Deep Learning for Medical Imaging

Zhiyuan Hu
Maurice Kraus
Maxim Kuznetsov
Kevin Huy Nguyen

TECHNISCHE UNIVERSITÄT DARMSTADT

FB 20 Informatik
Graphisch-Interaktive Systeme

Contents

1 Week 01: Planning **2**

1.1 Motivation for choosing architecture 2

2 Week 02: Implementation 1 **3**

2.1 Work about loading the data 3

3 Week 03: Implementation 2 **5**

3.1 First prototype 5

4 Week 04: Implementation 3 **8**

4.1 Extensions to prototype, debugging. 8

5 Week 05: Implementation 4 **9**

5.1 Main work 9
5.2 WGAN-GP 9
5.3 Higher resolution 10
5.4 condition 10

Bibliography **11**

1 Week 01: Planning

1.1 Motivation for choosing architecture

Motivation for choosing architecture: For our initial architecture we choose DC-GAN, because our task is to generate new images based on a given dataset. We believe that the structure of having a generator and discriminator would lead to the results we want. For the training, we would like to begin from WGAN-GP(Wasserstein GAN with gradient penalty) as training method, which gives a significant improvement by stability of training. So far we tested different GAN variants (vanilla GAN, WGAN and WGAN-GP) on the MNIST Dataset. WGAN-GP showed a prominent improvement in the outcome.

As to VAE, VAE tends to generate imagines that somehow blurry. We test the example code of pyro. So we have not chosen it.

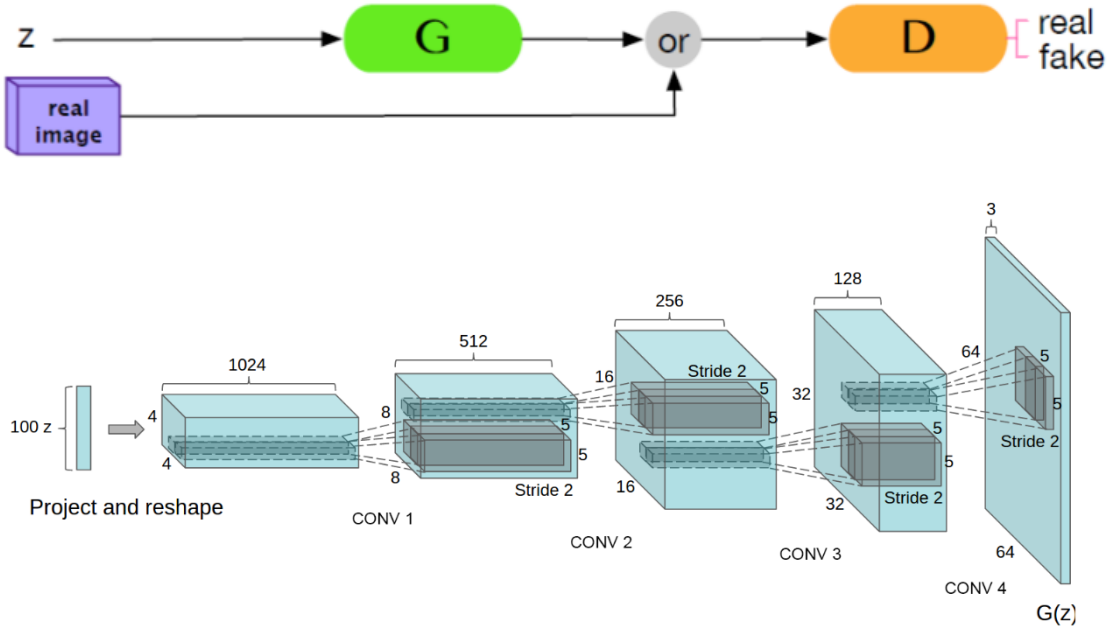
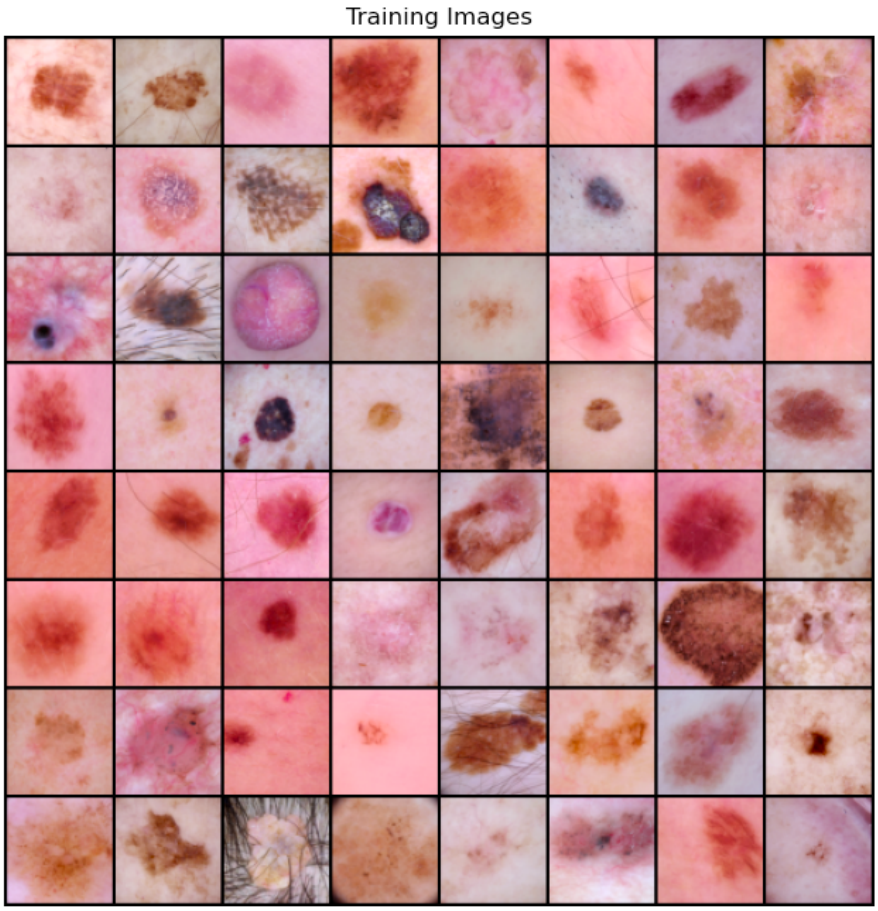


