assignment3

May 23, 2023

1 Assignment 3: Pytorch Segmentation + CAM

For this assignment, in the first part, we're going to use Deep Learning for a new task: semantic segmentation. In the second part, you will interpret networks with the class activation map (CAM) as discussed in classes.

1.1 Short recap of semantic segmentation

The goal of semantic segmentation is to classify each pixel of the image to a corresponding class of what the pixel represent. One major diference between semantic segmentation and classification is that for semantic segmentation, model output a label for each pixel instead of a single label for the whole image.

1.2 CMP Facade Database and Visualize Samples

In this assignment, we use a new dataset named: CMP Facade Database for semantic segmentation. This dataset is made up with 606 rectified images of the facade of various buildings. The facades are from different cities arount the world with different architectural styles.

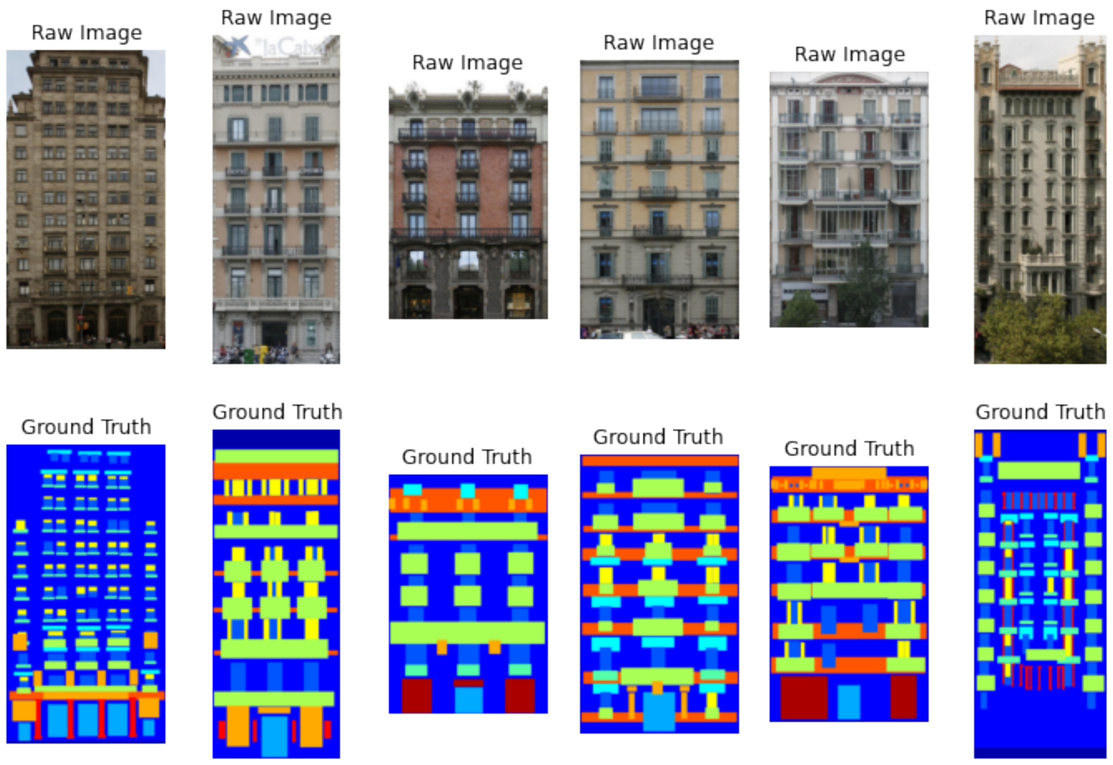
CMP Facade DB include 12 semantic classes:

- facade
- molding
- cornice
- pillar
- window
- door
- sill
- blind
- balcony
- shop
- deco
- background

In this assignment, we should use a model to classify each pixel in images to one of these 12 classes.

For more detail about CMP Facade Dataset, if you are intereseted, please check: https://cmp.felk.cvut.cz/~tylecr1/facade/

[1]:
```
import matplotlib.pyplot as plt
import numpy as np

idxs = [1, 2, 5, 6, 7, 8]
fig, axes = plt.subplots(nrows=2, ncols=6, figsize=(12, 8))
for i, idx in enumerate(idxs):
    pic = plt.imread("dataset/base/cmp_b000{}.jpg".format(idx))
    label = plt.imread("dataset/base/cmp_b000{}.png".format(idx), format="PNG")

    axes[0][i].axis('off')
    axes[0][i].imshow(pic)
    axes[0][i].set_title("Raw Image")

    axes[1][i].imshow(label)
    axes[1][i].axis('off')
    axes[1][i].set_title("Ground Truth")
```



1.3 Build Dataloader and Set Up Device

[2]:
```
#pip install torch
```

[3]:
```python
import torch
import torch.nn as nn
from torch.utils.data import Dataset
import torchvision
import torchvision.transforms as transforms
import torchvision.datasets as dset
import torchvision.transforms as T
import PIL
from PIL import Image
import numpy as np
import os
# import os.path as osp

from FCN.dataset import CMP_Facade_DB

os.environ["CUDA_VISIBLE_DEVICES"]="0"


def get_full_list(
    root_dir,
    base_dir="base",
    extended_dir="extended",
):
    data_list = []
    for name in [base_dir, extended_dir]:
        data_dir = os.path.join(
            root_dir, name
        )
        data_list += sorted(
            os.path.join(data_dir, img_name) for img_name in
            filter(
                lambda x: x[-4:] == '.jpg',
                os.listdir(data_dir)
            )
        )
    return data_list

TRAIN_SIZE = 500
VAL_SIZE = 30
TEST_SIZE = 70
full_data_list = get_full_list("dataset")

train_data_set = CMP_Facade_DB(full_data_list[: TRAIN_SIZE])
val_data_set = CMP_Facade_DB(full_data_list[TRAIN_SIZE: TRAIN_SIZE + VAL_SIZE])
test_data_set = CMP_Facade_DB(full_data_list[TRAIN_SIZE + VAL_SIZE:])

print("Training Set Size:", len(train_data_set))
print("Validation Set Size:", len(val_data_set))
print("Test Set Size:", len(test_data_set))

train_loader = torch.utils.data.DataLoader(
    train_data_set, batch_size=1, shuffle=True
)
val_loader = torch.utils.data.DataLoader(
    val_data_set, batch_size=1, shuffle=True
)
test_loader = torch.utils.data.DataLoader(
    test_data_set, batch_size=1, shuffle=False
)



USE_GPU = True

dtype = torch.float32 # we will be using float throughout this tutorial

if USE_GPU and torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

# Constant to control how frequently we print train loss
print_every = 100

print('using device:', device)
```

```
Training Set Size: 500
Validation Set Size: 30
Test Set Size: 76
using device: cuda
```

1.4 Fully Convolutional Networks for Semantic Segmentation

Here we are going to explore the classical work: “Fully Convolutional Networks for Semantic Segmentation”(FCN).

