Barbara Butz Regina Gerber Patricia Windler

Cycle GAN

Task:

Our task was to implement the Cycle-Gan-Architecture from the paper "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks" (Zhu, Park, Isola, Efros). This architecture is used to transform between two image domains (image-to-image-translation). We chose the MNIST dataset (handwritten digits) and the SVHN dataset (street view house numbers) as image domains for our implementation, so our aim was to generate realistic looking svhn-images from mnist-inputs and vice versa.

Recap GAN:

A generative adversarial network consists of two artificial neural networks, namely a generator and a discriminator. The generator gets n random samples of a domain X (e.g. vectors or MNIST-images) as an input and strives to transform them into samples of a domain Y (e.g. SVHN-images). The discriminator gets the output of the generator together with n real samples of domain Y as an input and aims to classify all samples correctly, which means to label the generated ones with "0" and the real ones with "1". The objective of the generator on the other hand is to generate samples that are able to "fool" the discriminator (i.e. are labeled with "1" by the discriminator). Note that GANs do not rely on Input-Output-pairs as training data - no information is given about which sample of domain X matches which sample of domain Y. Thus GANs are very powerful and useful for tasks where paired training data might not be (easily) available.

Motivation for Cycle GAN:

In the lecture we learned that GANs can be used to generate new samples from any dataset they were trained on. So one might think two separate GANs are sufficient to transform between two (image-)domains; one could be trained to transform samples of domain X to domain Y and the other to transform samples from domain Y to domain X.

While this might work in some cases, often the problem of mode collapse arises. Mode collapse occurs when a GAN maps all input samples to the same output sample and the optimization thus fails to make progress. This is possible because only the discriminator loss is taken into account when training a "classical" GAN. This loss (also called adversarial loss or GAN-loss) only indicates the similarity between the generated samples and the real samples of the output domain but not the similarity between an input sample and the corresponding generated output sample. (Simplified: if a generator created one output sample which was able to fool the discriminator, it might not have an incentive to create a different sample for other inputs.)

But even without taking the possibility of mode collapse into account, two separate GANs might not be the best solution for the task of transforming between two image domains, since a meaningful relation between the input and output image is often desired for practical applications. Zhu et al. used the Cycle-Gan architecture for example to transform between photos and Monet-paintings. The desired meaningful relation here is the content, which should stay the same between input and output while only the style changes. (I.e. a photo of a harbour should be mapped to a monet-style-painting of the same harbour and not to e.g. a monet-style-painting of a mountain.)

However, as explained above, a classical GAN has no way to measure (and therefore optimize) the similarity between an input sample and an output sample. Therefore an extension of said classical GAN is needed to ensure a meaningful relation between the input image of one domain and the generated output image of the other domain.

Introduction to Cycle GAN:

A Cycle GAN contains two generators: A generator G, striving to translate images of domain X (in our case MNIST-images) to images from domain Y (in our case SVHN-images) and a generator F, striving to to translate images of domain Y to images from domain X. Additionally, it comprises two discriminators; a discriminator D_Y, which encourages G to produce outputs indistinguishable from domain Y and a discriminator D_X, which encourages F to produce outputs indistinguishable from domain X.

So up to here, the Cycle GAN seems to consist of two separate GANs: G and D_Y forming the first one and F and D_X forming the second one. To connect the two GANs, eliminate the problem of mode collapse and encourage a meaningful connection between input and output image, Zhu et al. use the concept of cycle-consistency. This means that if an image is translated from one domain to the other and back, the output image should be (roughly) the same as the input image. I.e.
$x \rightarrow G(x) \rightarrow F(G(x)) \approx x$ (forward cycle-consistency) and $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$ (backward cycle-consistency). To ensure cycle-consistency, a cycle-consistency-loss is introduced (see below).

Architecture and Implementation (according to Zhu et al.):

In the following, we will describe the architecture and (some of the) implementation as devised by Zhu et al. . Our sources are the paper itself as well as the PyTorch implementation by the authors of the paper (Links below, see “Sources”).