Figure 1.1: Basic architecture as presented in the lecture

2 Week 02: Implementation 1

2.1 Work about loading the data

As you can see in the code snippet below, we implemented the data loader. In ??, we plotted 64 randomly picked images. While looking at the metadata, we noticed that for some of the data points, certain attributes were missing. Some images did not have the age of the patient or the analytical method used to find about the type of skin lesion. For those we have to think about how to fill in the missing attributes in the future. Looking into the metadata, we also thought about generating skin lesion images from certain attributes, e.g. a skin lesion image of a patient of a certain age. But because this is not the main focus for now, we postponed that thought and will come back to it later, when we have a working prototype. Lastly, we created a Github repository to maintain our code at one place as we also have written the boilerplate code for the DC-GAN.




```python
import copy
import glob
import logging

import torch
import torch.cuda
from PIL import Image
from torch.utils.data import Dataset
from torchvision.transforms import transforms

from dataloader.process_meta_data import get_lesion_infos

log = logging.getLogger(__name__)
# log.setLevel(logging.WARN)
# log.setLevel(logging.INFO)
```

```python
log.setLevel(logging.DEBUG)


class SkinLesionDataset(Dataset):
    def __init__(self,
                 val_stride=0,
                 is_validation=None,
                 lesion_id=None,
                 transform=None,
                 ):
        if transform:
            self.transform = transform
        else:
            self.transform = transforms.ToTensor()

        self.lesion_infos = copy.copy(get_lesion_infos())

        self.jpgs_paths = glob.glob('/usr/src/rawdata/*.jpg')

        if lesion_id:
            self.lesion_infos = [
                x for x in self.lesion_infos if x.lesion_id == lesion_id
            ]

        if is_validation:
            assert val_stride > 0, val_stride
            self.lesion_infos = self.lesion_infos[::val_stride]
            assert self.lesion_infos
        elif val_stride > 0:
            del self.lesion_infos[::val_stride]
            assert self.lesion_infos

        log.info('{!r}: {} {} samples'.format(
            self,
            len(self.lesion_infos),
            'validation' if is_validation else 'training',
        ))

    def __len__(self):
        return len(self.lesion_infos)

    def __getitem__(self, idx):
        lesion_infos_tup = self.lesion_infos[0][idx]
        image_path = ''
        for path in self.jpgs_paths:
            image_id = path.split('/')[-1].split('.')[0]
            if image_id == lesion_infos_tup.image_id:
                image_path = path
                break

        if len(image_path) == 0:
            log.error('{!r}: {} was not found in dataset'.format(
            self,
            lesion_infos_tup.image_id))

        image = Image.open(image_path).convert('RGB')
        tensor_image = self.transform(image)
        return tensor_image, torch.tensor(lesion_infos_tup[2:]),
```

