Programming Assignment 4: CycleGAN

Deadline: April 3, at 11:59pm
TAs: Guodong Zhang (csc321staff@cs.toronto.edu)
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 files `models.py` and `cycle_gan.py`. 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 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.

Important Note

The training scripts run **much faster** ($\sim$ 5x faster) on the teaching lab machines if you add `MKL_NUM_THREADS=1` before the Python call, as follows:

```
MKL_NUM_THREADS=1 python cycle_gan.py --load=pretrained_cycle --train_iters=100
```

Remember to add `MKL_NUM_THREADS=1` before each Python command in this assignment.

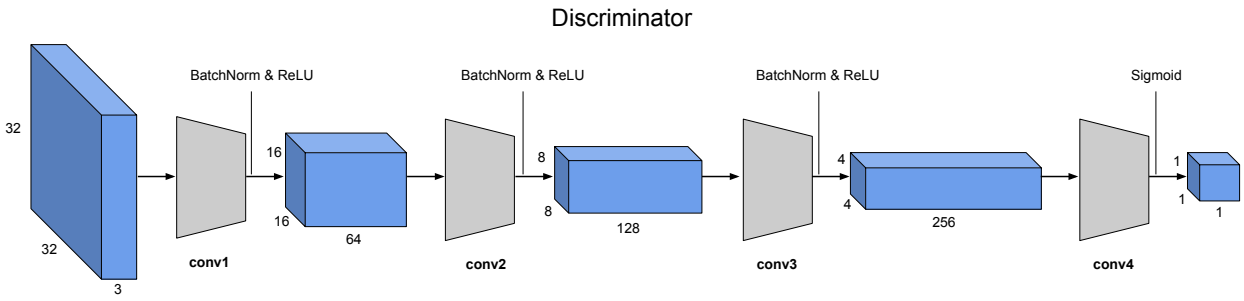
Part 1: Deep Convolutional GAN (DCGAN)

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.

[1] https://markus.teach.cs.toronto.edu/csc321-2018-01
[2] http://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/syllabus.pdf

Implement the Discriminator of the DCGAN [10%]

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



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 $K = 4$ 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).

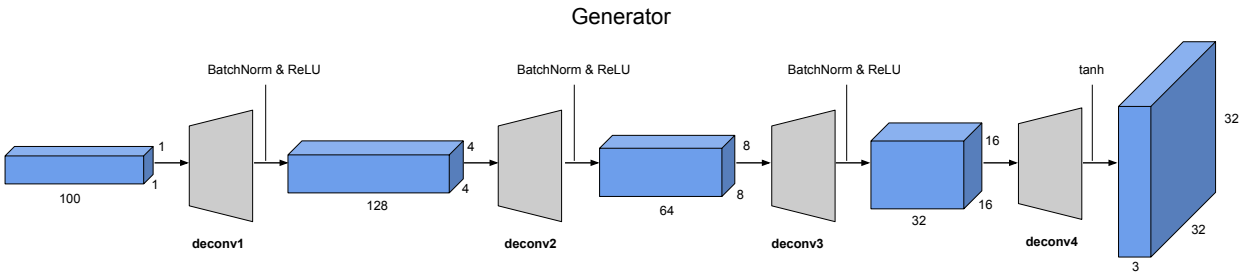
2. **Implementation:** Implement this architecture by filling in the `__init__` method of the `DCDiscriminator` class in `models.py`, shown below. Note that the forward pass of `DCDiscriminator` is already provided for you.

```
def __init__(self, conv_dim=64):
    super(DCDiscriminator, self).__init__()

    ########################################
    ##   FILL THIS IN: CREATE ARCHITECTURE   ##
    ########################################

    # self.conv1 = conv(...)
    # self.conv2 = conv(...)
    # self.conv3 = conv(...)
    # self.conv4 = conv(...)
```

Note: The function `conv` in `models.py` 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 [20%]

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 we'll use in this DCGAN has the following architecture:



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

```
def __init__(self, noise_size, conv_dim):
    super(DCGenerator, self).__init__()

    ########################################
    ##   FILL THIS IN: CREATE ARCHITECTURE   ##
    ########################################

    # self.deconv1 = deconv(...)
    # self.deconv2 = deconv(...)
    # self.deconv3 = deconv(...)
    # self.deconv4 = deconv(...)
```

Note: Use the `deconv` function (analogous to the `conv` function used for the discriminator above) in your generator implementation.