For the generators, Zhu et al. proposed the following architecture:
- a 7 × 7 Convolution-InstanceNorm-ReLU layer with 64 kernels and stride 1
- a 3 × 3 Convolution-InstanceNorm-ReLU layer with 128 kernels and stride 2
- a 3 × 3 Convolution-InstanceNorm-ReLU layer with 256 kernels and stride 2
- several residual blocks[1], each containing two 3 × 3 convolutional layers with 256 kernels on both layers. The number of residual blocks depends on the image size: 6 blocks for 128x128 images, 9 blocks for 256x256 images
- a 3 × 3 fractional-strided-Convolution-InstanceNorm-ReLU layer (i.e. backward convolutional layer) with 128 kernels and stride 1/2
- a 3 × 3 fractional-strided-Convolution-InstanceNorm-ReLU layer with 64 kernels and stride 1/2
- a 7 × 7 Convolution-InstanceNorm-tanh[2] layer with 3 kernels and stride 1 (or with 1 kernel, if a grayscale-output is desired, see PyTorch implementation)

[1] A residual block is basically a layer consisting of two “inner” convolutional layers. The output of a residual block not only feeds into the next layer/block but into the next three or four layers/blocks (skip-connections). This is useful to eliminate some problems that usually arise when a network consists of too many layers (e.g. vanishing gradients). If a network consisting of residual blocks has more layers than it needs to successfully perform the desired task, it can be trained make use of the skip-connections and skip some layers. The skip-connections are realised by applying the activation function after the second convolutional layer not only to the output of said layer, but to the output of

the second layer plus the input of the residual block (see helper function for residual blocks in our code for a better understanding).
[2] According to the paper, this is a Convolution-InstanceNorm-ReLU layer. But in the PyTorch implementation, tanh was used as an activation function and since - in our opinion - this makes more sense for the last layer, we assume this was a mistake in the paper.

The discriminators consist of the subsequent layers:
- a 4 × 4 Convolution-InstanceNorm-LeakyReLU layer with 64 filters and stride 2
- a 4 × 4 Convolution-InstanceNorm-LeakyReLU layer with 128 filters and stride 2
- a 4 × 4 Convolution-InstanceNorm-LeakyReLU layer with 256 filters and stride 2
- a 4 × 4 Convolution-InstanceNorm-LeakyReLU layer with 512 filters and stride 2
- a convolutional layer with activation function tanh to produce a 1-dimensional output

Together they form a so called “PatchGAN”. A PatchGAN classifies small overlapping patches of the image into the categories “fake” or “real”, instead of the whole image at once. The classification result is then the rounded average classification of all patches.

For training the networks, Zhu et al. used different kinds of losses, depending on the task and the type of network. The discriminators only seek to minimize a GAN loss (least-squares-loss):

$$\boldsymbol{E_{y\sim p_{data}(y)}[(D(y)-1)^2] + E_{x\sim p_{data}(x)}[D(G(x))^2]}$$

I.e. the aim of the discriminators is to minimize the number of real images classified as generated (first part of the formula) as well as the number of generated images classified as real (second part of the formula). In short: the discriminators aim to minimize the number of wrongly classified images. (Note: the discriminator loss is divided by two to slow the rate at which the disrciminators learn, in relation to the generators.)
The generators on the other hand seek to minimize the weighted sum of a GAN loss and a cycle consistency loss. The GAN loss of the generators is also a least-squares-loss, but the formula differs slightly from the GAN loss of the discriminators:

$$\boldsymbol{E_{x\sim p_{data}(x)}[(D(G(x))-1)^2]}$$

The reason for the difference is that the generators “do not care” how the discriminators classify the real images - their only aim is to “fool” the discriminators with their generated output, which means to minimize the number of generated images classified as generated.
The cycle consistency loss is defined as follows:

$$\boldsymbol{L_{cyc}(G,F) = E_{x\sim p_{data}(x)}[\|F(G(x))-x\|_1] + E_{y\sim p_{data}(y)}[\|G(F(y))-y\|_1]}$$

I.e. the sum of the absolute difference of an input image x and the same input x transformed to domain Y and back to domain X plus the absolute difference of an input image y and the same input y transformed to the domain X and back to domain Y.
As stated above, the generators are trained using a weighted sum of the GAN loss and the cycle consistency loss. The parameter λ controls the weight of the cycle consistency loss in relation to the GAN loss. Zhu et al. set λ=10 in their implementation, which means that the cycle consistency loss is ten times as important as the GAN loss for training the generators.

Furthermore, an identity loss ($L_{\text{identity}}(G,F) = E_{y\sim p_{data}(y)}[\|G(y) - y\|_1] + E_{x\sim p_{data}(x)}[\|F(x) - x\|_1]$) is proposed for applications, where the color composition between output and input should be preserved (e.g. transforming between photos and paintings).

Zhu et al. train their networks for 200 epochs from scratch. They use the adam optimizer with a batch size of 1, a beta1-value of 0.5 and a learning rate of 0.0002, which is kept for the first 100 epochs and then linearly decayed to zero over the last 100 epochs.

Our implementation:

For our implementation we used the generator architecture with six residual blocks, because Zhu et al. used the network with nine blocks only for images with a resolution of 256 x 256 pixels or higher, which is significantly bigger than our images are - MNIST images have a resolution of 28 x 28 while SVHN images have a resolution of 32 x 32 pixels.