3 Week 03: Implementation 2

3.1 First prototype

We have implemented the first DCGAN architechture on HAM10000. The architechture is defined as follows. The result looks like figure 3.1. As mentioned before we will try the Wasserstein-GAN next, but didn't have the time for it yet unfortunately. Below you can also see some code excerpts of our current prototype. For the generator we have a random seed as input and transform it to an image using 5 strided two dimensional convolutional transpose layers, each paired with a 2d batch normalization layer and a relu activation. At the end we feed the result into a tanh function to get the values to be between 0 and 1. For the discriminator we have an image as input and 5 strided two dimensional convolutional layers, each paired with a 2d batch normalization layer and a leaky relu activation. At the end we feed the result to a sigmoid function to get a probability value for our input image. For now we are using the Adam optimizer with BCE-loss function and a learning rate of 0.0003. Figure 3.1 shows the generated images at epoch 109. The following hyperparameters were used:

- learning rate: 3e-4
- batch size: 64
- num features: 64
- image size: 64
- noise vector size: 64

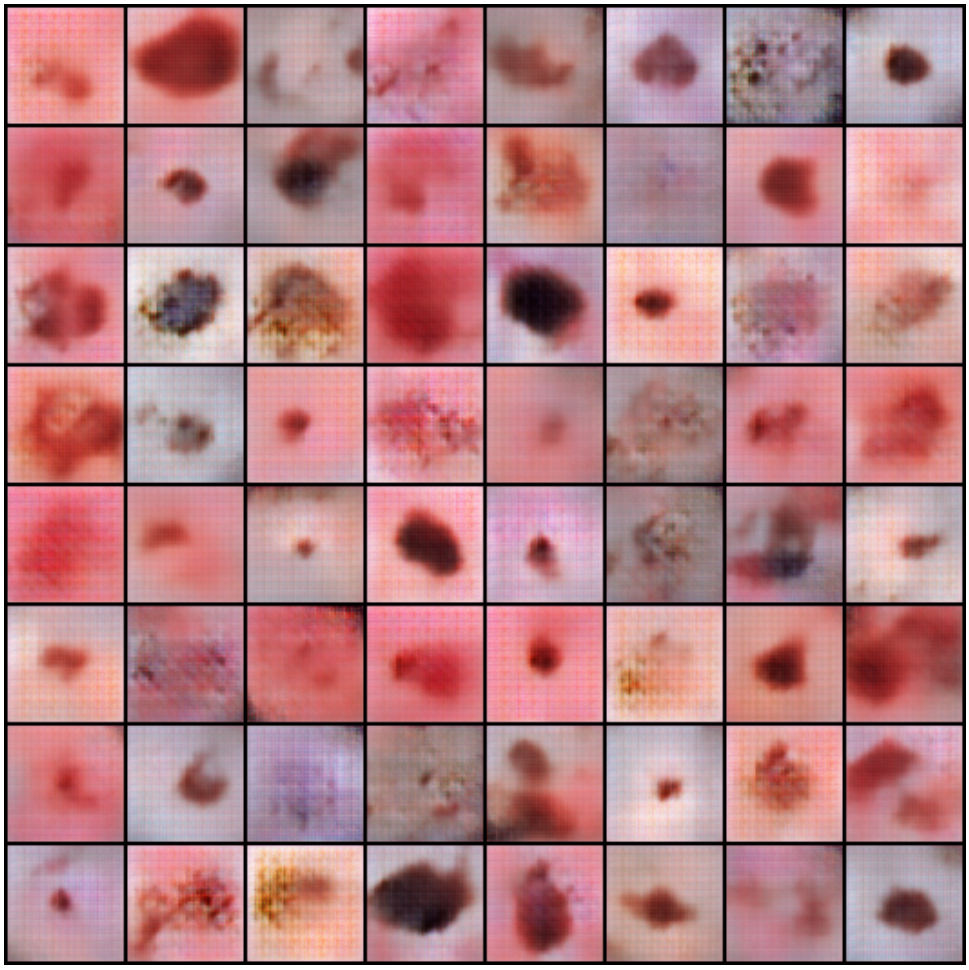


Figure 3.1: First prototype output after 109 epochs

```
import torch.nn as nn
```

```python


class Discriminator(nn.Module):
    def __init__(self, image_channels, features_d, ngpu=1):
        super().__init__()
        self.ngpu = ngpu
        self.disc = nn.Sequential(
            # input dims: N x 3 x 64 x 64
            nn.Conv2d(
                image_channels, features_d, kernel_size=4, stride=2, padding=1
            ),
            nn.LeakyReLU(0.2),
            # _block(in_channels, out_channels, kernel_size, stride, padding)
            self._block(features_d, features_d * 2, 4, 2, 1),
            self._block(features_d * 2, features_d * 4, 4, 2, 1),
            self._block(features_d * 4, features_d * 8, 4, 2, 1),
            # Conv2d down to 1x1
            nn.Conv2d(features_d * 8, 1, kernel_size=4, stride=2, padding=0),
            nn.Sigmoid(),
        )

    def _block(self, in_channels, out_channels, kernel_size, stride, padding):
