Exercise 10: Semantic Segmentation

Semantic Segmentation

- Output of the model:
  - Assign label of classes to each pixel in the image.

- Loss metric:
  - We use pixel-wise cross-entropy loss.
  - Note that there are unlabelled pixels, and we should filter the unlabelled pixels and compute the loss only over remaining pixels.

Network Architecture – Feature

```python
class SegmentationNN(pl.LightningModule):

    def __init__(self, num_classes=23, hparams=None):
        super().__init__()
        self.hparams = hparams
        ########################################################################
        #                             YOUR CODE                                #
        ########################################################################

        from torchvision import models
        self.features = models.alexnet(pretrained=True).features
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Conv2d(256, 4096, kernel_size=1, padding=0),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Conv2d(4096, num_classes, kernel_size=1, padding=0),
            nn.Upsample(scale_factor=40),
            nn.Conv2d(num_classes, num_classes, kernel_size=3, padding=1),
        )

        ########################################################################
        #                          END OF YOUR CODE                            #
        ########################################################################
```

- Remark: For the feature extraction we use the pretrained alexnet, this model is quite lightweight and suitable for our task.

Network Architecture – Classifier

```python
class SegmentationNN(pl.LightningModule):

    def __init__(self, num_classes=23, hparams=None):
        super().__init__()
        self.hparams = hparams
        #######################################################################
        #                             YOUR CODE                               #
        #######################################################################

        from torchvision import models
        self.features = models.alexnet(pretrained=True).features
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Conv2d(256, 4096, kernel_size=1, padding=0),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Conv2d(4096, num_classes, kernel_size=1, padding=0),
            nn.Upsample(scale_factor=40),
            nn.Conv2d(num_classes, num_classes, kernel_size=3, padding=1),
        )

        #######################################################################
        #                          END OF YOUR CODE                           #
        #######################################################################
```

- Remark: For the classifier we offer a simple architecture that has a nn.Upsample to let the final Height and Width align with our original input size.

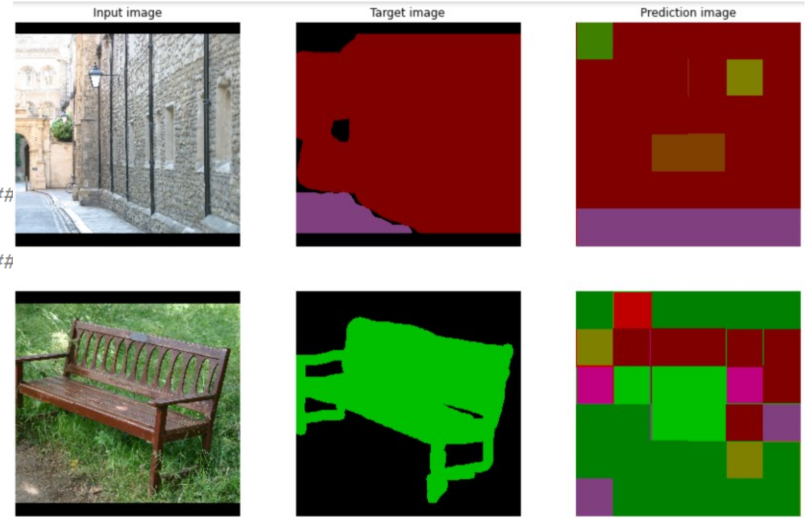
What is the problem of the model?

```python
class SegmentationNN(pl.LightningModule):

    def __init__(self, num_classes=23, hparams=None):
        super().__init__()
        self.hparams = hparams
        ###############################################################
        #                            YOUR CODE
        ###############################################################

        from torchvision import models
        self.features = models.alexnet(pretrained=True).features
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Conv2d(256, 4096, kernel_size=1, padding=0),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Conv2d(4096, num_classes, kernel_size=1, padding=0),
            nn.Upsample(scale_factor=40),
            nn.Conv2d(num_classes, num_classes, kernel_size=3, padding=1),
        )

        ########################################################################
        #                             END OF YOUR CODE                         #
        ########################################################################
```



- Still enough to pass the evaluation!