In FCN, the model uses the Transpose Convolution layers, which we’ve already learned during the lecture, to recover high resolution feature maps. For the overall introduction of Transpose Convolution and Fully Convolutional Networks, please review the lecture recording and lecture slides on Canvas(Lecture 10).

Here we do not cover all the details in FCN. Please check the original paper: https://arxiv.org/pdf/1411.4038.pdf for more details.

Besides of transpose Convolution, there are also some differences compared with the models we’ve been working on:

- Use 1x1 Convolution to replace fully connected layers to output score for each class.
- Use skip connection to combine high-level feature and local feature.

2 Part 1: FCN-32s (20%)

In this section, we first try to implement simple version of FCN without skip connection (i.e., FCN-32s) with VGG-16 as the backbone.

Compared with VGG-16, FCN-32s * replaces the fully connecteed layers with 1x1 convolution * adds a Transpose Convolution at the end to output dense prediction.

Task: 1. Complete FCN-32s in the notebook as instructed. 2. Train FCN-32s for 10 epochs and record the best model. Visualize the prediction results and report the test accuracy. 3. Train FCN-32s for 20 epochs with pretrained VGG-16 weights and record the best model. Visualize the prediction results and report the test accuracy.

2.1 1.1 Complete the FC-32s architecture:

The following Conv use kernel size = 3, padding = 1, stride =1 (except for conv1_1 where conv1_1 should use padding = 100)

- [conv1_1(3,64)-relu] -> [conv1_2(64,64)-relu] -> [maxpool1(2,2)]
- [conv2_1(64,128)-relu] -> [conv2_2(128,128)-relu] -> [maxpool2(2,2)]
- [conv3_1(128,256)-relu] -> [conv3_2(256,256)-relu] ->[conv3_3(256,256)-relu] -> [maxpool3(2,2)]
- [conv4_1(256,512)-relu] -> [conv4_2(512,512)-relu] ->[conv4_3(512,512)-relu] -> [maxpool4(2,2)]
- [conv5_1(512,512)-relu] -> [conv5_2(512,512)-relu] ->[conv5_3(512,512)-relu] -> [maxpool5(2,2)]

The following Conv use stride = 1, padding = 0 (KxK denotes kernel size, dropout probability=0.5) * [fc6=conv7x7(512, 4096)-relu-dropout2d] * [fc7=conv1x1(4096, 4096)-relu-dropout2d] * [score=conv1x1(4096, num_classes)]

The transpose convolution: kernal size = 64, stride = 32, bias = False * [transpose_conv(n_class, n_class)]

Hint: The output of the transpose convolution might not have the same shape as the input, take [19: 19 + input_image_width], [19: 19 + input_image_height] for width and height dimension of the output to get the output with the same shape as the input

[4]:
```
class FCN32s(nn.Module):
    def __init__(self, n_class=21):
        super(FCN32s, self).__init__()
        ␣
 ↪###############################################################################
        # TODO: Implement the layers for FCN32s.                                  ␣
 ↪        #
        ␣
 ↪###############################################################################
```

```
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        self.conv1_1 = nn.Conv2d(3, 64, kernel_size = 3, padding = 150, stride␣
↪= 1)
        self.relu1_1 = nn.ReLU()
        self.conv1_2 = nn.Conv2d(64, 64, kernel_size = 3, padding = 1, stride =␣
↪1)
        self.relu1_2 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size = 2, stride = 2)
        self.conv2_1  = nn.Conv2d(64, 128, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu2_1 = nn.ReLU()
        self.conv2_2 = nn.Conv2d(128, 128, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu2_2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size = 2, stride = 2)
        self.conv3_1 = nn.Conv2d(128, 256, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu3_1 = nn.ReLU()
        self.conv3_2 = nn.Conv2d(256, 256, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu3_2 = nn.ReLU()
        self.conv3_3 = nn.Conv2d(256, 256, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu3_3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size = 2, stride = 2)
        self.conv4_1 = nn.Conv2d(256, 512, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu4_1 = nn.ReLU()
        self.conv4_2 = nn.Conv2d(512, 512, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu4_2 = nn.ReLU()
        self.conv4_3 = nn.Conv2d(512, 512, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu4_3 = nn.ReLU()
        self.pool4 = nn.MaxPool2d(kernel_size = 2, stride = 2)
        self.conv5_1 = nn.Conv2d(512, 512, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu5_1 = nn.ReLU()
        self.conv5_2 = nn.Conv2d(512, 512, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu5_2 = nn.ReLU()
        self.conv5_3 = nn.Conv2d(512, 512, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu5_3 = nn.ReLU()
        self.pool5 = nn.MaxPool2d(kernel_size = 2, stride = 2)
        self.fc6 = nn.Conv2d(512, 4096, kernel_size = 7, padding = 0, stride =␣
↪1)
        self.dropout1 = nn.Dropout2d(p = 0.5)
        self.fc7 = nn.Conv2d(4096, 4096, kernel_size = 1, padding = 0, stride =␣
↪1)
        self.dropout2 = nn.Dropout2d(p = 0.5)
        self.score = nn.Conv2d(4096, n_class, kernel_size = 1, stride = 1,␣
↪padding = 0)
        self.transpose_conv = nn.ConvTranspose2d(n_class, n_class, kernel_size␣
↪= 64, stride = 32, bias = False)
        self.relu = nn.ReLU()
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
      ␣
↪################################################################################
        #                                 END OF YOUR CODE                      ␣
↪         #
      ␣
↪################################################################################

        self._initialize_weights()

    def get_upsampling_weight(self, in_channels, out_channels, kernel_size):
        """Make a 2D bilinear kernel suitable for upsampling"""
        factor = (kernel_size + 1) // 2
        if kernel_size % 2 == 1:
            center = factor - 1
        else:
            center = factor - 0.5
        og = np.ogrid[:kernel_size, :kernel_size]
        filt = (1 - abs(og[0] - center) / factor) * \
               (1 - abs(og[1] - center) / factor)
        weight = np.zeros((in_channels, out_channels, kernel_size, kernel_size),
                          dtype=np.float64)
        weight[range(in_channels), range(out_channels), :, :] = filt
        return torch.from_numpy(weight).float()

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                m.weight.data.zero_()
                if m.bias is not None:
                    m.bias.data.zero_()
            if isinstance(m, nn.ConvTranspose2d):
                assert m.kernel_size[0] == m.kernel_size[1]
                initial_weight = self.get_upsampling_weight(
                    m.in_channels, m.out_channels, m.kernel_size[0])
                m.weight.data.copy_(initial_weight)

    def forward(self, x):
        ␣
↪################################################################################
        # TODO: Implement the forward pass for FCN32s.                            ␣
↪         #
        ␣
↪################################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        h = self.relu1_1(self.conv1_1(x))
        h = self.pool1(self.relu1_2(self.conv1_2(h)))
        h = self.relu2_1(self.conv2_1(h))
        h = self.pool2(self.relu2_2(self.conv2_2(h)))
        h = self.relu3_1(self.conv3_1(h))
        h = self.relu3_2(self.conv3_2(h))
        h = self.pool3(self.relu3_3(self.conv3_3(h)))
        h = self.relu4_1(self.conv4_1(h))
        h = self.relu4_2(self.conv4_2(h))
        h = self.pool4(self.relu4_3(self.conv4_3(h)))
        h = self.relu5_1(self.conv5_1(h))
        h = self.relu5_2(self.conv5_2(h))
        h = self.pool5(self.relu5_3(self.conv5_3(h)))
        h = self.dropout1(self.relu(self.fc6(h)))
        h = self.dropout2(self.relu(self.fc7(h)))
        h = self.score(h)
        h = self.transpose_conv(h)

        h = h[:, :, 19 : 19 + x.size(2), 19 : 19 + x.size(3)]
#          print(h.shape)
#          print('-')
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ␣
↪################################################################################
        #                                END OF YOUR CODE                          ␣
↪         #
        ␣
↪################################################################################

        return h


    def copy_params_from_vgg16(self, vgg16):
        features = [
            self.conv1_1, self.relu1_1,
            self.conv1_2, self.relu1_2,
            self.pool1,
            self.conv2_1, self.relu2_1,
            self.conv2_2, self.relu2_2,
            self.pool2,
            self.conv3_1, self.relu3_1,
            self.conv3_2, self.relu3_2,
            self.conv3_3, self.relu3_3,
            self.pool3,
            self.conv4_1, self.relu4_1,
            self.conv4_2, self.relu4_2,
            self.conv4_3, self.relu4_3,
            self.pool4,
            self.conv5_1, self.relu5_1,
            self.conv5_2, self.relu5_2,
            self.conv5_3, self.relu5_3,
            self.pool5,
        ]
        for l1, l2 in zip(vgg16.features, features):
            if isinstance(l1, nn.Conv2d) and isinstance(l2, nn.Conv2d):
                assert l1.weight.size() == l2.weight.size()
                assert l1.bias.size() == l2.bias.size()
                l2.weight.data = l1.weight.data
                l2.bias.data = l1.bias.data
        for i, name in zip([0, 3], ['fc6', 'fc7']):
            l1 = vgg16.classifier[i]
            l2 = getattr(self, name)
            l2.weight.data = l1.weight.data.view(l2.weight.size())
            l2.bias.data = l1.bias.data.view(l2.bias.size())
    def _hist(pred, gt, n_class):
        pred = resize_label(pred, gt.shape)  # Resize predicted labels
        mask = (gt >= 0) & (gt < n_class)
        hist = np.bincount(
                n_class * gt[mask].astype(int) +
                pred[mask], minlength=n_class ** 2
                ).reshape(n_class, n_class)
        return hist
```