        return nn.Sequential(
            nn.Conv2d(
                in_channels,
                out_channels,
                kernel_size,
                stride,
                padding,
                bias=False,
            ),
            nn.BatchNorm2d(out_channels),
            nn.LeakyReLU(0.2),
        )

    def forward(self, x):
        if x.is_cuda and self.ngpu > 1:
            output = nn.parallel.data_parallel(self.disc, x, range(self.ngpu))
        else:
            output = self.disc(x)
        return output.view(-1, 1).squeeze(1)


class Generator(nn.Module):
    def __init__(self, noise_vector, image_channels, features_g, ngpu=1):
        super().__init__()
        self.ngpu = ngpu
        self.gan = nn.Sequential(
            # input dims: N x noise_vector x 1 x 1
            self._block(noise_vector, features_g * 16, 4, 1, 0), # img: 4x4
            self._block(features_g * 16, features_g * 8, 4, 2, 1), # img: 8x8
            self._block(features_g * 8, features_g * 4, 4, 2, 1), # img: 16x16
            self._block(features_g * 4, features_g * 2, 4, 2, 1), # img: 32x32
            nn.ConvTranspose2d(
                features_g * 2, image_channels, kernel_size=4, stride=2, padding=1
            ),
            # output dims: N x 3 x 64 x 64
            nn.Tanh(),
        )

    def _block(self, in_channels, out_channels, kernel_size, stride, padding):
        return nn.Sequential(
            nn.ConvTranspose2d(
                in_channels,
```

```
            out_channels,
            kernel_size,
            stride,
            padding,
            bias=False,
        ),
        nn.BatchNorm2d(out_channels),
        nn.ReLU(),
    )

    def forward(self, x):
        if x.is_cuda and self.ngpu > 1:
            output = nn.parallel.data_parallel(self.gan, x, range(self.ngpu))
        else:
            output = self.gan(x)
        return output


def init_weights(model):
    """
    Initialize weights based on DC-GAN paper
    :param model:
    """
    for m in model.modules():
        if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d, nn.BatchNorm2d)):
            nn.init.normal_(m.weight.data, 0.0, 0.02)
```