Network-forward

```python
def forward(self, x):
    """
    Forward pass of the convolutional neural network. Should not be called
    manually but by calling a model instance directly.

    Inputs:
    - x: PyTorch input Variable
    """
    #######################################################################
    #                             YOUR CODE                               #
    #######################################################################

    x = self.features(x)
    x = self.classifier(x)

    #######################################################################
    #                           END OF YOUR CODE                          #
    #######################################################################
    return x
```

- Remark: We need to make sure the tensor taken by the classifier has the same channels that our feature extractor produces, and the final output should have the size of

  (B, #classes, H, W)

We can also train without transfer learning

```python
class SegmentationNN(nn.Module):

    def __init__(self, num_classes=23, hp=None):
        super().__init__()
        self.hp = hp
        #######################################################################
        #                             YOUR CODE                               #
        #######################################################################

        self.mid_channels = 64
        self.pre_process = ConvLayer(3, self.mid_channels, 3, 1, 1)
        self.encoder_blocks = nn.ModuleList([
            ConvLayer(self.mid_channels, self.mid_channels * 2, 3, 1, 1), # 120x120
            ConvLayer(self.mid_channels * 2, self.mid_channels * 4, 3, 1, 1), # 60x60
            ConvLayer(self.mid_channels * 4, self.mid_channels * 4, 3, 1, 1), # 30x30
            ConvLayer(self.mid_channels * 4, self.mid_channels * 8, 3, 1, 1), # 15x15
        ])

        self.decoder_blocks = nn.ModuleList([
            ConvLayer(self.mid_channels * (8 + 8), self.mid_channels * 4, 3, 1, 1),
            ConvLayer(self.mid_channels * (4 + 4), self.mid_channels * 4, 3, 1, 1),
            ConvLayer(self.mid_channels * (4 + 4), self.mid_channels * 2, 3, 1, 1),
            ConvLayer(self.mid_channels * (2 + 2), self.mid_channels, 3, 1, 1),

        ])
        self.downsample = nn.MaxPool2d(2, 2)
        self.upsample = nn.Upsample(scale_factor=2, mode="bicubic")
        self.classifier = nn.Conv2d(self.mid_channels * 2 + 3, num_classes, kernel_size=3, padding=1)

        #######################################################################
        #                          END OF YOUR CODE                           #
        #######################################################################
```

```python
def forward(self, x):
    #######################################################################
    #                             YOUR CODE                               #
    #######################################################################

    # A unet shape network
    proc_x = self.pre_process(x)
    enc_1 = self.encoder_blocks[0](proc_x)
    tmp = self.downsample(enc_1)
    enc_2 = self.encoder_blocks[1](tmp)
    tmp = self.downsample(enc_2)
    enc_3 = self.encoder_blocks[2](tmp)
    tmp = self.downsample(enc_3)
    enc_4 = self.encoder_blocks[3](tmp)
    bottleneck = self.downsample(enc_4)

    tmp = self.upsample(bottleneck)
    dec_1 = self.decoder_blocks[0](torch.cat([tmp, enc_4], dim=1))
    tmp = self.upsample(dec_1)
    dec_2 = self.decoder_blocks[1](torch.cat([tmp, enc_3], dim=1))
    tmp = self.upsample(dec_2)
    dec_3 = self.decoder_blocks[2](torch.cat([tmp, enc_2], dim=1))
    tmp = self.upsample(dec_3)
    dec_4 = self.decoder_blocks[3](torch.cat([tmp, enc_1], dim=1))

    dec_4 = torch.cat([dec_4, proc_x], dim=1)
    dec_4 = torch.cat([dec_4, x], dim=1)
    x = self.classifier(dec_4)

    #######################################################################
    #                          END OF YOUR CODE                           #
    #######################################################################
```

We can also train without transfer learning

- Remark: We need to train more epochs without pre-trained weights

```
num_epochs = 50
log_nth = 5  # log_nth: log training accuracy and loss every nth iteration
batch_size = 16

optimizer = optim.Adam(
    model.parameters(),
    lr=1e-4,
    )
scheduler = torch.optim.lr_scheduler.OneCycleLR(
    optimizer, max_lr=5e-04, steps_per_epoch=len(train_loader),
    epochs=num_epochs, div_factor=2, pct_start=0.05
)
```