2.2 1.2 Train FCN-32s from scratch

[5]:
```
from FCN.trainer import Trainer

model32 = FCN32s(n_class=12)
model32.to(device)

best_model = Trainer(
    model32,
    train_loader,
    val_loader,
    test_loader,
    num_epochs=10
)
```

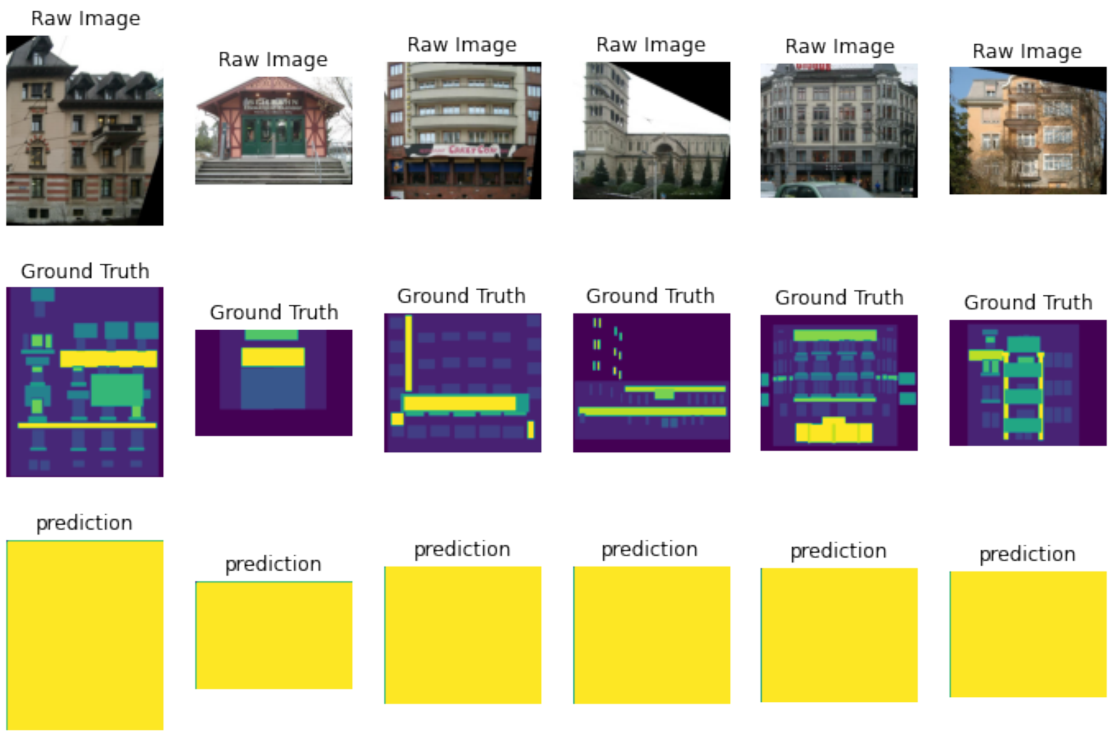
```
/opt/conda/lib/python3.9/site-packages/torch/nn/functional.py:718: UserWarning:
Named tensors and all their associated APIs are an experimental feature and
subject to change. Please do not use them for anything important until they are
released as stable. (Triggered internally at
/pytorch/c10/core/TensorImpl.h:1156.)
  return torch.max_pool2d(input, kernel_size, stride, padding, dilation,
ceil_mode)
```

```
Init Model
Avg Acc: 0.2307, Mean IoU: 0.01922
Epochs: 0
Epoch Loss: 2.43, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 1
Epoch Loss: 2.202, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 2
Epoch Loss: 2.008, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 3
Epoch Loss: 1.943, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 4
Epoch Loss: 1.93, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 5
Epoch Loss: 1.925, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 6
Epoch Loss: 1.922, Avg Acc: 0.3432, Mean IoU: 0.02862
Epochs: 7
Epoch Loss: 1.92, Avg Acc: 0.3434, Mean IoU: 0.02871
Epochs: 8
Epoch Loss: 1.918, Avg Acc: 0.3453, Mean IoU: 0.0295
Epochs: 9
Epoch Loss: 1.916, Avg Acc: 0.3475, Mean IoU: 0.03035
Test Acc: 0.3475, Test Mean IoU: 0.03035
```

