Automatic Image Generation Report

Lauren S. R. Sampaio, École Nationale Supérieure d'Ingénieurs de Caen 19/02/2020

Exercise 1 - Traditional implementation

The objective of this first exercise was to compare the PyTorch implementation of the ReLu function with a traditional implementation. The Listing 1 shows the traditional implementation functions, as asked by the topics 1.1 (ReLU function), 1.2 (backward of ReLU) and 1.3 (backward of the layer).

The ReLU function has the following mathematical definition:

$$f(x) = max(0, x) \tag{1}$$

Its derivative is given by

$$f'(x) = \begin{cases} 0, & \text{if } x \leq 0 \\ 1, & \text{otherwise} \end{cases} \tag{2}$$

Listing 1: ReLU functions implementation.

```
1  import torch
2  # EXERCISE 1.1: Implement ReLU forward and backward
3  def my_ReLU(x):
4      x_c = x.clone()
5      x_c[x_c<=0] = 0
6      return x_c
7
8  # EXERCISE 1.2: Backward of ReLU
9  def ReLU_backward(x,dzdy):
10     # dydx is 1 x>0 and 0 x<=0
11     dzdy_c = dzdy.clone()
12     dzdy_c[x<=0] = 0
13     dzdx = dzdy_c
14     return dzdx
15
16 # EXERCISE 1.3: Implement backward of matrix multiplication w.r.t. x
17 def mm_backward(A,x,dzdy):
18     dydx = A.t()
19     dzdx = torch.mm(dydx,dzdy)
20     return dzdx
```

After executing the testing segment of the code, we obtained the exact gradient vector for both implementations, which leads to the conclusion that the traditional implementation is correct. We can verify the result in Figure 1.

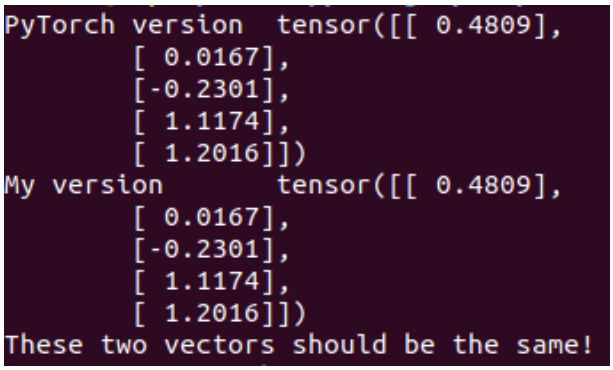


Figure 1: Result of executing the comparison code.

Exercise 2 - Neural synthesis

In this exercise we trained a VGG-19 network to generate a texture. The main parts of the code, where the topics 2.1 through 2.5 are defined, are given in the Listing 2.

Listing 2: Main lines of Exercise 2.

```
...
# # EXERCISE 2.1: Implement the Gram matrix computation
class GramMatrix(nn.Module):
    def forward(self, input):
        b, c, h, w = input.size()
        F = input.view(c, h * w)
        G = torch.mm(F, F.t())
        return G / (b*c)

...

# Exercise 2.5: part 1: replaces imsizes0, imsizes1 with resolutions ↩
   imsize0/2^(N_scales-1)... imsize0/2^(0)
imsizes0 = []
imsizes1 = []

for i in range(N_scales):
    imsizes0 += [int(imsize0 / 2 ** i)]
    imsizes1 += [int(imsize1 / 2 ** i)]

for s,cur_imsize0 in enumerate(imsizes0):

```

```
22      cur_imsize1 = imsizes1[s]
23
24      # Exercise 2.2: Replace this scale layer with nn.AdaptiveAvgPool2d and the correct resolution (using imsizes)
25      scale_layer = nn.AdaptiveAvgPool2d((cur_imsize0,cur_imsize1))
26      y = scale_layer(y0.detach())
27      y_feats = [gram_matrix(out.detach()) for out in vgg_net(y)]
28
29      # Exercise 2.3: Display size of every feature map from y_feats
30      for ind, feat in enumerate(y_feats):
31          print("Features at " + str(ind) + ": " + str(feat.shape))
32
33      if s==0:
34          x = my_utils.get_input_noise(cur_imsize0,cur_imsize1)
35
36      else:
37          alpha = .9
38          x = scale_layer(x.detach().cpu())
39          x = my_utils.smooth_image(x)
40
41      optimizer = torch.optim.LBFGS([x.requires_grad_()],max_iter=1)
42
43      for i in range(N_optim_iter):
44          def closure():
45              optimizer.zero_grad()
46              x_feats = vgg_net(x.cuda().clamp(-1,1))
47
48              loss = 0
49              for ll,x_feat,y_feat in zip(loss_lambda,x_feats,y_feats):
50                  # Exercise 2.4:
51                  loss += L2_loss(gram_matrix(x_feat),y_feat)
52
53              loss.backward()
54              print('iter=%.d, loss=%e'%(i,loss.item()))
55              return loss
56          optimizer.step(closure)
57          if i%save_every==0:
58              torchvision.utils.save_image(x.cpu().detach().clamp(-1,1),
                    im_folder + 'scale_%d_iteration_%d.jpg'%(s,i),normalize=
                    True)
59
60      torchvision.utils.save_image(x.cpu().detach().clamp(-1,1),im_folder +
            'final_synthesis.jpg',normalize=True)
```