4 Week 04: Implementation 3

4.1 Extensions to prototype, debugging.

This week we have tried two different ideas: One was to fine-tune the existing prototype to return better results than last time and the other idea was to create a W-GAN.
For the first we adjusted the learning rate several times and tried to run the learning longer, so more than 109 epochs. We also tried to change the batch sizes to 16, 32, and 64. These measures sadly did not yield significantly better results. Next on the list is to try to train on the full resolution of the dataset instead of just using the lowered resolution and cropped dataset. Also we want to test out different input noise vector sizes.
For the second idea, we tried the W-GAN, which was already implemented from one of our members for a past project. That implementation worked in MNIST, but applied to our dataset, it did not perform. Currently we are trying out different hyperparameters and are trying to figure out the reason for the W-GAN not performing. Another idea was to try the W-GAN with gradient penalty. Hopefully we have better results to report for next week.

5 Week 05: Implementation 4

5.1 Main work

This week the works are mainly concentrated following points:

* Improvement in training technique-> wasserstein GAN with gradient penalty(WGAN-GP) TTUR(discriminator uses higher learning rate)

* Higher resolution -> use deeper network to generate higher resolution test MSG-GAN

* Add condition to control the output

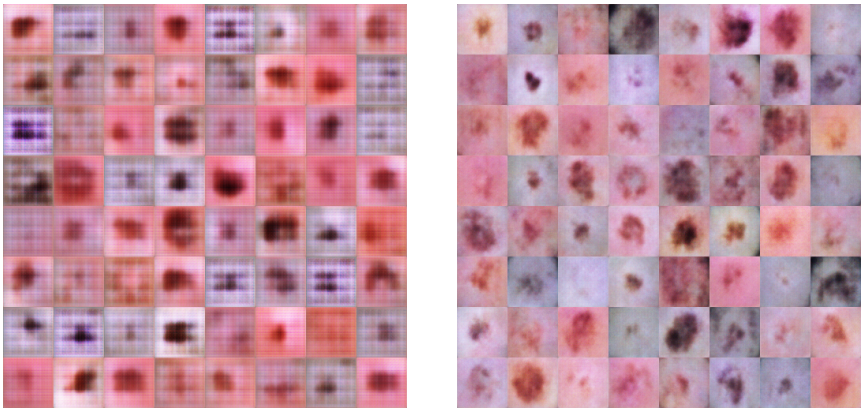
* data augmentation, (random horizontal/vertical flip, random crop with random size and then resize)

* help features, e.g. checkpoint and recover from checkpoint, speed up(reduce for loop while loading data and some other calculation), more friendly argparse operation, summary the network(torchsummaryX)

5.2 WGAN-GP

the calculation of wgan-gp follows the famous paper of "Improved Training of Wasserstein GANs", the change in prior DCGAN is two points: remove the last layer sigmoid activate in discriminator, use the loss. The formula of new loss is as $D_w(\tilde{x}) - D_W(x) + \lambda(\|\nabla_{\hat{x}}(D_w(\hat{x}))$ with $\hat{x}$ is created from $\hat{x} = \epsilon x + (1-\epsilon)\tilde{x}, \epsilon \sim \mathrm{U}(0,1)$. With the help of WGAN-GP, the training process is highly speeded up. We showed a example at figure 5.1. We can see a significant improvement. Just in the 4th epoch, the DCGAN can learn to generate some figure, although it is still very blurry, while DCGAN still doesn't learn.
Our code is implemented as follows, significant improvement is at line 2, it calculate batch data's gradient efficiently.

```
1 # gradient penalty
2 grad_p_x = torch.autograd.grad(p_mix.sum(), data_mixed, retain_graph=True, create_graph=True)[0]
3 # p_mix.sum(), trick to cal \par y_i / \parx_i independentl
4 assert grad_p_x.shape == data_mixed.shape
5 grad_norm = torch.sqrt(grad_p_x.square().sum(axis=(1, 2, 3)) + 1e-14)
6 loss_2 = self.gp_lambda * torch.square(grad_norm - 1.)
```