[6]:
```
from FCN.trainer import visualize
visualize(best_model, test_loader)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



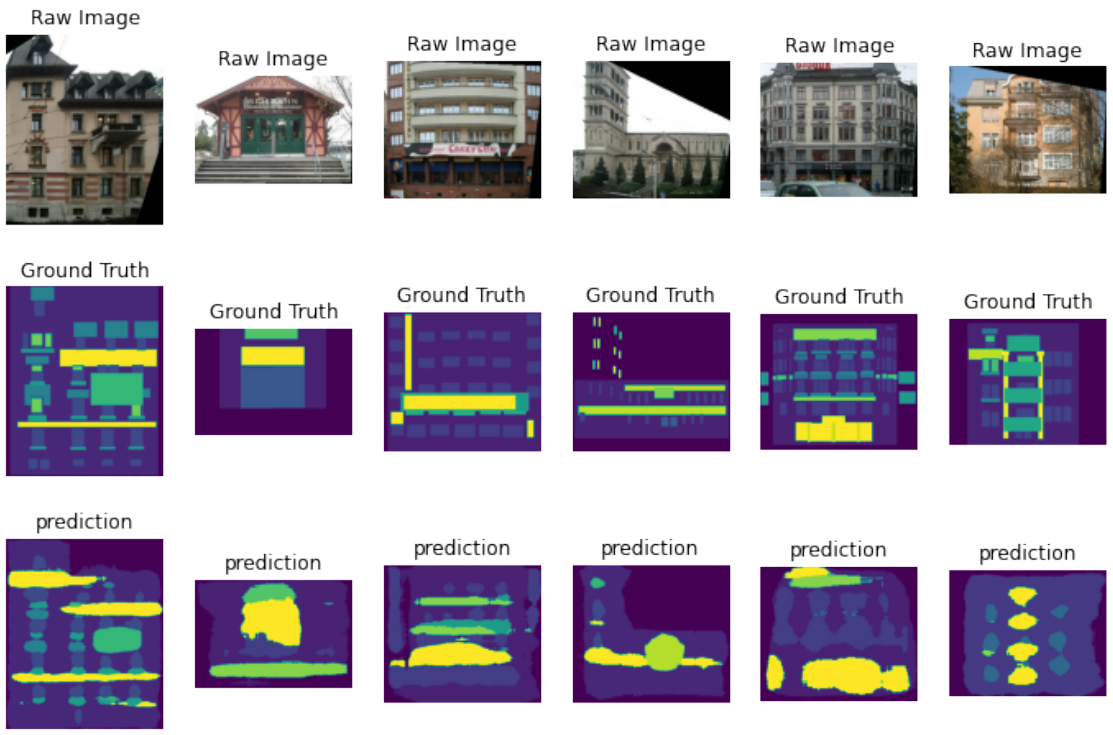
2.3 1.3 Train FCN-32s with the pretrained VGG16 weights

[7]:
```python
import torchvision
from FCN.trainer import Trainer

vgg16 = torchvision.models.vgg16(pretrained=True)

model32_pretrain = FCN32s(n_class=12)
model32_pretrain.copy_params_from_vgg16(vgg16)
model32_pretrain.to(device)

best_model_pretrain = Trainer(
    model32_pretrain,
    train_loader,
    val_loader,
    test_loader,
    num_epochs=20
)
```

```
Init Model
Avg Acc: 0.2307, Mean IoU: 0.01922
Epochs: 0
Epoch Loss: 1.638, Avg Acc: 0.455, Mean IoU: 0.1476
Epochs: 1
Epoch Loss: 1.364, Avg Acc: 0.5385, Mean IoU: 0.1694
Epochs: 2
Epoch Loss: 1.225, Avg Acc: 0.5583, Mean IoU: 0.2185
Epochs: 3
Epoch Loss: 1.119, Avg Acc: 0.5902, Mean IoU: 0.2542
Epochs: 4
Epoch Loss: 1.016, Avg Acc: 0.5906, Mean IoU: 0.2708
Epochs: 5
Epoch Loss: 0.9562, Avg Acc: 0.6239, Mean IoU: 0.287
Epochs: 6
Epoch Loss: 0.8746, Avg Acc: 0.6188, Mean IoU: 0.3085
Epochs: 7
Epoch Loss: 0.7999, Avg Acc: 0.5991, Mean IoU: 0.299
Epochs: 8
Epoch Loss: 0.7664, Avg Acc: 0.651, Mean IoU: 0.3486
Epochs: 9
Epoch Loss: 0.7125, Avg Acc: 0.611, Mean IoU: 0.334
Epochs: 10
Epoch Loss: 0.6722, Avg Acc: 0.6474, Mean IoU: 0.3417
Epochs: 11
Epoch Loss: 0.6372, Avg Acc: 0.6625, Mean IoU: 0.3573
Epochs: 12
Epoch Loss: 0.6106, Avg Acc: 0.6641, Mean IoU: 0.3465
Epochs: 13
Epoch Loss: 0.5894, Avg Acc: 0.6675, Mean IoU: 0.3599
Epochs: 14
Epoch Loss: 0.5711, Avg Acc: 0.6652, Mean IoU: 0.3588
Epochs: 15
Epoch Loss: 0.5511, Avg Acc: 0.6603, Mean IoU: 0.3632
Epochs: 16
Epoch Loss: 0.5327, Avg Acc: 0.6606, Mean IoU: 0.3599
Epochs: 17
Epoch Loss: 0.5223, Avg Acc: 0.6738, Mean IoU: 0.3433
Epochs: 18
Epoch Loss: 0.502, Avg Acc: 0.6787, Mean IoU: 0.3808
Epochs: 19
Epoch Loss: 0.4914, Avg Acc: 0.6754, Mean IoU: 0.3701
Test Acc: 0.6787, Test Mean IoU: 0.3808
```

[8]:
```
from FCN.trainer import visualize
visualize(best_model_pretrain, test_loader)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



3 Part 2: FCN-8s(30%)

In this section, we explore with another technique introduced in FCN paper: Skip Connection.

Task: Read the paper and understand the skip connection, then 1. Complete FCN-8s in the notebook as instructed. 2. Train the network for 20 epochs with pretrained VGG-16 weights and record the best model. Visualize the prediction results and report the test accuracy.

