CSC2516: PA #4

Due on Mar 29

Roger Grosse, Jimmy Ba

Thomas Hollis

Problem 1.1.1

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? Show your work (e.g. the formula you used to derive the padding).

Solution

To calculate the padding, we can use the following equation (derived from the convolution diagram):

$$O = \frac{W - K + 2P}{S} + 1 \qquad \therefore P = \frac{(O-1) \times S + K - W}{2} \tag{1}$$

In (1), O is the output volume, W is the input volume, K is the kernel size, P is the zero-padding and S is the stride.

Thus from (1) we can calculate all the padding required for each convolutional layer (adding a ceiling to get a whole number to input in the code):

$$P_{conv1} = ceil\Big(\frac{(O_{conv1} - 1) \times S + K - W_{conv1}}{2}\Big) = ceil\Big(\frac{(16-1) \times 2 + 5 - 32}{2}\Big) = 2 \tag{2}$$

$$P_{conv2} = ceil\Big(\frac{(O_{conv2} - 1) \times S + K - W_{conv2}}{2}\Big) = ceil\Big(\frac{(8-1) \times 2 + 5 - 16}{2}\Big) = 2 \tag{3}$$

$$P_{conv3} = ceil\Big(\frac{(O_{conv3} - 1) \times S + K - W_{conv3}}{2}\Big) = ceil\Big(\frac{(4-1) \times 2 + 5 - 8}{2}\Big) = 2 \tag{4}$$

$$P_{conv4} = ceil\Big(\frac{(O_{conv4} - 1) \times S + K - W_{conv4}}{2}\Big) = ceil\Big(\frac{(1-1) \times 2 + 5 - 4}{2}\Big) = 1 \tag{5}$$

Therefore from equations (2)-(5) the padding should be 2 (conv1), 2 (conv2), 2 (conv3) and 1 (conv4).

Problem 1.1.2

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.

Solution

See code in `cycle_gan.ipynb`.

Problem 1.2.1

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

Solution

See code in `cycle_gan.ipynb`.

Problem 1.2.2

Fill in the `gan_training_loop` function in the GAN section of the notebook.

Solution

See code in `cycle_gan.ipynb`.

Problem 1.3

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.

Solution

See code in `cycle_gan.ipynb`.

The output of training over 5000 iterations gives the following images:

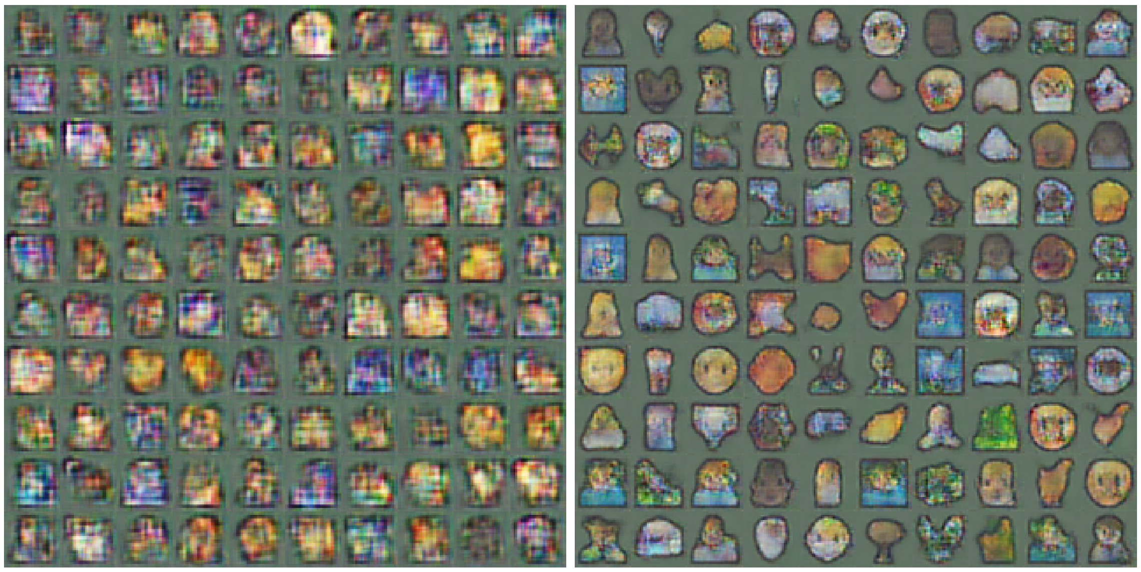

Figure 1: Samples 200 (left) and 5000 (right) of 5000 iterations of DCGAN

From Figure 1 above, we can clearly see that the samples after 200 iterations look far less like emojis than those after 5000 iterations. Indeed, the image on the right shows that the model has learned that emojis are rarely rectangular and often are yellow in colour with dots for eyes and a mouth, in contrast with the image on the left which looks much more like quasi-rectangles of noise.

Indeed, what has happened is that the generator has sampled examples from noise which it has tried to pass through the discriminator. As the number of iterations increase, both the discriminator and the generator get better at discriminating and generating respectively (as their loss function is a function of this adversarial task).

Note that, as the TA mentioned that we are allowed to slightly tweak the architecture, the above result is the best quality output I could get while remaining within the constraints detailed in the assignment handout.

Problem 2.1.1

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

Solution

See code in `cycle_gan.ipynb`.

Problem 2.2.1

Complete the `cyclegan_training_loop` function.

Solution

See code in `cycle_gan.ipynb`.

Problem 2.3.1

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.

Solution

After training the CycleGAN we get the following outputs:

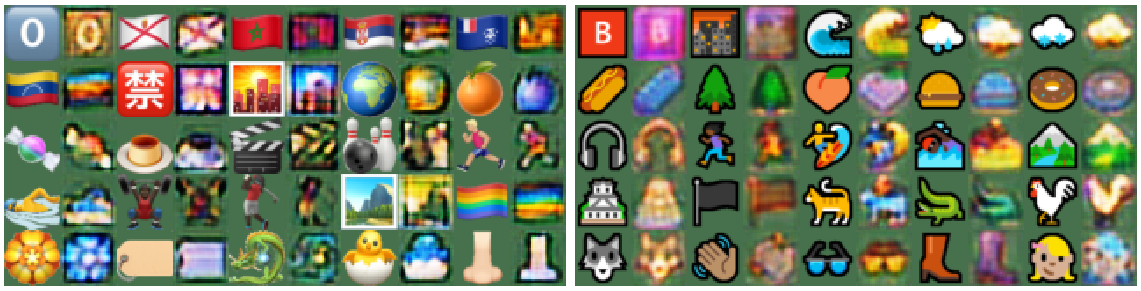

Figure 2: Apple to Windows (left) and Windows to Apple (right) emojis (200/5000 iter of CycleGAN)

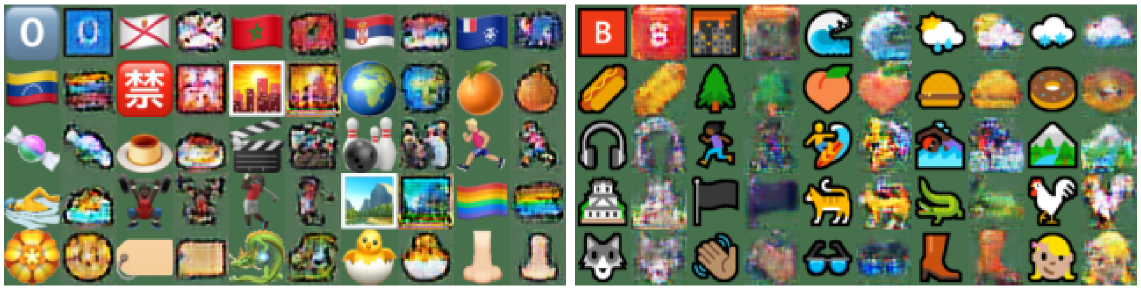

Figure 3: Apple to Windows (left) and Windows to Apple (right) emojis (5000/5000 iter of CycleGAN)

Problem 2.3.2

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

Solution

After changing the random seed from 4 to 2, this yields the following output:

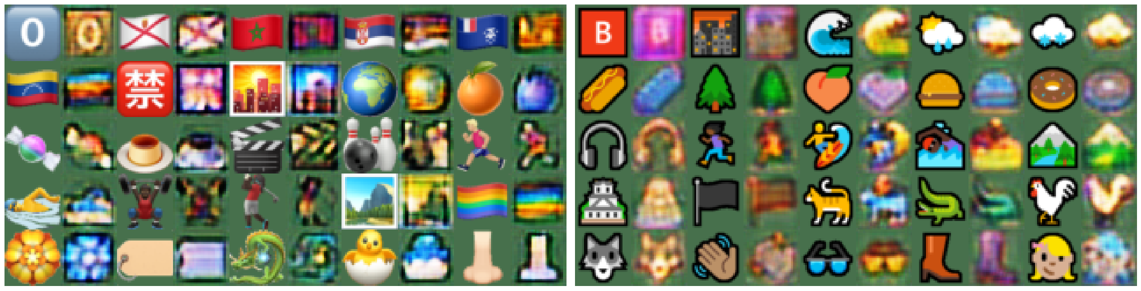

Figure 4: Apple to Windows (left) and Windows to Apple (right) emojis (200/5000 iter with seed 4)

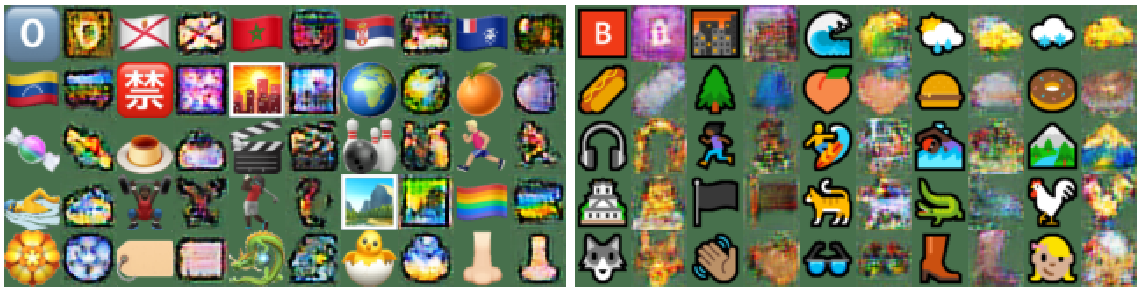

Figure 5: Apple to Windows (left) and Windows to Apple (right) emojis (5000/5000 iter with seed 2)

From figure 4 and figure 5 we can tell that changing the random seed simply changes the small details of the shapes/styles of the GAN-generated emojis (seed 4 images are a bit more smooth than seed 2). By contrast, unlike in figures 2 and 3, the overall colours do not change significantly when changing the seed. This is because the seed just means that iteration 5000 will have a slightly different generator loss (thus slightly different generation style and quality)

Problem 2.3.3

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?

Solution

After training with different values of the `lambda_cycle`, this yields the following outputs:

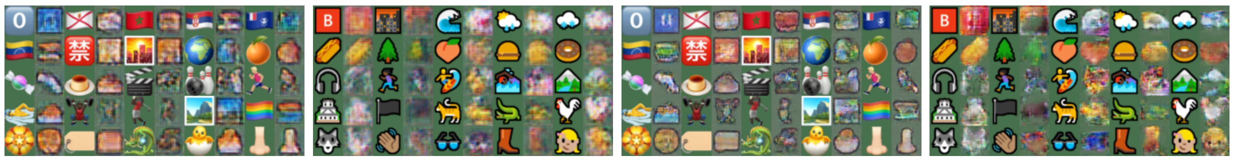

Figure 6: CycleGAN iter 200 (left pair) and 5000 (right pair), with seed 2 and `lambda_cycle` = 0.0000

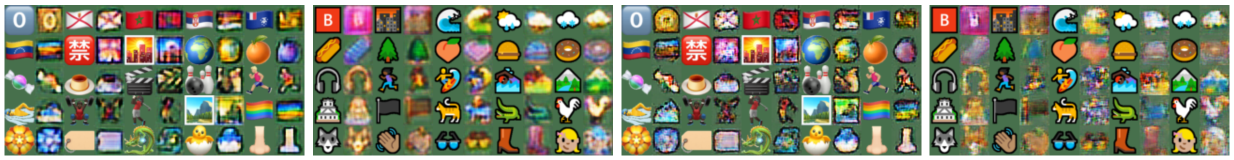

Figure 7: CycleGAN iter 200 (left pair) and 5000 (right pair), with seed 2 and `lambda_cycle` = 0.010

