Programming Homework 4

Alexander Harnisch

1.1 Padding

With Kernel size K, stride S and input dimension D_x, we want to determine the required padding P to achieve the output dimension D_y. We have:

$$D_y = \frac{D_x - K + 2P}{S} + 1$$

Which we can simply rearrange to get

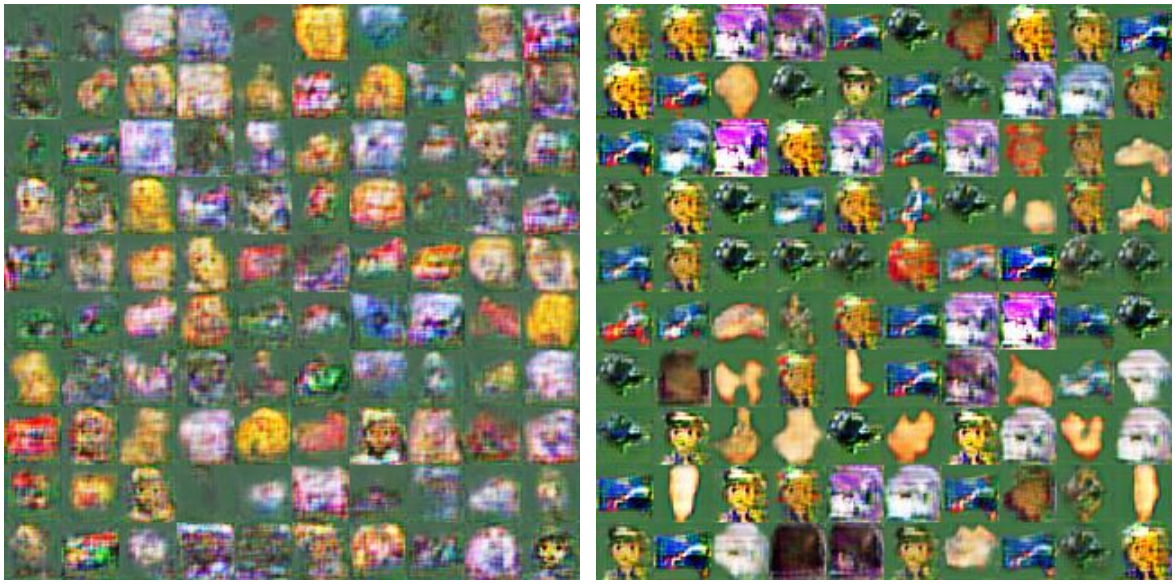
$$P = \frac{(D_y - 1)S + K - D_x}{2}.$$

Using that expression we find that the padding for the layers *conv1, conv2, conv3* and *conv4* should be: 2, 2, 2, 1, respectively.

1.4 DCGAN Experiments

1. Training the DCGAN

Here is a sample from early in training at iteration 1000 (left) and a sample from later in training at iteration 20000 (right):

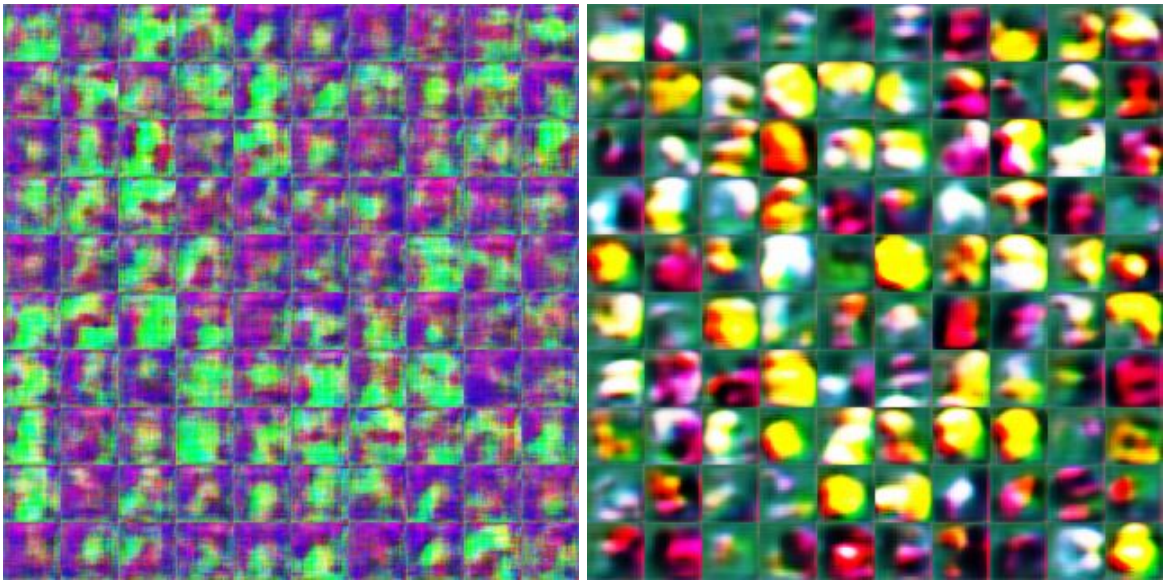


It is immediately obvious that the late training sample is of much higher quality than the early training sample. The images are less blurry and almost appear to resemble emojis.