3.1 Here we provide the structure of FCN-8s, the variant of FCN with skip connections.

FCN-8s architecture:

- [conv4_1(256,512)-relu] -> [conv4_2(512,512)-relu] ->[conv4_3(512,512)-relu] -> [max-pool4(2,2)]
- [conv5_1(512,512)-relu] -> [conv5_2(512,512)-relu] ->[conv5_3(512,512)-relu] -> [max-pool5(2,2)]

The following Conv use stride = 1, padding = 0 (KxK denotes kernel size, dropout probability=0.5) * [fc6=conv7x7(512, 4096)-relu-dropout2d] * [fc7=conv1x1(4096, 4096)-relu-dropout2d] * [score=conv1x1(4096, num_classes)]

The Additional Score Pool use kernel size = 1, stride = 1, padding = 0 * [score_pool_3 =conv1x1(256, num_classes)] * [score_pool_4 =conv1x1(512, num_classes)]

The transpose convolution: kernal size = 4, stride = 2, bias = False * [upscore1 = transpose_conv(n_class, n_class)]

The transpose convolution: kernal size = 4, stride = 2, bias = False * [upscore2 = transpose_conv(n_class, n_class)]

The transpose convolution: kernal size = 16, stride = 8, bias = False * [upscore3 = transpose_conv(n_class, n_class)]

Different from FCN-32s which has only single path from input to output, there are multiple data path from input to output in FCN-8s.

The following graph is from original FCN paper, you can also find the graph there.

“Layers are shown as grids that reveal relative spatial coarseness. Only pooling and prediction layers are shown; intermediate convolution layers (including converted fully connected layers) are omitted.” —- FCN

Detailed path specification:

- score_pool_3
  - input: output from layer “pool3”
  - take [9: 9 + upscore2_width], [9: 9 + upscore2_height]
- score_pool_4,
  - input: output from layer “pool4”
  - take [5: 5 + upscore1_width], [5: 5 + upscore1_height]
- upscore1
  - input: output from layer “score”
- upscore2:
  - input: output from layer “score_pool_4” + output from layer “upscore1”
- upscore3:
  - input: output from layer “score_pool_3” + output from layer “upscore2”
  - take [31: 31 + input_image_width], [31: 31 + input_image_height]

[5]:
```python
import torch.nn as nn

class FCN8s(nn.Module):

    def __init__(self, n_class=12):
        super(FCN8s, self).__init__()
    ␣
↪###########################################################################
        # TODO: Implement the layers for FCN8s.                             ␣
↪         #
    ␣
↪###########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        self.conv1_1 = nn.Conv2d(3, 64, kernel_size = 3, padding = 100, stride␣
↪= 1)
        self.relu1_1 = nn.ReLU()
        self.conv1_2 = nn.Conv2d(64, 64, kernel_size = 3, padding = 1, stride =␣
↪1)
        self.relu1_2 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size = 2, stride = 2)
        self.conv2_1  = nn.Conv2d(64, 128, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu2_1 = nn.ReLU()
        self.conv2_2 = nn.Conv2d(128, 128, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu2_2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size = 2, stride = 2)
        self.conv3_1 = nn.Conv2d(128, 256, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu3_1 = nn.ReLU()
        self.conv3_2 = nn.Conv2d(256, 256, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu3_2 = nn.ReLU()
        self.conv3_3 = nn.Conv2d(256, 256, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu3_3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size = 2, stride = 2)
        self.conv4_1 = nn.Conv2d(256, 512, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu4_1 = nn.ReLU()
        self.conv4_2 = nn.Conv2d(512, 512, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu4_2 = nn.ReLU()
        self.conv4_3 = nn.Conv2d(512, 512, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu4_3 = nn.ReLU()
        self.pool4 = nn.MaxPool2d(kernel_size = 2, stride = 2)
        self.conv5_1 = nn.Conv2d(512, 512, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu5_1 = nn.ReLU()
        self.conv5_2 = nn.Conv2d(512, 512, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu5_2 = nn.ReLU()
        self.conv5_3 = nn.Conv2d(512, 512, kernel_size = 3, padding = 1, stride␣
↪= 1)
        self.relu5_3 = nn.ReLU()
        self.pool5 = nn.MaxPool2d(kernel_size = 2, stride = 2)
        self.fc6 = nn.Conv2d(512, 4096, kernel_size = 7, padding = 0, stride =␣
↪1)
        self.dropout1 = nn.Dropout2d(p = 0.5)
        self.fc7 = nn.Conv2d(4096, 4096, kernel_size = 1, padding = 0, stride =␣
↪1)
        self.dropout2 = nn.Dropout2d(p = 0.5)
        self.score = nn.Conv2d(4096, n_class, kernel_size = 1, stride = 1,␣
↪padding = 0)
        #self.transpose_conv = nn.ConvTranspose2d(n_class, n_class, kernel_size␣
↪= 64, stride = 32, bias = False)
        self.relu = nn.ReLU()

        self.score_pool_3 = nn.Conv2d(256, n_class, kernel_size = 1, stride =␣
↪1, padding = 0)
        self.score_pool_4 = nn.Conv2d(512, n_class, kernel_size = 1, stride =␣
↪1, padding = 0)

        self.upscore1 = nn.ConvTranspose2d(n_class, n_class, kernel_size = 4,␣
↪stride = 2, bias = False)
        self.upscore2 = nn.ConvTranspose2d(n_class, n_class, kernel_size = 4,␣
↪stride = 2, bias = False)
        self.upscore3 = nn.ConvTranspose2d(n_class, n_class, kernel_size = 16,␣
↪stride = 8, bias = False)
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
      ␣
↪################################################################################
        #                                  END OF YOUR CODE                    ␣
↪       #
      ␣
↪################################################################################

        self._initialize_weights()

    def get_upsampling_weight(self, in_channels, out_channels, kernel_size):
        """Make a 2D bilinear kernel suitable for upsampling"""
        factor = (kernel_size + 1) // 2
        if kernel_size % 2 == 1:
            center = factor - 1
        else:
            center = factor - 0.5
        og = np.ogrid[:kernel_size, :kernel_size]
        filt = (1 - abs(og[0] - center) / factor) * \
               (1 - abs(og[1] - center) / factor)
        weight = np.zeros((in_channels, out_channels, kernel_size, kernel_size),
                          dtype=np.float64)
        weight[range(in_channels), range(out_channels), :, :] = filt
        return torch.from_numpy(weight).float()

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                m.weight.data.zero_()
                if m.bias is not None:
                    m.bias.data.zero_()
            if isinstance(m, nn.ConvTranspose2d):
                assert m.kernel_size[0] == m.kernel_size[1]
                initial_weight = self.get_upsampling_weight(
                    m.in_channels, m.out_channels, m.kernel_size[0])
                m.weight.data.copy_(initial_weight)


    def forward(self, x):
       ␣
↪################################################################################
        # TODO: Implement the forward pass for FCN8s.                               ␣
↪       #
       ␣
↪################################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        h = self.relu1_1(self.conv1_1(x))
        h = self.pool1(self.relu1_2(self.conv1_2(h)))
        h = self.relu2_1(self.conv2_1(h))
        h = self.pool2(self.relu2_2(self.conv2_2(h)))
        h = self.relu3_1(self.conv3_1(h))
        h = self.relu3_2(self.conv3_2(h))
        pool3 = self.pool3(self.relu3_3(self.conv3_3(h)))
        h = self.relu4_1(self.conv4_1(pool3))
        h = self.relu4_2(self.conv4_2(h))
        pool4 = self.pool4(self.relu4_3(self.conv4_3(h)))
        h = self.relu5_1(self.conv5_1(pool4))
        h = self.relu5_2(self.conv5_2(h))
        h = self.pool5(self.relu5_3(self.conv5_3(h)))
        h = self.dropout1(self.relu(self.fc6(h)))
        h = self.dropout2(self.relu(self.fc7(h)))
        h = self.score(h)
        #h = self.transpose_conv(h)
        spool_3 = self.score_pool_3(pool3)
        spool_4 = self.score_pool_4(pool4)

        h = self.upscore1(h)
        spool_4 = spool_4[:,:, 5 : 5 + h.size(2), 5 : 5 + h.size(3)]
        h = self.upscore2(h + spool_4)
        spool_3 = spool_3[:,:, 9 : 9 + h.size(2), 9 : 9 + h.size(3)]
        h = self.upscore3(h + spool_3)

        h = h[:, :, 31 : 31 + x.size(2), 31 : 31 + x.size(3)].contiguous()

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
       ␣
↪################################################################################
        #                                  END OF YOUR CODE                                  ␣
↪         #
       ␣
↪################################################################################

        return h

    def copy_params_from_vgg16(self, vgg16):
        features = [
            self.conv1_1, self.relu1_1,
            self.conv1_2, self.relu1_2,
            self.pool1,
            self.conv2_1, self.relu2_1,
            self.conv2_2, self.relu2_2,
            self.pool2,
            self.conv3_1, self.relu3_1,
            self.conv3_2, self.relu3_2,
            self.conv3_3, self.relu3_3,
            self.pool3,
            self.conv4_1, self.relu4_1,
            self.conv4_2, self.relu4_2,
            self.conv4_3, self.relu4_3,
            self.pool4,
            self.conv5_1, self.relu5_1,
            self.conv5_2, self.relu5_2,
            self.conv5_3, self.relu5_3,
            self.pool5,
        ]
        for l1, l2 in zip(vgg16.features, features):
            if isinstance(l1, nn.Conv2d) and isinstance(l2, nn.Conv2d):
                assert l1.weight.size() == l2.weight.size()
                assert l1.bias.size() == l2.bias.size()
                l2.weight.data.copy_(l1.weight.data)
                l2.bias.data.copy_(l1.bias.data)
        for i, name in zip([0, 3], ['fc6', 'fc7']):
            l1 = vgg16.classifier[i]
            l2 = getattr(self, name)
            l2.weight.data.copy_(l1.weight.data.view(l2.weight.size()))
            l2.bias.data.copy_(l1.bias.data.view(l2.bias.size()))
```

