Computer Vision 2: Exercise Sheet 5

Summary:

1. Encoder-Decoder network for dense prediction tasks
2. Loss functions for saliency prediction
3. Training the encoder-decoder network
4. Enabling GPU acceleration

1 Encoder-decoder network for saliency prediction

So far, we have exclusively considered networks applied to a classification task where the output is a single class label. The saliency prediction task takes as input an image, and outputs a saliency map of the same size. Each pixel in the saliency map indicates the probability that the pixel will be fixated by a human viewer. These tasks are often called *dense prediction tasks*, since their outputs is in the shape of an image. Another example of a dense prediction task is semantic segmentation.

For tasks where the input and output are both images, an encoder-decoder is a popular neural network architecture. An encoder-decoder consists of two parts: the encoder and the decoder, illustrated in Figure 1 for the saliency prediction task. The encoder "compresses" the input image into a latent vector representation. The decoder "decompresses" the latent vector into an output. The idea is that the encoder learns to extract a low-dimensional feature vector that allows the decoder to reconstruct the desired output. The latent vector is sometimes also called a bottleneck.

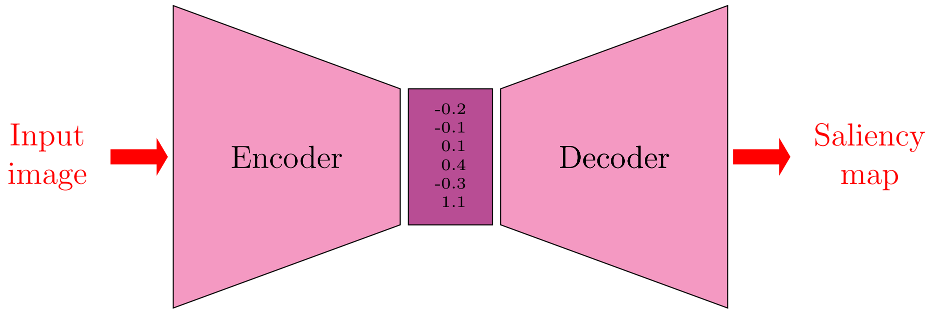

Figure 1: An encoder-decoder network.

In this exercise, we implement an encoder-decoder network that may be used for saliency prediction. In the next exercises, we create network modules for the encoder and decoder, and define a suitable loss function.

You can use the encoder-decoder network from this exercise as a building block for your course project.

1.1 Encoder

The encoder is based on the VGG16 "D" network. Implement a class called `Encoder` that is a subclass of `nn.Module` and defines a CNN shown in Table 1.

Table 1: The encoder is based on the VGG16 “D” network. The fully connected layers and the last pooling layer are omitted.

Layer	Output channels	Kernel	Stride	Padding	Activation
conv1_1	64	1 x 1	1	1	ReLU
conv1_2	64	3 x 3	1	1	ReLU
pool1		2 x 2	2	0	
conv2_1	128	3 x 3	1	1	ReLU
conv2_2	128	3 x 3	1	1	ReLU
pool2		2 x 2	2	0	
conv3_1	256	3 x 3	1	1	ReLU
conv3_2	256	3 x 3	1	1	ReLU
conv3_3	256	3 x 3	1	1	ReLU
pool3		2 x 2	2	0	
conv4_1	512	3 x 3	1	1	ReLU
conv4_2	512	3 x 3	1	1	ReLU
conv4_3	512	3 x 3	1	1	ReLU
pool4		2 x 2	2	0	
conv5_1	512	3 x 3	1	1	ReLU
conv5_2	512	3 x 3	1	1	ReLU
conv5_3	512	3 x 3	1	1	ReLU

Extend the encoder to allow using pre-trained weights of VGG16, available in the file `vgg16−conv.pth` on Moodle. Below is an code excerpt that shows one possible implementation. This should be replicated for all convolutional layers.

```python
class Encoder(nn.Module):
  def __init__(self, model_dict=None):
    super(Encoder, self).__init__()

    self.conv1_1 = nn.Conv2d(3, 64, 3, padding=1)
    if model_dict is not None:
        self.conv1_1.weight.data = model_dict['conv1_1_weight']
        self.conv1_1.bias.data = model_dict['conv1_1_bias']
```

The initializer can then be called without any arguments for default initialization of the parameters, and with a model dictionary for pre-trained weights:

```python
enc = Encoder()
# − or −
vgg_conv_weights = torch.load('vgg16−conv.pth')
enc = Encoder(vgg_conv_weights)
```

The inputs are of shape 3-by-224-by-224. What is the shape of the output of layer `conv5_3`?

1.2 Decoder

The encoder is based on the VGG16 “D” network. Implement a class called `Decoder` that is a subclass of `nn.Module` and defines a CNN shown in Table 2.

For the upsampling layers, you should use `torch.nn.Upsample`, with the following settings:

```
nn.Upsample(scale_factor=2, mode='bilinear', align_corners=False)
```

Use a sigmoid activation function for the output layer.

Table 2: The decoder network.