Training Loop (1)

```
########################################################################
# TODO - Train Your Model                                              #
########################################################################
import torch.optim as optim

num_epochs = 3
log_nth = 5  # log_nth: log training accuracy and loss every nth iteration
batch_size = 8


train_loss_history = []
train_acc_history = []
val_acc_history = []
val_loss_history = []

train_loader = torch.utils.data.DataLoader(train_data,batch_size=batch_size, shuffle=True,num_workers=0)
val_loader = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=False, num_workers=0)

optimizer = optim.Adam(
    model.parameters(),
    lr=1e-4,
    betas=(0.9, 0.999),
    eps=1e-8,
    weight_decay=0.0
    )

iter_per_epoch = len(train_loader)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model.to(device)
```

- Remark: We can define a Pytorch-style training loop directly inside the coding block.

Training Loop (2)

```python
print('START TRAIN.')

for epoch in range(num_epochs):
    # TRAINING
    train_acc_epoch = []
    for i, (inputs, targets) in enumerate(train_loader, 1):
        inputs, targets = inputs.to(device), targets.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = loss_func(outputs, targets)
        loss.backward()
        optimizer.step()

        train_loss_history.append(loss.cpu().detach().numpy())
        if log_nth and i % log_nth == 0:
            last_log_nth_losses = train_loss_history[-log_nth:]
            train_loss = np.mean(last_log_nth_losses)
            print('[Iteration %d/%d] TRAIN loss: %.3f' %
                  (i + epoch * iter_per_epoch,
                   iter_per_epoch * num_epochs,
                   train_loss))

        _, preds = torch.max(outputs, 1)

        # Only allow images/pixels with label >= 0 e.g. for segmentation
        targets_mask = targets >= 0
        train_acc = np.mean((preds == targets)[
                            targets_mask].cpu().detach().numpy())
        train_acc_history.append(train_acc)
        train_acc_epoch.append(train_acc)
```

```python
    if log_nth:
        train_acc =  np.mean(train_acc_epoch)
        print('[Epoch %d/%d] TRAIN acc/loss: %.3f/%.3f' % (epoch + 1,
                                                           num_epochs,
                                                           train_acc,
                                                           train_loss))
    # VALIDATION
    val_losses = []
    val_scores = []
    model.eval()
    for inputs, targets in val_loader:
        inputs, targets = inputs.to(device), targets.to(device)

        outputs = model.forward(inputs)
        loss = loss_func(outputs, targets)
        val_losses.append(loss.detach().cpu().numpy())

        _, preds = torch.max(outputs, 1)

        # Only allow images/pixels with target >= 0 e.g. for
        # segmentation
        targets_mask = targets >= 0
        scores = np.mean((preds == targets)[
                         targets_mask].cpu().detach().numpy())
        val_scores.append(scores)

    model.train()
    val_acc, val_loss = np.mean(val_scores), np.mean(val_losses)
    val_acc_history.append(val_acc)
    val_loss_history.append(val_loss)
    if log_nth:
        print('[Epoch %d/%d] VAL   acc/loss: %.3f/%.3f' % (epoch + 1,
                                                           num_epochs,
                                                           val_acc,
                                                           val_loss))
```

- Remark: This is the actual training loop, where we have the forward and the backward pass of the model, compute the loss, optimize the parameters, and log the loss/accuracy information.

Hyperparameters

```
num_epochs = 3
log_nth = 5  # lo
batch_size = 8
```

```
optimizer = optim.Adam(
    model.parameters(),
    lr=1e-4,
    betas=(0.9, 0.999),
    eps=1e-8,
    weight_decay=0.0
    )
```

- Remark: This is the hyperparameters that we set in our sample solution. You can also use a hyperparameter search to get a set of hyperparameters that suit your model well.

- For the sample model and this set of hyperparameters, we can reach an accuracy around 88%.

Semantic Segmentation

- Different Model Designs:
  - In this exercise, we show an easy way to achieve reasonable scores on our assignment by using pretrained model, you can try with different pretrained models that pytorch offers and compare them.
  - For this field of task, there are also some famous models that you may also learn from the lecture, e.g., FCN, U-Net, etc., for which you can have a closer look if you are interested.

Long et al. "Fully Convolutional Networks for Semantic Segmentation", CVPR, 2015
Ronneberger et al. "U-Net: Convolutional Networks for Biomedical Image Segmentation", MICCAI, 2015

Questions? Piazza

☺