[6]:
```
from FCN.trainer import Trainer
import torchvision

vgg16 = torchvision.models.vgg16(pretrained=True)

model8 = FCN8s(n_class=12)
model8.copy_params_from_vgg16(vgg16)
model8.to(device)

best_model_fcn8s = Trainer(
    model8,
    train_loader,
    val_loader,
    test_loader,
    num_epochs=20
)
```

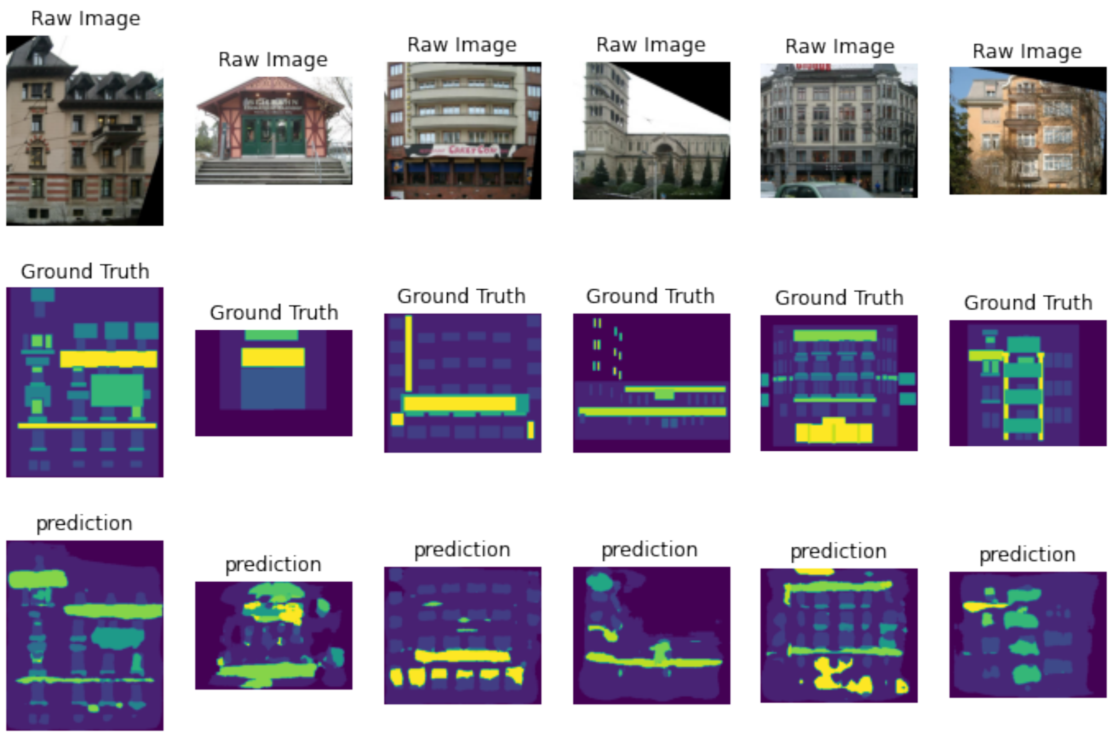
```
/opt/conda/lib/python3.9/site-packages/torch/nn/functional.py:718: UserWarning:
Named tensors and all their associated APIs are an experimental feature and
subject to change. Please do not use them for anything important until they are
released as stable. (Triggered internally at
/pytorch/c10/core/TensorImpl.h:1156.)
  return torch.max_pool2d(input, kernel_size, stride, padding, dilation,
ceil_mode)
```

```
Init Model
Avg Acc: 0.2307, Mean IoU: 0.01922
Epochs: 0
Epoch Loss: 1.211, Avg Acc: 0.6168, Mean IoU: 0.307
Epochs: 1
Epoch Loss: 0.9711, Avg Acc: 0.6441, Mean IoU: 0.3594
Epochs: 2
Epoch Loss: 0.8665, Avg Acc: 0.6613, Mean IoU: 0.3894
Epochs: 3
Epoch Loss: 0.7902, Avg Acc: 0.6944, Mean IoU: 0.43
Epochs: 4
Epoch Loss: 0.7296, Avg Acc: 0.6951, Mean IoU: 0.417
Epochs: 5
Epoch Loss: 0.6677, Avg Acc: 0.7025, Mean IoU: 0.4239
Epochs: 6
```

```
Epoch Loss: 0.6022, Avg Acc: 0.7158, Mean IoU: 0.3961
Epochs: 7
Epoch Loss: 0.5468, Avg Acc: 0.7026, Mean IoU: 0.4241
Epochs: 8
Epoch Loss: 0.5169, Avg Acc: 0.7124, Mean IoU: 0.4379
Epochs: 9
Epoch Loss: 0.4683, Avg Acc: 0.6917, Mean IoU: 0.4515
Epochs: 10
Epoch Loss: 0.4416, Avg Acc: 0.7264, Mean IoU: 0.4635
Epochs: 11
Epoch Loss: 0.4091, Avg Acc: 0.7237, Mean IoU: 0.4549
Epochs: 12
Epoch Loss: 0.3899, Avg Acc: 0.7286, Mean IoU: 0.4597
Epochs: 13
Epoch Loss: 0.368, Avg Acc: 0.7264, Mean IoU: 0.455
Epochs: 14
Epoch Loss: 0.4065, Avg Acc: 0.7297, Mean IoU: 0.4516
Epochs: 15
Epoch Loss: 0.3461, Avg Acc: 0.7341, Mean IoU: 0.4764
Epochs: 16
Epoch Loss: 0.3179, Avg Acc: 0.734, Mean IoU: 0.4397
Epochs: 17
Epoch Loss: 0.2983, Avg Acc: 0.7397, Mean IoU: 0.4821
Epochs: 18
Epoch Loss: 0.2892, Avg Acc: 0.7376, Mean IoU: 0.4746
Epochs: 19
Epoch Loss: 0.2842, Avg Acc: 0.7369, Mean IoU: 0.4783
Test Acc: 0.7397, Test Mean IoU: 0.4821
```