Layer	Output channels	Kernel	Stride	Padding	Activation
conv6_1	512	3 x 3	1	1	ReLU
conv6_2	512	3 x 3	1	1	ReLU
conv6_3	512	3 x 3	1	1	ReLU
upsample1		2 x 2	2	0	
conv7_1	512	3 x 3	1	1	ReLU
conv7_2	512	3 x 3	1	1	ReLU
conv7_3	512	3 x 3	1	1	ReLU
upsample2		2 x 2	2	0	
conv8_1	256	3 x 3	1	1	ReLU
conv8_2	256	3 x 3	1	1	ReLU
conv8_3	256	3 x 3	1	1	ReLU
upsample3		2 x 2	2	0	
conv9_1	128	3 x 3	1	1	ReLU
conv9_2	128	3 x 3	1	1	ReLU
upsample4		2 x 2	2	0	
conv10_1	512	3 x 3	1	1	ReLU
conv10_2	512	3 x 3	1	1	ReLU
output	1	1 x 1	1	0	Sigmoid

1.3 Encoder-Decoder

Create a class called `Generator` that is a subclass of `nn.Module`. Given an input image, this class first uses your `Encoder` and then applies your `Decoder` to produce an output. For 3-by-224-by-224 input images, the generator should produce 1-by-224-by-224 saliency maps.

2 Loss function for saliency prediction

Recall that the value of a pixel in the saliency indicates the probability that it is attended by a human viewer. Since many pixels in an image may be attended, it is appropriate to treat each predicted value as independent of others. A suitable loss function for this case is the average binary cross entropy (BCE) across all pixels. Given a prediction $\hat{y}$ and the round truth saliency y, the average BCE over all N pixels is

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^{N} y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i). \tag{1}$$

It has also been noted that it is possible to compute the loss on *downscaled* versions of the predicted and ground truth saliency maps[1].

[1]See, e.g., the paper "SalGAN: visual saliency prediction with adversarial networks", `https://arxiv.org/pdf/1701.01081.pdf`

We can create a callable loss function object that implements downscaling and the average BCE loss.

```
class BCELossWithDownsampling():
  def __init__(self):
    self.downsample = torch.nn.AvgPool2d(4, stride=4,
                                         count_include_pad=False)
    self.loss_fcn = torch.nn.BCEWithLogitsLoss()

  def __call__(self, pred, y):
    return self.loss_fcn(self.downsample(pred), self.downsample(y))
```

This loss function is then used as usual.

```
loss_fcn = BCELossWithDownsampling()
# ...
pred = model(batch['image'])
loss = loss_fcn(pred, batch['fixation'])
```

3 Training the encoder-decoder network

Create training and validation sets for the saliency data for the course project. See Exercise sheet 3 for more details. Use the transforms `Rescale` and `ToTensor` from Exercise 3. **Important**: add a third transform that normalizes the images (**not** the fixation maps). The required transform is:

```
transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                           std=[0.229, 0.224, 0.225])
```

This transform is best implemented as a class, and applied after `ToTensor`. Finally, create dataloaders `train_dl` and `valid_dl` for both datasets; enable shuffling on the training data.

Running the training loop on CPU only is very slow. If you only have access to a CPU, use a batch size of 1 and add printing for every training batch so that you can verify training is working.

A simple training loop with a validation at every epoch is shown below.

```
for epoch in range(epochs):
    gen.train()
    for batch in train_dl:
        pred = gen(batch['image'])
        loss = loss_fcn(pred, batch['fixation'])

        loss.backward()
        opt.step()
        opt.zero_grad()

    gen.eval()
    with torch.no_grad():
        valid_loss = sum(loss_fcn(gen(batch['image']),
                    batch['fixation']) for batch in valid_dl)
    print(epoch, valid_loss / len(valid_dl))
```

4 Enabling GPU acceleration

GPU acceleration provides a massive improvement in runtimes when working with large convolutional neural networks. You can check if PyTorch has access to a GPU device available to accelerate training with the following code snippet:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

The code should print `cuda:0` if a GPU is available. Note that even if you don't have a CUDA GPU available, all code that follows next will still work – it will however run on the CPU. It is a good idea to write code in a way that it works both with and without a GPU available.

To enable GPU acceleration, you need to move your network to the GPU. Assuming your model is stored in the instance `gen`, the following convert the parameters and buffers to CUDA tensors:

```
gen.to(device)
```

Finally, in your training loop, you also have to convert your data to CUDA tensors, for example:

```
image, fixation = batch['image'].to(device), batch['fixation'].to(device)
# next, use CUDA tensors...
pred = gen(image)
loss = loss_fcn(pred, fixation)
# etc.
```

With the network architecture of this exercise, a training batch size of 8 images is feasible to use with a GPU with 4 GB of memory. If your GPU has less memory, you can decrease the batch size to fit in the available memory. If you have more memory available, you can increase the batch size to improve performance.

Do work where it is most convenient. Before using a remote computer with a GPU, make sure all your code works as expected on your own computer where you can easily modify code and debug it. In the development phase, use small batch sizes, skip processing all data, and verify loading/saving works as expected. Only after you are sure of the correctness of your code, deploy it to a remote PC and run it.