Early on the GAN seems to generate a lot of rectangular shapes, while later in training it learns that emojis usually have a rounder shape. It also learned to draw emoji faces fairly well (e.g. the image in the lower left corner of the late training sample).

2. Gradient Penalty

Here is a sample from early in training at iteration 1000 (left) and a sample from later in training at iteration 20000 (right):



Looking at samples from different stages of training, the training does indeed appear more stable in terms of oszillation and mode collapse. However, the results appear to be of much lower quality than the previous model. This might simply be due to insufficiently tuned hyperparameters and too few training iterations.

Adding a gradient penalty to the loss function is an alternative way of a Lipschitz constraint, the simplest being gradient clipping, which requires careful hyperparameter tuning. The gradient penalty instead penalizes gradients with a norm largely different from 1 directly in the cost function itself. This might in some cases reduce or eliminate the need for hyperparameter tuning to enforce the Lipschitz constraint.

3. Changing other Hyperparameters

For all of the following, only the stated parameter has been changed from the default settings used in part 1 of this section. For all of them a sample from iteration 200 is shown on the left, and a sample from iteration 20000 on the right.

Spectral Norm Enabled

Learning Rate set to 0.00015 (half the default)

d_train_iters set to 5

None of the measures seem to significantly help with stabilizing the training and the final results are far from perfect. Especially mode collapse seems to be a severe issue with this architecture for all sets of hyperparameters I have tried.

However, in theory those measures can help, when the parameters are finely tuned. Applying the spectral norm technique can help with stabilizing the discriminator by normalizing its weight matrices. Lowering the learning rate can generally be helpful for almost any machine learning model, albeit very computationally expensive. Increasing the number of discriminator updates per generator update can be helpful in stabilizing the training by reducing the generators tendency towards oscillating between generating different samples from step to step.

2.5 CycleGAN Experiments

1. Training the CycleGAN

Iteration 400. Left: X→Y, Right: Y→X

Iteration 4000. Left: X→Y, Right: Y→X

Iteration 5000. Left: X→Y, Right: Y→X

2. Changing the Random Seed

After changing the random seed from 11 to 1234**:**

Iteration 5000. Left: X⟶Y, Right: Y⟶X

We can see a significant difference in terms of small details and color, compared to the previous random seed. While the general shapes remain largely the same, for me the color differences are the most striking.

The reason why a different random seed significantly impacts the final result is simple: It changes the entire training dynamic. In GAN training we don't simply minimize a fixed cost function whose landscape always looks the same, no matter the minimization approach. When training a GAN, the entire landscape changes from iteration to iteration. Instead of finding a minimum of a well defined function, we are looking to find a Nash equilibrium state between the two interacting networks (players). This process is inherently unstable, and changing the random noise (seed) significantly impacts the nature of the final equilibrium, or oscillation dynamic.

3. Changing the default lambda cycle hyperparameters

All of the following samples are from training iteration 5000. And for all of them X⟶Y (Apple to Windows) is the left image, and Y⟶X (Windows to Apple) the right one. The random seed used was 11.

lambda_cycle=0 at Iteration 400

lambda_cycle=0 at Iteration 5000

lambda_cycle=0.005 at Iteration 400

lambda_cycle=0.005 at Iteration 5000

lambda_cycle=0.010 at Iteration 400

lambda_cycle=0.010 at Iteration 5000

lambda_cycle=0.015 at Iteration 400

lambda_cycle=0.015 at Iteration 5000

lambda_cycle=0.025 at Iteration 400

lambda_cycle=0.025 at Iteration 5000

lambda_cycle=0.05 at Iteration 400

lambda_cycle=0.05 at Iteration 5000

lambda_cycle=0.1 at Iteration 400

lambda_cycle=0.1 at Iteration 5000

lambda_cycle=0.5 at iteration 400

lambda_cycle=0.5 at iteration 5000

lambda_cycle=1 at iteration 400

lambda_cycle=1 at iteration 5000

We can see a significant impact of the lambda cycle hyper parameter on the generated outputs of the CycleGAN. It appears like the best results are achieved for a value of around 0.015 (the default value in the provided template).

For values lower than 0.015, the generated images look almost nothing like the originals. The reason is simple: When the lambda cycle hyperparameter is too low, the loss does not penalise insufficient consistency enough, so the network will tend to generate samples vastly different from their originals, with almost no resemblance.

On the contrary, for large lambda cycle values, the penalty is so strong that the model has no freedom in the style transfer whatsoever, and essentially can't change the shape/pattern of the input images. The output images are almost copies of their inputs, only the colors are changed. This is very obvious in the last Figure, showing the result for a lambda cycle value of 1.