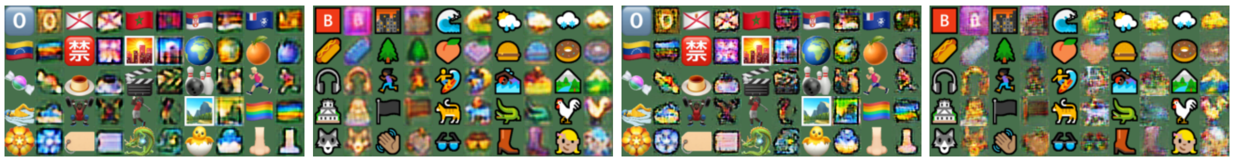

Figure 8: CycleGAN iter 200 (left pair) and 5000 (right pair), with seed 2 and `lambda_cycle` = 0.015

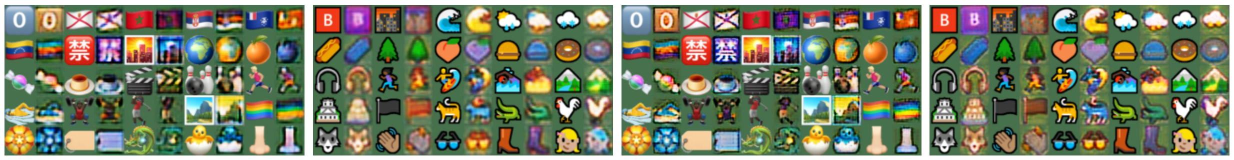

Figure 9: CycleGAN iter 200 (left pair) and 5000 (right pair), with seed 2 and `lambda_cycle` = 0.050

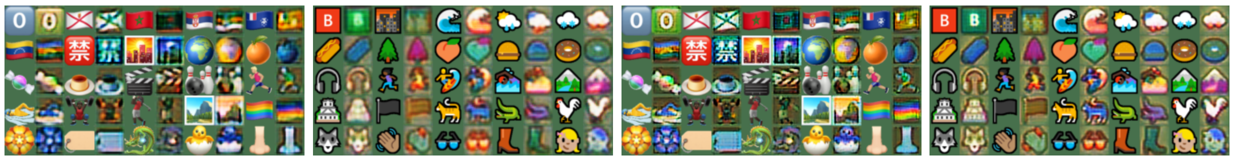

Figure 10: CycleGAN iter 200 (left pair) and 5000 (right pair), with seed 2 and `lambda_cycle` = 0.100

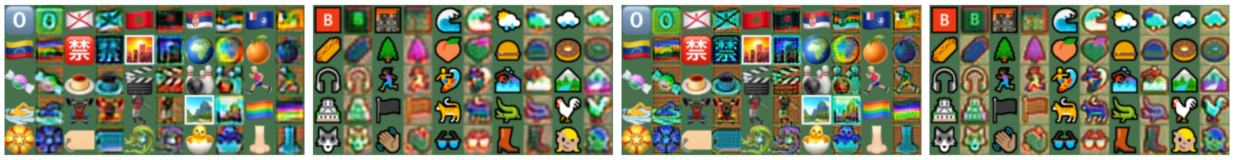

Figure 11: CycleGAN iter 200 (left pair) and 5000 (right pair), with seed 2 and `lambda_cycle` = 0.500

Thus we can notice from Figures 6 to 11 that `lambda_cycle` has a significant impact on the outputs. The best result of these experiments seems to be around `lambda_cycle` = 0.015.

Indeed, for `lambda_cycle` lower than 0.015, we observe worse style transfer as the images look almost nothing like the original image. The explanation behind this is that if `lambda_cycle` is too low, the loss will not be penalised enough for consistency so the network will generate style transfers very different from the original.

On the other hand, for `lambda_cycle` lower than 0.015, we observe worse style transfer as the images look almost identical to the original image but with different colours. The explanation behind this is that if the `lambda_cycle` is too high, the loss will be very highly penalised for consistency thus will lead to almost copies of the original shape constraining the network to mostly play with colour.