Training Loop [20%]

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.

1. **Implementation:** Open up the file `vanilla_gan.py` and fill in the indicated parts of the `training_loop` function, starting at line 149, i.e., where it says

```
# FILL THIS IN
# 1. Compute the discriminator loss on real images
# D_real_loss = ...
```

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.

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:

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

  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:

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

  Update the parameters of the generator

Experiment [10%]

1. Train the DCGAN with the command:

```
python vanilla_gan.py --num_epochs=40
```

   By default, the script runs for 40 epochs (5680 iterations), and should take approximately 30 minutes on the teaching lab machines (it may be faster on your own computer). 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. **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.

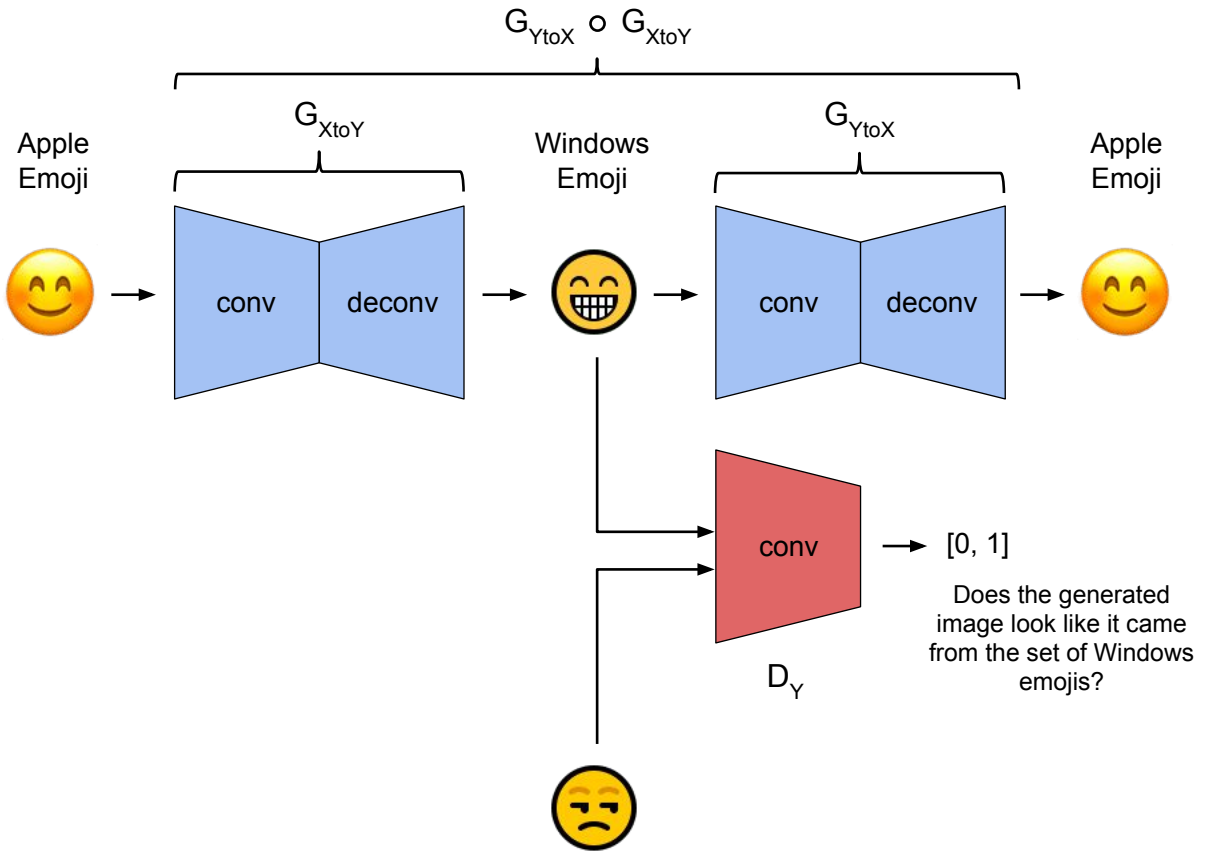
Generator [20%]

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.