Exercise 2.1 (lines 3-8) consists in defining the Gram matrix of a given input, which is calculated by computing the inner product between the original input and its transpose divided by its

dimensions.

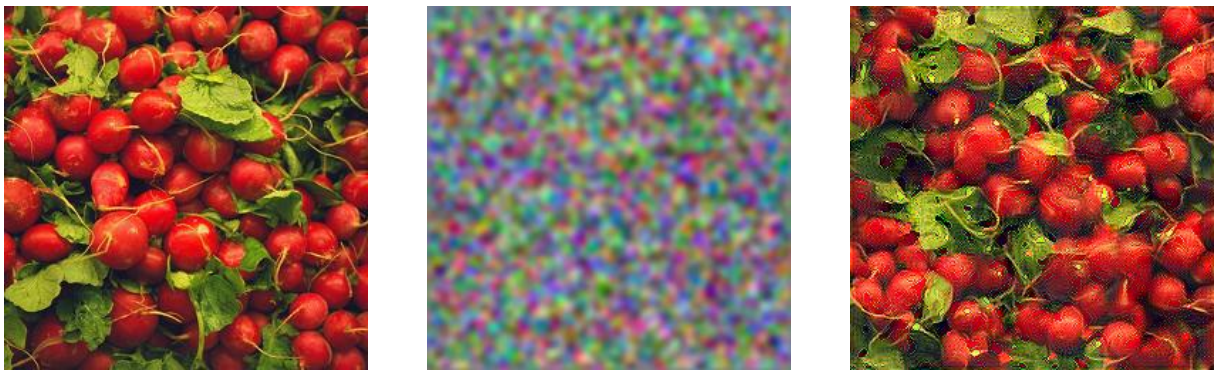
Exercise 2.2 (lines 25-27) uses an Adaptive Average Pooling to reduce the resolution of the input image before getting its Gram matrix. A traditional average pooling takes as input the stride and kernel size, but an adaptive method takes only the desired output size.

Exercise 2.3 (lines 85-86) displays the size of each feature map in the set or Gram matrices obtained in the previous step.

Exercise 2.4 (line 51) consists in the accumulation of the L2 loss analyzed for each output of the VGG network and the corresponding label. The L2 loss analyzes the euclidean distance between the output and the desired label.

Finally, exercise 2.5 (lines 16-22) creates a list of different image sizes, so we can train the VGG-19 network in different resolutions, and therefore obtain different patch sizes.

For this exercise, we analyzed the results of multiple scales, as given in Figure 2.



(a) The original image. (b) Noise input. (c) Final synthesis.

Figure 2: Results of the optimization approach.

We verify that the larger is the scale, the more semantically correct is the result image. However, the synthesis ends being a copy of the patches from the original image. The resolution is also low, when compared to the original image.

If we analyze the output of the executed code, we verify that the feature maps have always the same size (64×64, 128×128, 256×256 and 512×512) because they are not based on the patch size (scale), but on the input image size (512×512).

Exercise 3 - Feed-forward Texture Synthesis

This exercise is based on the article by Ulyanov et al., where there is not an optimization process, but rather a generation one. We train the neural network to generate a texture given any input; therefore, it has to have a generalistic architecture.

The main lines of the exercise are given in Listing 3.

Listing 3: Main lines of Exercise 3.

```
...
for n_iter in range(max_iter):
    optimizer.zero_grad()
    # element by element to allow the use of large training sizes
    for i in range(batch_size):
        sz = [img_size/1,img_size/2,img_size/4,img_size/8,img_size/16,img_size/32]
        zk = [torch.rand(1,n_input_ch,int(szk),int(szk)) for szk in sz]
        z_samples = [Variable(z.cuda()) for z in zk ]

        # EXERCISE 3.1: COMPLETE THIS LINE:
        batch_sample = gen(z_samples)

        sample = batch_sample[0,:,:,:].unsqueeze(0)

        # EXERCISE 3.2: COMPLETE THIS LINE:
        # Compute the VGG activations
        out = vgg_net(sample, out_keys)

        # EXERCISE 3.3: COMPLETE THIS LINE:
        gram_loss = GramMSELoss()
        norm_layer = I = Normalize_gradients.apply
        losses = [gram_loss(I(o), targets[i]) for o, i in zip(out, range(len(out)))]


        single_loss = (1/(batch_size))*sum(losses)
        single_loss.backward(retain_graph=False)
        loss_history[n_iter] = loss_history[n_iter] + single_loss.item()
        del out, losses, single_loss, batch_sample, z_samples, zk
        ...
```

We obtained the following images (given the input) presented in Figure 3. We verify some inconsistencies in the semantic field, however the image generated has a similar texture to the input one.

