially even if you were unable to finish earlier questions. If you were unable to run the experiments, please discuss what outcomes you might hypothetically expect from the experiments. If you think your computational results are incorrect, please say so; that may help you get partial credit.

Introduction

In this assignment, you'll 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). We'll 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). We'll train the CycleGAN to convert between Apple-style and Windows-style emojis.

In both parts, you'll gain experience implementing GANs by writing code for the generator, discriminator, and training loop, for each model.

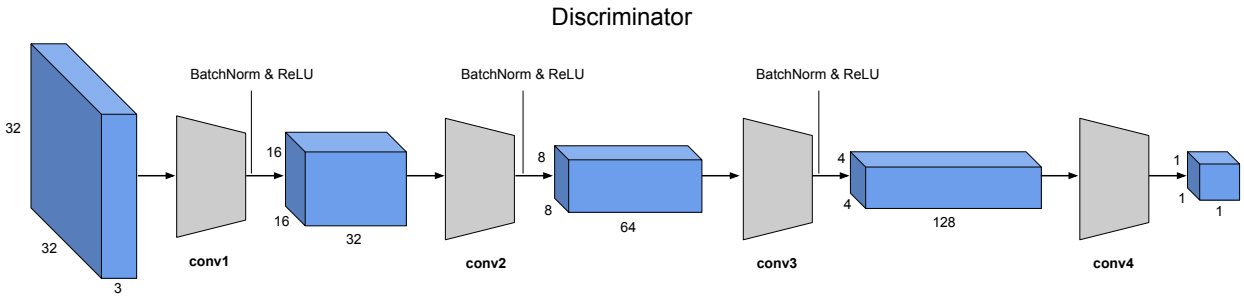
Part 1: Deep Convolutional GAN (DCGAN)

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%]

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

[1] `https://markus.teach.cs.toronto.edu/csc321-2018-01`
[2] `http://cs.toronto.edu/~rgrosse/courses/csc421_2019/syllabus.pdf`



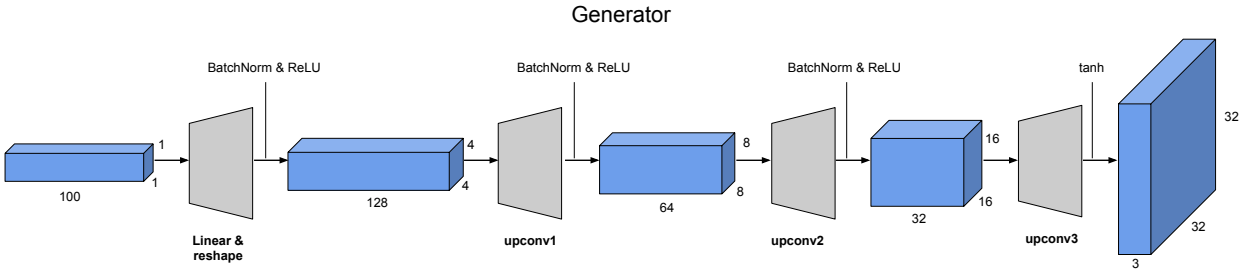
1. **Padding:** In each of the convolutional layers shown above, we downsample the spatial dimension of the input volume by a factor of 2. Given that we use kernel size $K = 5$ and stride $S = 2$, what should the padding be? Write your answer in your writeup, and show your work (e.g., the formula you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

2. **Implementation:** Implement this architecture by filling in the `__init__` method of the `DCDiscriminator` class, shown below. Note that the forward pass of `DCDiscriminator` is already provided for you.

   Note: The function `conv` in `Helper Modules` has an optional argument `batch_norm`: if `batch_norm` is `False`, then `conv` simply returns a `torch.nn.Conv2d` layer; if `batch_norm` is `True`, then `conv` returns a network block that consists of a `Conv2d` layer followed by a `torch.nn.BatchNorm2d` layer. **Use the conv function in your implementation.**

Generator [10%]

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



1. **Implementation:** Implement this architecture by filling in the `__init__` method of the `DCGenerator` class, shown below. Note that the forward pass of `DCGenerator` is already provided for you.