[7]:
```python
from FCN.trainer import visualize
visualize(best_model_fcn8s, test_loader)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



4 Part 3: Questions(20%):

Question 1: Compare the FCN-32s training from scratch with the FCN-32s with pretrained weights? What do you observe? Does pretrained weights help? Why? Please be as specific as possible. Using pretrained VGG weights greatly improved the performance of the FCN-32s model compared to training it from scratch. The IOU score increased from 0.03035 to 0.3808, indicating that pretrained weights effectively enhance segmentation performance. These weights provide a stronger starting point, enabling the model to learn better representations and achieve higher accuracy in segmentation. They transfer learned features from a large-scale image classification task, capturing both low-level and high-level visual patterns. This initialization helps the model converge faster and enhances its ability to generalize by leveraging knowledge from the VGG network.

Visualizations clearly demonstrate that the FCN-32 model without pretrained weights struggles with segmentation, as most of the image is colored yellow. However, when pretrained VGG weights are used with FCN-32, the model's performance significantly improves, as shown by the comparison with ground truth images. Therefore, it can be concluded that FCN-32 with pretrained weights outperforms FCN-32 without pretrained weights by a substantial margin.

Question 2: Compare the performance and visualization of FCN-32s and FCN-8s (both with pretrained weights). What do you observe? Which performs better? Why? Please be as specific as possible. The FCN-8s model, initialized with pretrained weights, outperforms the FCN-32s model significantly in terms of performance and visualization.

The incorporation of skip connections in FCN-8s is crucial for its improved performance. These connections allow the model to capture fine-grained details and high-level semantic information simultaneously by including lower-resolution feature maps in the final prediction layer. This leads to more accurate and refined segmentation results. The skip connections help recover lost spatial information during upsampling and enable the model to leverage both local and global contextual information, improving segmentation performance. With a more complex architecture and deeper layers, FCN-8s can learn intricate representations and capture finer details, contributing to its superior performance compared to FCN-32s.

The observed improvement in IOU from 0.3808 in FCN-32s to 0.4783 in FCN-8s demonstrates the significant impact of skip connections and the deeper architecture in achieving more precise and detailed segmentations. Visually, FCN-8s produces sharper boundaries and better delineation of object classes, thanks to its ability to capture fine details and incorporate both local and global context through the skip connections.

When comparing the visualizations, there may not be a significant difference in the output segmented images. However, upon closer examination, the distinction is evident. FCN-8s performs better than FCN-32s when compared with the ground truth images.

5 Part 4: Class Activation Maps (30%)

In this section, we are going to interpret neural networks decisions with the technique class activation maps (CAM). As discussed in the class, the idea is that one can highlight the image region that is important to the prediction.

The resnet-18 uses global average pooling for downsampling layer 4 features and then applies an FC layer to predict the class probabilities. We select the class with the highest probability as our best guess and we denote the corresponding FC weight as w.

Let $f_4(x, y)$ denote the layer 4 feature at spatial location (x,y). Now we can directly apply the learned FC weight w to $f_4(x, y)$ to get the network prediction for this spatial location $CAM(x, y)$. CAM can be obtained by repeating this for all spatial locations.

You may refer to the paper (http://cnnlocalization.csail.mit.edu/Zhou_Learning_Deep_Features_CVPR_2016_
for more details. In this part, we are going to use the pretrained resnet18 as the backbone.

Task: understand the approach, then 1. For each image, show the top-1 class prediction and probability. 2. For each image, plot the CAM using layer4 features and fc weights (corresponding to the top-1 prediction).

[200]:
```
def get_cls_pred(logit):
    """
    Input:
        logit: (1, 1000) # the predicted logits from resnet18
    Output:
        cls_idx: (1, ) # the class index with highest probability
    """

    # load the imagenet category list
    LABELS_file = 'files/imagenet-simple-labels.json'
    with open(LABELS_file) as f:
        classes = json.load(f)

   ␣
 ↪################################################################################
    # TODO:
    #      1. Use softmax to get the class prediction probability from logits
    #      2. Use torch.sort to get the top-1 class prediction probability␣
 ↪(top1_prob)
    #         and the corresponding class index (top1_idx)
   ␣
 ↪################################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        smax = nn.Softmax(dim=1)
        prob = smax(logit)
        top1_prob, top1_idx = torch.max(prob, dim=1)
        top1_prob = top1_prob.item()

        #print(top1_prob, top1_idx)
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
   ␣
 ↪################################################################################
    #                                END OF YOUR CODE                          ␣
 ↪   #
   ␣
 ↪################################################################################