For the implementation of the forward and backward convolutional layers of the generators and discriminators we used the helper functions defined in the sample solution of Homework 8 - with one small change: We added a parameter for the padding, since in our implementation, not all layers needed same-padding. Sometimes reflection-padding had already been applied, in which case the respective layer needed no further padding (i.e. valid-padding needed to be applied). We also used the provided helper functions for flattening and batch normalisation. For creating residual blocks we defined an additional helper function which in turn uses the helper function for creating convolutional layers.

While we used the mentioned helper functions to avoid some duplicate code, we decided against writing a function for creating the generators and discriminators (which would have reduced the amount of duplicate code even more), because we wanted to be able to change the networks individually without much hassle (e.g. for accounting for the size difference between MNIST and SVHN images, see below).

We did not use an identity loss, because we did not have a need to preserve the color composition between input and output of the generators. (Sidenote: Not only would it not have made sense to preserve color composition in our case - it would have been impossible, because the MNIST dataset consists of grayscale images, while the SVHN dataset contains RGB images.)

We strived to stay as similar as possible to the original implementation and frequently consulted the PyTorch implementation - especially models/cycle_gan_model.py and models/networks.py (in the respective github repository, see link below) - to achieve this goal. Nevertheless, we had to make some changes, mainly due to the restrictions of our computational power:

First of all, Zhu et al. proposed to train the discriminators using not one image at a time, but a buffer of 50 previously generated image to reduce model oscillation. While we understand the usefulness of such a buffer in theory, our program already has a very high runtime while we just feed the generators and discriminators one image in each training step, respectively. (see below under “Evaluation”) Feeding each discriminator 50 images in every training step would have prolonged the already high runtime exponentially. Thus we did not include the image buffer in our implementation.

Furthermore we had to make some small changes to the generators to account for the size differences between the MNIST images (28 x 28 px.) and the SVHN images (32 x 32 px.). In the first layer of generator *G* we used a reflection padding of 5 instead of 3 to artificially enlarge the MNIST-input. In the last layer of generator *F* on the other hand, we used a reflection padding of 1 instead of 3 to cut the MNIST-output to the correct size.

Evaluation

Unfortunately, we didn't manage to consistently generate realistic looking MNIST or SVHN images (see example_images - folder). In one test run, we managed to generate somewhat realistic looking MNIST output; the central number (2) was clearly recognizable, and the output indeed looked like handwriting, but the adjoining ciphers from the housenumber were still visible (see images_good_mnist/mnist).

The generated SVHN images never looked remotely real - usually black shapes (similar to the MNIST input) on a grey background with some red, green, and blue pixels in a regular pattern on top were generated (see images_good_mnist/svhn).
We have some theories on why we weren't sucessful:

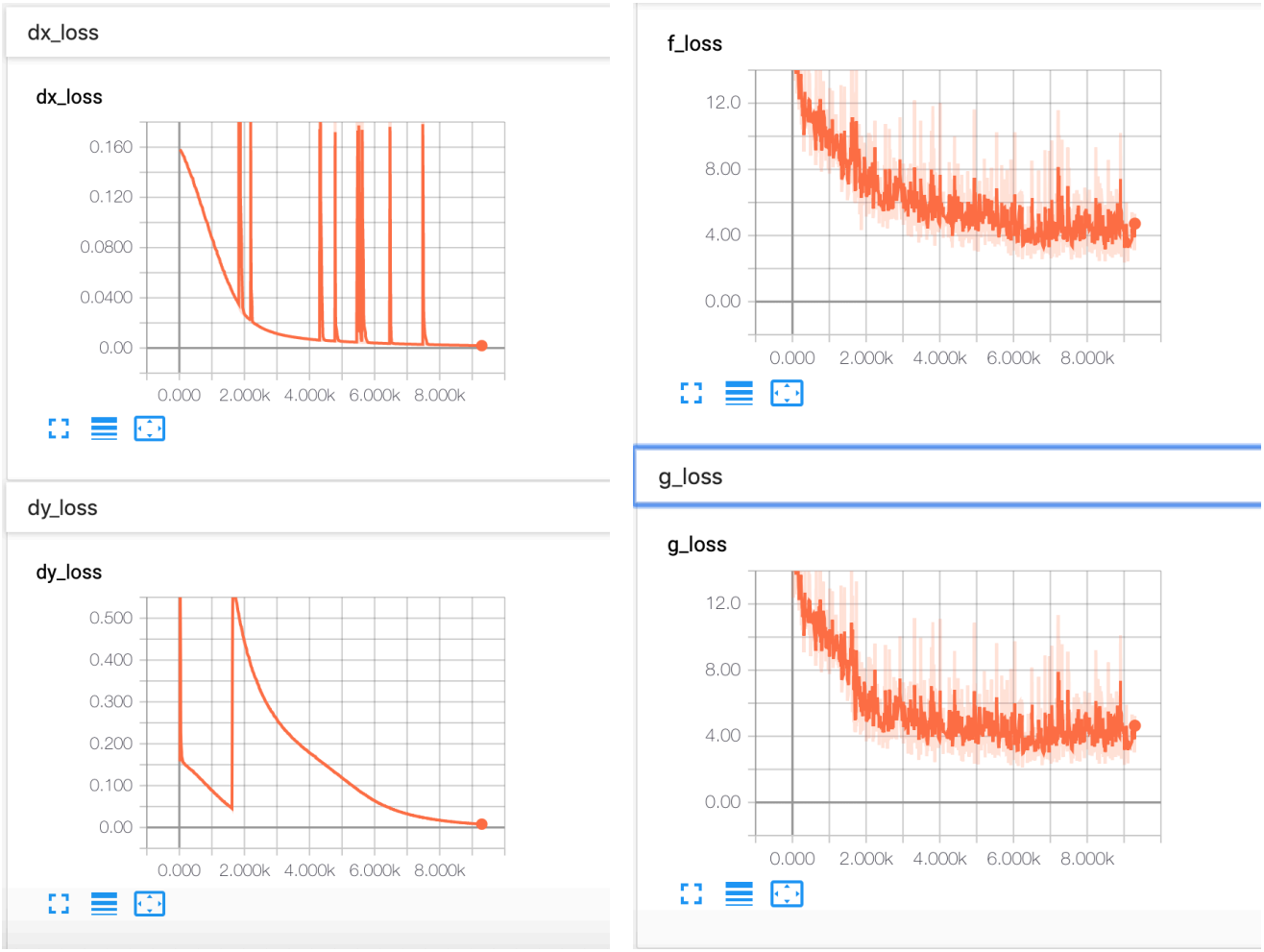
Firstly, we thought that the size difference (28 x 28 vs. 32 x 32) or the difference in color distribution (grayscale vs. rgb) might have been a problem. We rejected this theory pretty quickly - the size difference shouldn't be a problem, as the network should definitely be able to learn to transform between "framed"/padded MNIST-images and SVHN images. The difference in color shouldn't be a problem either - the PyTorch implementation accounts for RGB output as well as grayscale output; the relevant functions only need to be called with differing numbers of kernels for the last layer of the generator (1 for grayscale, 3 for RGB).

Secondly, due to the restrictions of our computational power, we were unable to run our program for the proposed 200 epochs. With the full training data sets (60.000 MNIST images and 73.257 SVHN) we weren't even able to run the program for one epoch, since 1/10 of an epoch had a runtime of roughly half an hour. With only 1000 images from each dataset, we were able to run the program for four epochs, which was an improvement but still nowhere near the proposed 200 epochs. Since the reduction of the dataset did not cause a significant improvement but rather a slight deterioration of the quality of the output (after the same number of training steps), we decided to work with the full datasets again. To put it all into a nutshell, we might not have been able to run enough epochs/training steps for the transformations to be successful. (Sidenote: Luke, if you run our program on the grid - please tell us about your findings/ the output!)

Thirdly, we learned in the lecture that training GANs might be difficult, since the classifications of the discriminators must neither be too good nor too bad in relation to the generators. In both cases, the generator would not have an incentive to create more realistic images anymore. So we thought that maybe our discriminators were learning too fast or too slow. The Tensorboard-visualisation seems to undermine the theory of too-fast learning discriminators (from folder example_summaries, see below). Unfortunately, we weren't able to find the right learning rate for the task of MNIST-SVHN-transformation.

Lastly, the content of the images of our two datasets might simply be too (geometrically) different from each other to be suitable for Cycle-Gan-based transformations. Zhus et al. attempt to transform

between images of Cats and Dogs failed as well, which led them to the conclusion that the generators, tailored for good performance on appearance changes, might be unable to handle more varied and extreme transformations. And while cats and dogs are arguably more geometrically complex than numbers, maybe the comparatively small geometrical differences between the MNIST numbers and the SVHN numbers were already too big for the Cycle GAN architecture.



Sources:

- Isola, P., Zhu, J., Zhou, T., & Efros, A. (2016). Image-to-Image Translation with Conditional Adversarial Networks. arXiv.org. Retrieved 17. March 2019, from https://arxiv.org/abs/1611.07004
- Cycle GAN project page: https://junyanz.github.io/CycleGAN/
- Cycle GAN implementation with PyTorch: https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix
- https://towardsdatascience.com/residual-blocks-building-blocks-of-resnet-fd90ca15d6ec
- sample solution to Homework 8