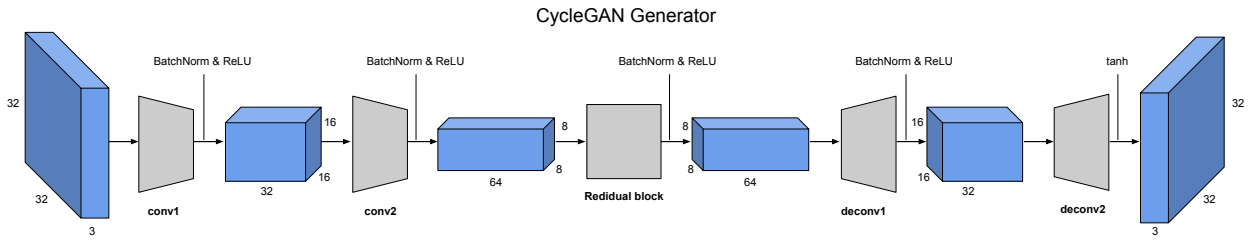
Implement the following generator architecture by completing the `__init__` method of the `CycleGenerator` class in `models.py`.

```
def __init__(self, conv_dim=64, init_zero_weights=False):
    super(CycleGenerator, self).__init__()

    ###########################################
    ##   FILL THIS IN: CREATE ARCHITECTURE   ##
    ###########################################

    # 1. Define the encoder part of the generator
    # self.conv1 = ...
```




```
# self.conv2 = ...

# 2. Define the transformation part of the generator
# self.resnet_block = ...

# 3. Define the decoder part of the generator
# self.deconv1 = ...
# self.deconv2 = ...
```

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.

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:

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

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

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

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 `training_loop` function in `cycle_gan.py`, starting from the following section:

```
# =========================================
#            TRAIN THE DISCRIMINATORS
# =========================================

########################################
##             FILL THIS IN           ##
########################################

# Train with real images
d_optimizer.zero_grad()

# 1. Compute the discriminator losses on real images
# D_X_loss = ...
```

```
# D_Y_loss = ...
```

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 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:

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

The loss for the $X \to Y \to X$ cycle is analogous.

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.

```
if opts.use_cycle_consistency_loss:
    reconstructed_X = G_YtoX(fake_Y)

    # 3. Compute the cycle consistency loss (the reconstruction loss)
    # cycle_consistency_loss = ...

    g_loss += cycle_consistency_loss
```

CycleGAN Experiments [10%]

Training the CycleGAN from scratch can be time-consuming if you don't have a GPU. In this part, you will train your models from scratch for just 600 iterations, to check the results. To save training time, we provide 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 *without* the cycle-consistency loss from scratch using the command:

   ```
   python cycle_gan.py
   ```

   This runs for 600 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 400 or 600, e.g., `sample-000400-X-Y.png` and `sample-000400-Y-X.png`.

2. Train the CycleGAN *with* the cycle-consistency loss from scratch using the command:

```
python cycle_gan.py --use_cycle_consistency_loss
```

Similarly, this runs for 600 iterations, and saves generated samples in the `samples_cyclegan_cycle` folder. Include in your writeup the samples from both generators at either iteration 400 or 600 as above.

3. Now, we'll switch to using pre-trained models, which have been trained for 40000 iterations. Run the pre-trained CycleGAN *without* the cycle-consistency loss using the command:

```
python cycle_gan.py --load=pretrained --train_iters=100
```

You only need 100 training iterations because the provided weights have already been trained to convergence. The samples from the generators will be saved in the folder `samples_cyclegan_pretrained`. **Include the sampled output from your model.**

4. Run the pre-trained CycleGAN *with* the cycle-consistency loss using the command:

```
python cycle_gan.py --load=pretrained_cycle \
                    --use_cycle_consistency_loss \
                    --train_iters=100
```

The samples will be saved in the folder `samples_cyclegan_cycle_pretrained`. **Include the final sampled output from your model.**

5. 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 between the two?

What you need to submit

- Three code files: `models.py`, `vanilla_gan.py`, and `cycle_gan.py`.
- 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. Unpaired image-to-image translation using cycle-consistent adversarial networks (Zhu et al., 2017)
2. Generative Adversarial Nets (Goodfellow et al., 2014)
3. An Introduction to GANs in Tensorflow
4. Generative Models Blog Post from OpenAI
5. Official PyTorch Implementations of Pix2Pix and CycleGAN