   Note: The original DCGAN generator uses `deconv` function to expand the spatial dimension. Odena et al. later found the `deconv` creates checker board artifacts in the generated samples. In this assignment, we will use `upcome` 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.

Training Loop [15%]

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.

Algorithm 1 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 samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from the noise 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:**

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

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

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

10: Update the parameters of the generator

1. **Implementation:** Fill in the `gan_training_loop` function in the GAN section of the notebook.

   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, although you will not lose marks for using multiple lines.

Experiment [10%]

1. We will train a DCGAN to generate Windows (or Apple) emojis in the Training - GAN section of the notebook. By default, the script runs for 5000 iterations, and should take approximately 10 minutes on Colab. The script saves the output of the generator for a fixed noise sample every 200 iterations throughout training; this allows you to see how the generator improves over time. You can stop the training after obtaining satisfactory image samples. **Include in your write-up one of the samples from early in training (e.g., iteration 200) 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.**

Part 2: CycleGAN

Now we are going to implement the CycleGAN architecture.

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 have to have 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: $\{x^{(i)}\}_{i=1}^{N}$ with each $x^{(i)} \in X$
  - Target set: $\{y^{(j)}\}_{j=1}^{M}$ with each $y^{(j)} \in Y$
  - 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

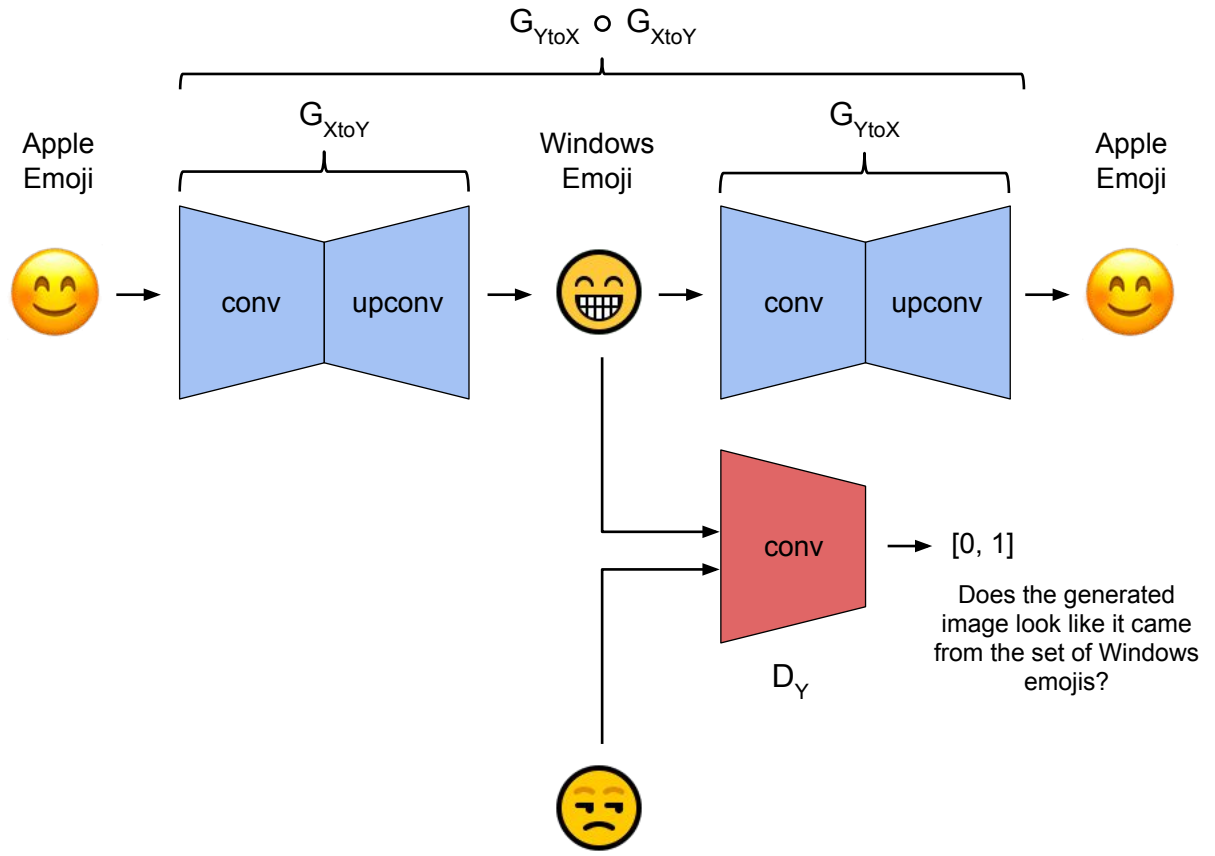
Emoji CycleGAN

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

Generator [20%]

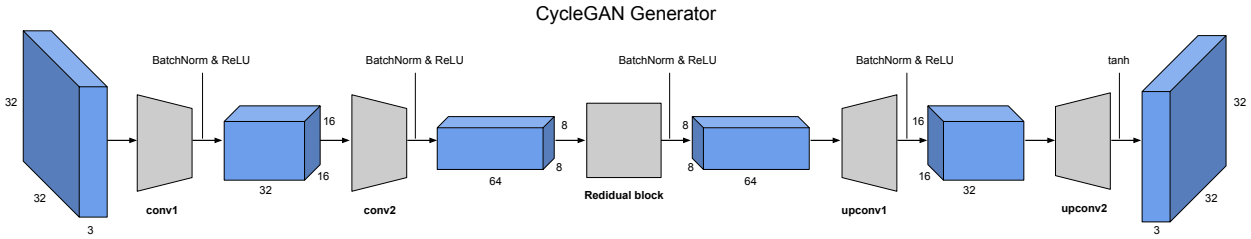
The generator in the CycleGAN has layers that implement three stages of computation: 1) the first stage *encodes* the input via a series of convolutional layers that extract the image features; 2) the second stage then *transforms* the features by passing them through one or more *residual blocks*; and 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.

To do this, you will need to use the `conv` and `upconv` functions, as well as the `ResnetBlock` class, all provided in `Helper Modules`.



