MIT CSAIL

6.869 Advances in Computer Vision

Fall 2018

Problem Set 7

Posted: Thursday, November 15, 2018 **Due:** Thursday, November 29, 2018

Submission Instructions: Please submit **two separate files:** 1) a report named ⟨your_kerberos⟩.pdf, including your responses to all required questions with images and/or plots showing your results, 2) a file named ⟨your_kerberos⟩.zip, containing relevant source code. **Submissions that do not adhere to these instructions are subject to an additional penalty.**

Late Submission Policy: We do not accept late submissions. The submission deadline has a 50-minute soft cut-off; after midnight Thursday, submissions are penalized 2% per minute late.

Collaborators: You are free to discuss problems with other students but all writing must be done individually. Please list all collaborators at the top of your report.

6.869 Required — 6.819 Extra Credit

Pset 7 is required for all students in the graduate version of this class: 6.869.

Pset 7 is optional for students in the undergraduate version: 6.819. Undergrad-registered students can receive up to 4 extra credit points for completing this problem set. Since each problem set is out of 10 possible points, this is the equivalent of 40% of one problem set. Points will be spread out and applied to problem sets where you have gotten points off. Students cannot receive more than 10 points (100%) for a problem set using these extra credit points, so if you have received full marks on every problem set these extra credit points won't give you any bonus. These points will not be applied to your final project grade. Happy Thanksgiving!

Introduction

In this assignment, you will get hands-on experience coding and training GANs. We will implement a GAN architecture called CycleGAN, which was designed for the task of *image-to-image translation*. We'll train the CycleGAN to convert between Apple-style and Windows-style emojis.

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

We provide a script to check your models, that you can run as follows:

```
python model_checker.py
```

This checks that the outputs of `CycleGenerator` match the expected outputs for specific inputs. This model checker is provided for convenience only. It may give false negatives, so do not use it as the only way to check that your model is correct.

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.

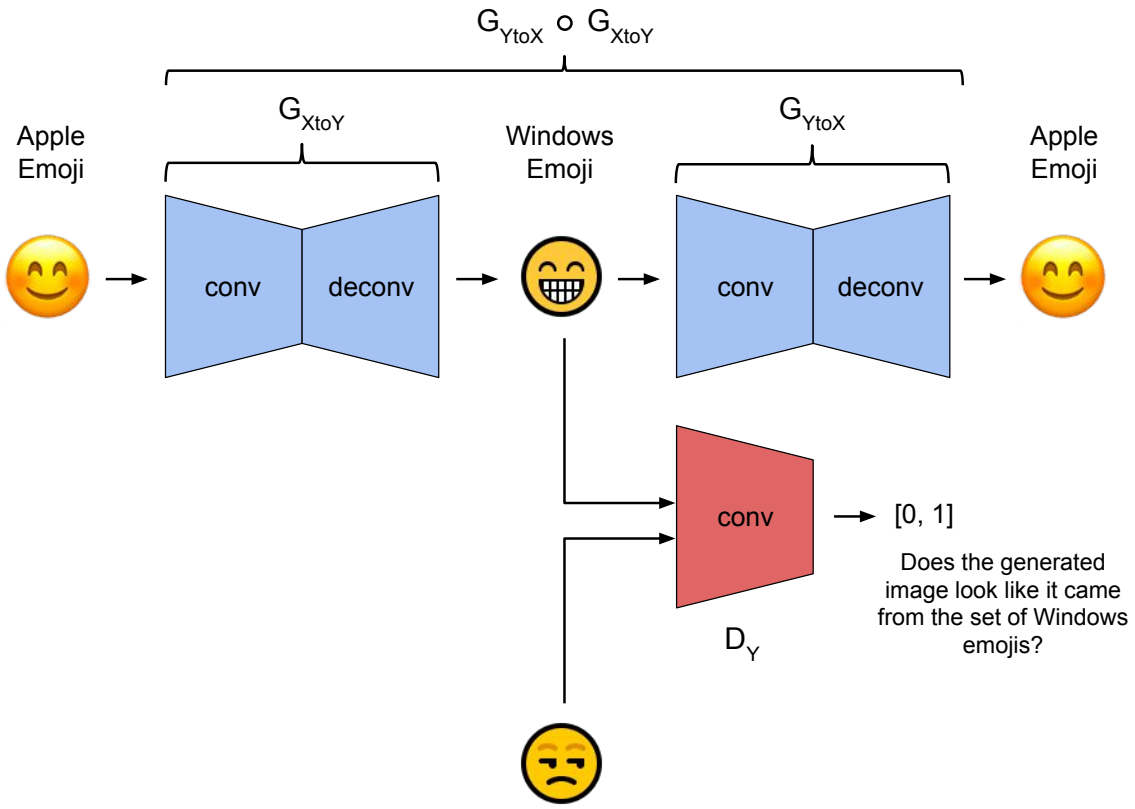
Recently, Generative Adversarial Networks have been successfully applied to image translation, and have sparked a resurgence of interest in the topic. The basic idea behind the GAN-based approaches is to use a conditional GAN to learn a mapping from input to output images. The loss functions of these 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 ↔ Apple emojis.

Generator [30%]

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__()

    ###########################################
```

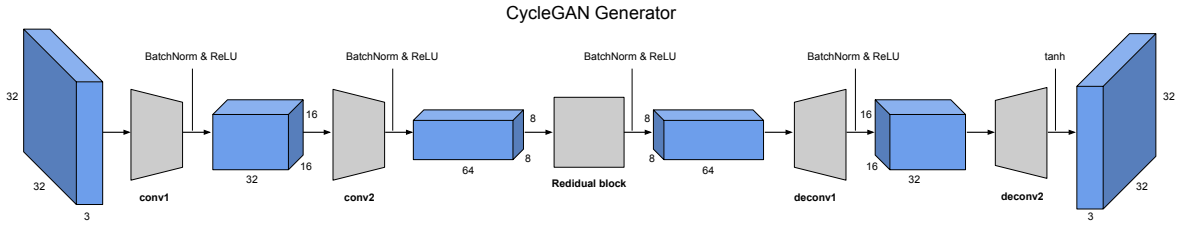
```
##   FILL THIS IN: CREATE ARCHITECTURE   ##
###########################################

# 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 [30%]

Next, we will implement the CycleGAN training procedure.

Algorithm 1 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:

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

Compute the discriminator loss on fake images:

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

Update the discriminators

Compute the $Y \to X$ generator loss:

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

Compute the $X \to Y$ generator loss:

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

Update the generators

Tthis 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 [40%]

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?

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