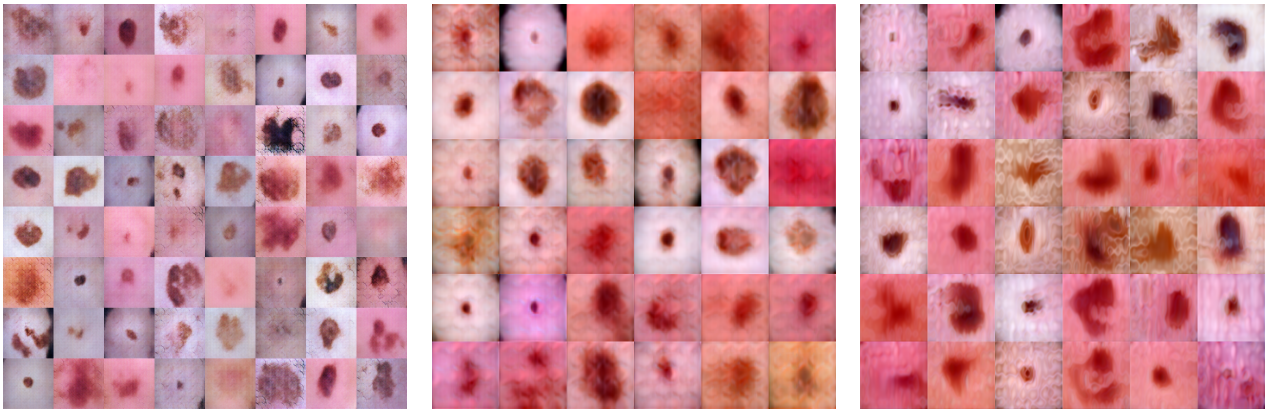
(a) DCGAN at step4 (b) WGAN at step 4

Figure 5.1: Camparison of DCGAN and WGAN-GP

5.3 Higher resolution

The standard DCGAN uses five layers and every layer uses 4*4 conv/convtrans kernel. We enlarge the depth to try generate the result i.e. we add more layers in the hidden layer with 4*4 conv/conv_trans to enlarge the depth. So we can control the generated figure's resolution. Depth 5 -> 64*64, Depth 6 -> 128*128, Depth 7 -> 256*256. And so on. We showed a result for 128x128 in figure 5.2(a).

In the training, 256*256 is still acceptable(90M params, 13.7G Mult-adds), but 512*512 is too huge for our training. Insteadly, we tried MSG-GAN, (Multi Scale Gradient GAN), which is a very interesting work. We use the author's code, (https://github.com/akanimax/BMSG-GAN). However, it requires too huge calculation resource and takes so long time. The result is still not very good. Although we tried in 4 RTX2080Ti with 100% data, and 1 GTX1080 with 60% data and trained for 2 days. But we think MSGGAN should work, since we tried a smaller MSGGAN network with depth 5 (generate from 4*4 to 64*64) for 179 epoch with roughly 30% HAM10000 dataset, it learns something, although still very unclear. The results are showed in figure 5.2



(a) W-DCGAN with GP, 128*128 resolution (b) MSG GAN for depth 5(max resolution 64*64) (c) MSG GAN for depth 7(max resolution 256*256)

Figure 5.2: Some result of MSGAN

5.4 condition

We add disease type as condition for training the network.

Firstly, add embedding layer in both generator and discriminator for label disease type.

Secondly, for generator, we concatenate the noise and embedded label as input.

Thirdly, for discriminator, originally the last layer should be conv-kernel with 4*4 size and 1 output channel so as to make sure that the output will be (batch_ size, 1), we change the kernel as (n_input_channel, output_channel=64, size=4*4). So the output would be (batch_size, 64). Then similarly to generator, we concatenate the output and embedded label and feed into a new linear layer.

The result is shown in figure 5.3. However, due to the time and calculation resource, we have not well trained the result. Every line is one class, although not well trained, we can see that truly there is different style between different lines.

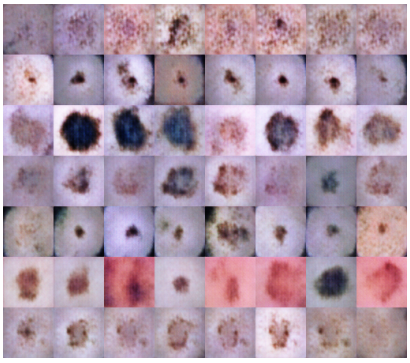


Figure 5.3: conditional W-DCGAN with gp

Bibliography