(a) The original image. (b) Training after 500 iterations. (c) Offline sample.

Figure 3: Results for the feed-forward network

Exercise 4 - Comparison between three methods

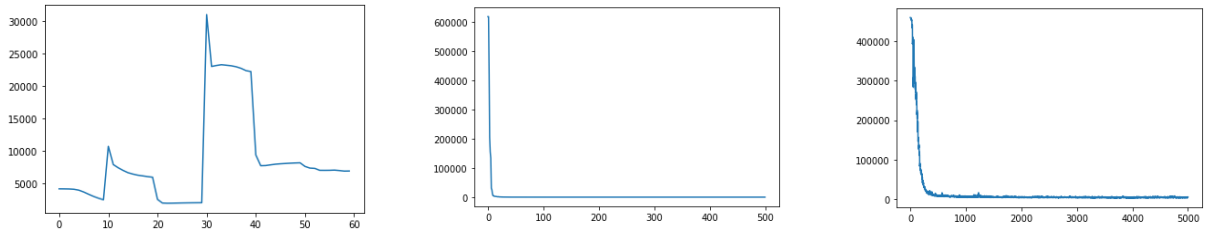
In this section we are going to compare the three synthesis methods in training/computation time, loss and image quality. Further, we are going to analyze the resulting patch mapping of each method, and its Gram matrix.

Analyzing Table 1, we verify that, even though the feed-forward method takes a long time to train, when trained, it has the quickest response time in terms of second to generate a similar texture of the one sent as input. The Gram matrix method, that consists in training a neural network to do the job of texture optimization, has the highest computation time.

Method	Training time	Computation time
Kwatra	-	0.83 s
Gram matrix	-	15.98 s
Feed forward	342.98 s	0.114 s

Table 1: Training and computation time for each method.

The loss of these three methods is shown in Figure 4. As we can verify, the methods that use a neural network have a decreasing loss, once this method has as objective to find the minimum of an optimization function. The Texture Optimization method, on the other hand has an unstable loss.

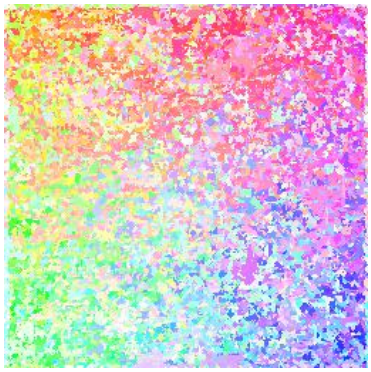


(a) Texture Optimization. (b) Gram matrix method. (c) Feed-forward method.

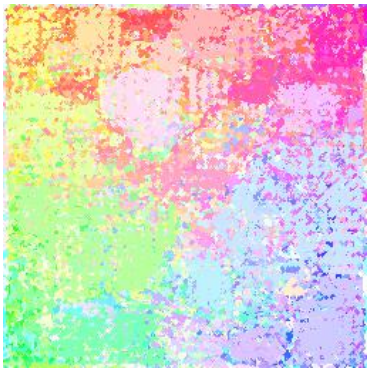
Figure 4: Comparing the losses.

In Figure 5, we see the comparison between the position maps of the three methods. It

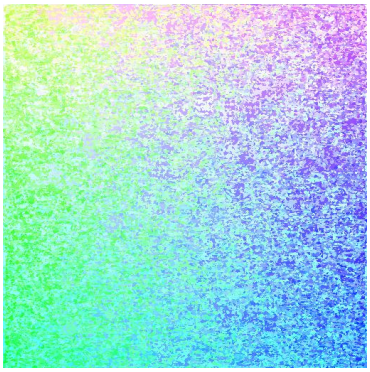
is clear that the texture optimization method ends copying a part of the original image into the result, which leads to some zones of a single color. The same happens in a smaller dimension with the Gram matrix method, once it is supposed to replicate the behavior of the first using a neural network. Finally, the feed-forward method does not preserve any complex structures from the original image, resulting in a regular position map.



(a) Mapping using Gram matrix.



(b) Mapping using Texture Optimization.



(c) Mapping using feed-forward.

Figure 5: Comparing the patches