    # output the prediction
    print('top1 prediction: {:.3f} -> {}'.format(top1_prob, classes[top1_idx]))

    return top1_idx


def returnCAM(feature_conv, weight_fc, idx):
    """
    Input:
        feature_conv: (1, 512, 7, 7) # layer4 feature
        weight_fc: (1000, 512) # fc weight
        idx: (1, ) # predicted class index
    Output:
        output_cam: (256, 256)
    """
    size_upsample = (256, 256)
    bz, nc, h, w = feature_conv.shape
```

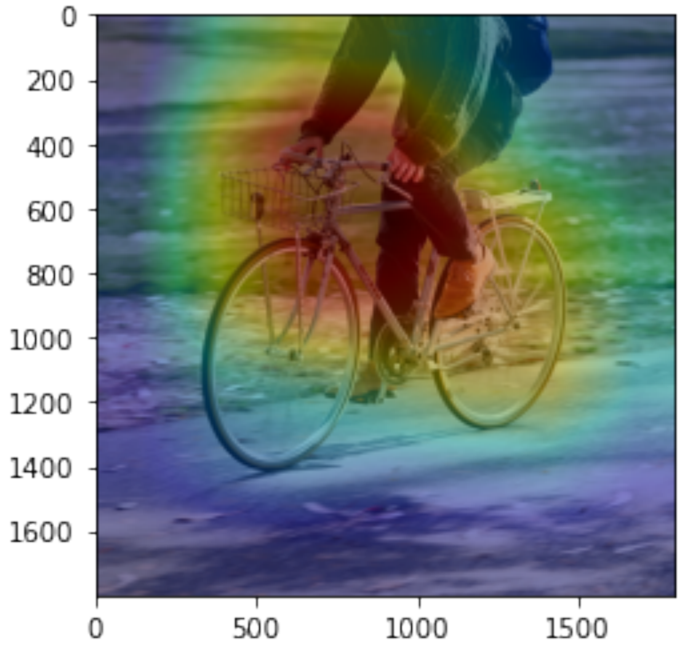
```
␣
↪###########################################################################
    # TODO: Implement CAM
    #     1. the product of the layer4 features and the fc weight corresponding␣
↪to
    #        the top-1 class prediction
    #     2. convert to cam_img of shape (7,7) and value range [0, 255]
␣
↪###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    cam_img = np.matmul(weight_fc[idx].reshape(1, nc), feature_conv.reshape(nc,␣
↪h * w))
    cam_img = (cam_img - np.amin(cam_img)) / (np.amax(cam_img) - np.
↪amin(cam_img)) * 255
    cam_img = cam_img.reshape(h, w)
    cam_img = (cam_img).astype(np.uint8)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
␣
↪###########################################################################
    #                                 END OF YOUR CODE                        ␣
↪   #
␣
↪###########################################################################

    # resize cam image to (256,256)
    output_cam = cv2.resize(cam_img, size_upsample)

    return output_cam
```

[201]:
```
import io
from PIL import Image
from torchvision import transforms
from torch.nn import functional as F
import numpy as np
import cv2
import json
from CAM.resnet import resnet18
import matplotlib.pyplot as plt


# load model
net = resnet18(pretrained=True)
net.eval()
```

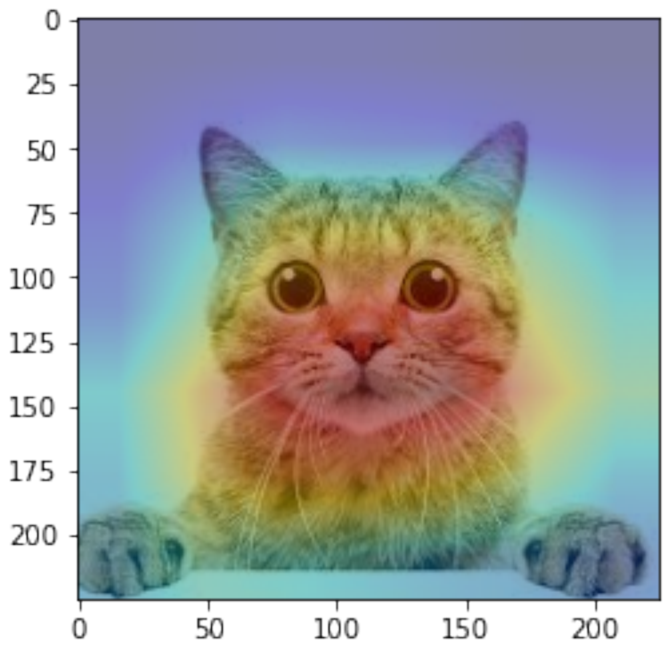
```
# image normalization
preprocess = transforms.Compose([
   transforms.Resize((224,224)),
   transforms.ToTensor(),
   transforms.Normalize(
   mean=[0.485, 0.456, 0.406],
   std=[0.229, 0.224, 0.225])
])

# load the imagenet category list
LABELS_file = 'files/imagenet-simple-labels.json'
with open(LABELS_file) as f:
    classes = json.load(f)

# load test image files/bike.jpg, files/
for image_file in ['files/bike.jpg', 'files/cat.jpg']:
    img = Image.open(image_file)
    img_tensor = preprocess(img)

    # extract predicted logits and layer4 feature
    logits, layer4_feat = net(img_tensor.unsqueeze(0))
    layer4_feat = layer4_feat.detach().numpy()

    # predicted top-1 class, needs to complete the function
    cls_idx = get_cls_pred(logits)

   ␣
 ↪################################################################################
    # TODO: extract the weight of fc layer and convert from torch.tensor to␣
 ↪numpy.array                                                                   #
   ␣
 ↪################################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
#     weight_fc =  net.fc.weight.detach().numpy()
    # weight_fc is of shape (1000, 512)
    weight_fc =  net.fc.weight # weight_fc is of shape (1000, 512)
    weight_fc = weight_fc.detach().numpy()
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
   ␣
 ↪################################################################################
    #                                 END OF YOUR CODE                          ␣
 ↪   #
   ␣
 ↪################################################################################
```

```
    # generate class activation mapping for the top1 prediction
    CAMs = returnCAM(layer4_feat, weight_fc, cls_idx)

    # render the CAM and output
    img = cv2.imread(image_file)
    height, width, _ = img.shape
    heatmap = cv2.applyColorMap(cv2.resize(CAMs,(width, height)), cv2.
↪COLORMAP_JET)
    result = heatmap * 0.3 + img * 0.5
    plt.imshow(result[:,:,::-1]/255)
    plt.show()
```

```
load pretrained weights
top1 prediction: 0.444 -> mountain bike
```



```
top1 prediction: 0.370 -> tabby cat
```


0
25
50
75
100
125
150
175
200
0
50
100
150
200