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.

CycleGAN Training Loop [20%]

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

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:

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

5: Compute the discriminator loss on fake images:

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

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

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

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

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

9: Update the generators

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 `cyclegan_training_loop` function, starting from the following section:

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 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 L1 distance 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:

$$\lambda_{cycle}\mathcal{J}_{cycle}^{(X\to Y\to X)} = \lambda_{cycle}\frac{1}{m}\sum_{i=1}^{m}\|y^{(i)} - G_{X\to Y}(G_{Y\to X}(y^{(i)}))\|_1,$$

where λ_{cycle} is a scalar hyper-parameter balancing the two loss terms: the cycle consistant loss and the GAN loss. The loss for the $X \to Y \to X$ cycle is analogous.

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

CycleGAN Experiments [15%]

1. Train the CycleGAN to translate Apple emojis to Windows emojis in the Training - CycleGAN section of the notebook. The script will train for 10,000 iterations, and saves generated samples in the `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 200 and samples from a later iteration.

2. Change the random seed and train the CycleGAN again. What are the most noticible difference 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 200 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?

What you need to submit

- Your code file: `cycle_gan.ipynb`.
- A PDF document titled `a4-writeup.pdf` containing samples generated by your DCGAN and CycleGAN models, and your answers to the written questions.

Further Resources

For further reading on GANs in general, and CycleGANs in particular, the following links may be useful:

1. Deconvolution and Checkerboard Artifacts (Odena et al., 2016)
2. Unpaired image-to-image translation using cycle-consistent adversarial networks (Zhu et al., 2017)
3. Generative Adversarial Nets (Goodfellow et al., 2014)
4. An Introduction to GANs in Tensorflow
5. Generative Models Blog Post from OpenAI
6. Official PyTorch Implementations of Pix2Pix and CycleGAN