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.

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.

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.

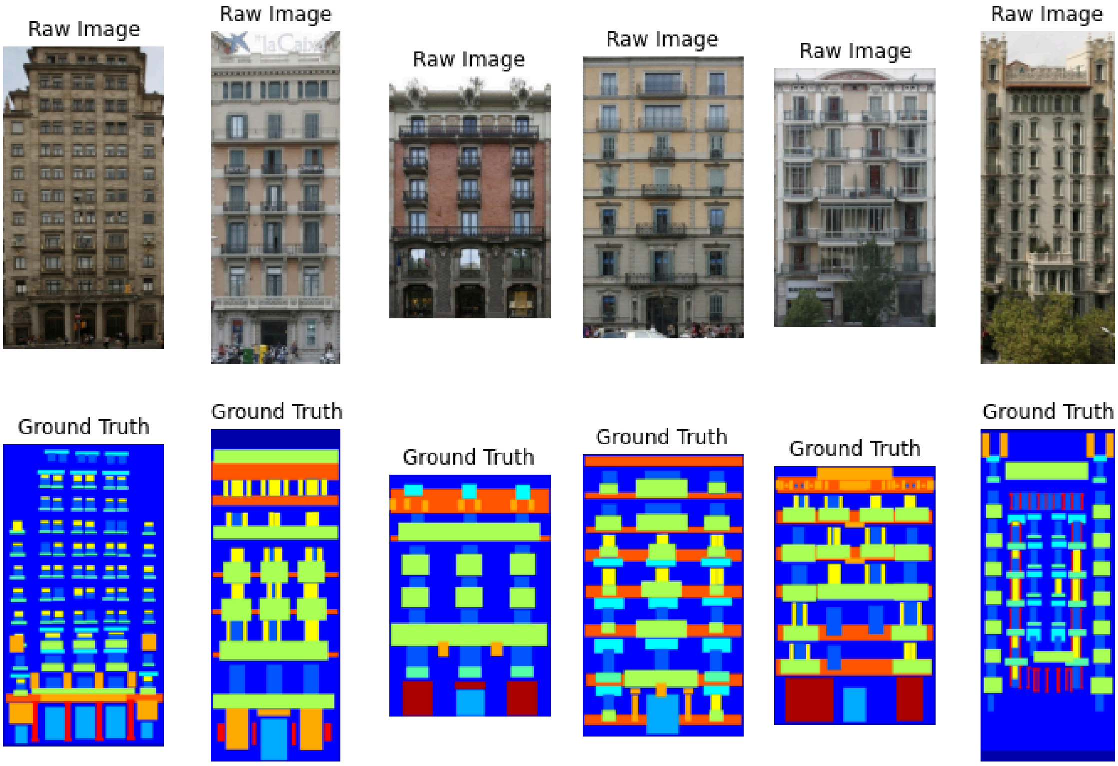
CMP Facade DB include 12 semantic classes:

- 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/

In [ ]:
```
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")
```



Build Dataloader and Set Up Device

In [ ]:
```
# torch.cuda.is_available()
```

In [ ]:
```
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
```

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.

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.

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

In [ ]:
```
import torch.nn.functional as F  # useful stateless functions
def flatten(x):
    N = x.shape[0] # read in N, C, H, W
    return x.view(N, -1)  # "flatten" the C * H * W values into a single vector per image




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.n_class = n_class



        self.conv1_1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, stride=1, padding=100)
        self.relu1_1 = nn.ReLU()
        self.conv1_2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1)
        self.relu1_2 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2, stride=2, ceil_mode=True)
        self.conv2_1 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1)
        self.relu2_1 = nn.ReLU()
        self.conv2_2 = nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, stride=1, padding=1)
        self.relu2_2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2, stride=2, ceil_mode=True)
        self.conv3_1 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=1, padding=1)
        self.relu3_1 = nn.ReLU()
        self.conv3_2 = nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, stride=1, padding=1)
        self.relu3_2 = nn.ReLU()
        self.conv3_3 = nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, stride=1, padding=1)
        self.relu3_3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(2, stride=2, ceil_mode=True)
        self.conv4_1 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, stride=1, padding=1)
        self.relu4_1 = nn.ReLU()
```

```python
        self.conv4_2 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, stride=1, padding=1)
        self.relu4_2 = nn.ReLU()
        self.conv4_3 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, stride=1, padding=1)
        self.relu4_3 = nn.ReLU()
        self.pool4 = nn.MaxPool2d(2, stride=2, ceil_mode=True)
        self.conv5_1 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, stride=1, padding=1)
        self.relu5_1 = nn.ReLU()
        self.conv5_2 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, stride=1, padding=1)
        self.relu5_2 = nn.ReLU()
        self.conv5_3 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, stride=1, padding=1)
        self.relu5_3 = nn.ReLU()
        self.pool5 = nn.MaxPool2d(2, stride=2, ceil_mode=True)
#         self.fc6 = FCN32sConvBlock(kernel_size=7, in_channel=512, out_channel=4096, padding1=0, mid_layers=0, drc
#         self.fc7 = FCN32sConvBlock(kernel_size=1, in_channel=4096, out_channel=4096, padding1=0, mid_layers=0, dr
        self.fc6 = nn.Conv2d(in_channels=512, out_channels=4096, kernel_size=7, stride=1, padding=0)
        self.relu6 = nn.ReLU()
        self.dropout6 = nn.Dropout2d(p=0.5)
        self.fc7 = nn.Conv2d(in_channels=4096, out_channels=4096, kernel_size=1, stride=1, padding=0)
        self.relu7 = nn.ReLU()
        self.dropout7 = nn.Dropout2d(p=0.5)
        self.score = nn.Conv2d(in_channels=4096, out_channels=self.n_class, kernel_size=1, stride=1, padding=0)
        self.transConv1 = nn.ConvTranspose2d(self.n_class, self.n_class, 64, stride=32, 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 FCN32s.                                 #
        ################################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
#         print(f"initial shape:{x.shape}")
        input_image_width, input_image_height = x.shape[3], x.shape[2]
#         x = self.conv1(x)
#         x = self.conv2(x)
#         x = self.conv3(x)
#         x = self.conv4(x)
#         x = self.conv5(x)

        x = self.relu1_1(self.conv1_1(x))
        x = self.relu1_2(self.conv1_2(x))
        x = self.pool1(x)
        x = self.relu2_1(self.conv2_1(x))
```

```
        x = self.relu2_2(self.conv2_2(x))
        x = self.pool2(x)
        x = self.relu3_1(self.conv3_1(x))
        x = self.relu3_2(self.conv3_2(x))
        x = self.relu3_3(self.conv3_3(x))
        x = self.pool3(x)
        x = self.relu4_1(self.conv4_1(x))
        x = self.relu4_2(self.conv4_2(x))
        x = self.relu4_3(self.conv4_3(x))
        x = self.pool4(x)
        x = self.relu5_1(self.conv5_1(x))
        x = self.relu5_2(self.conv5_2(x))
        x = self.relu5_3(self.conv5_3(x))
        x = self.pool5(x)
        x = self.relu6(self.fc6(x))
        x = self.dropout6(x)
        x = self.relu7(self.fc7(x))
        x = self.dropout7(x)
        x = self.score(x)
#         print(f"before t conv: {x.shape}")
        x = self.transConv1(x)

        h = x[..., 19: 19 + input_image_height, 19: 19 + input_image_width]
#         print(h.shape, input_image_height, input_image_width)
        # *****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())
```

1.2 Train FCN-32s from scratch

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

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

best_model = Trainer(
    model32,
    train_loader,
```

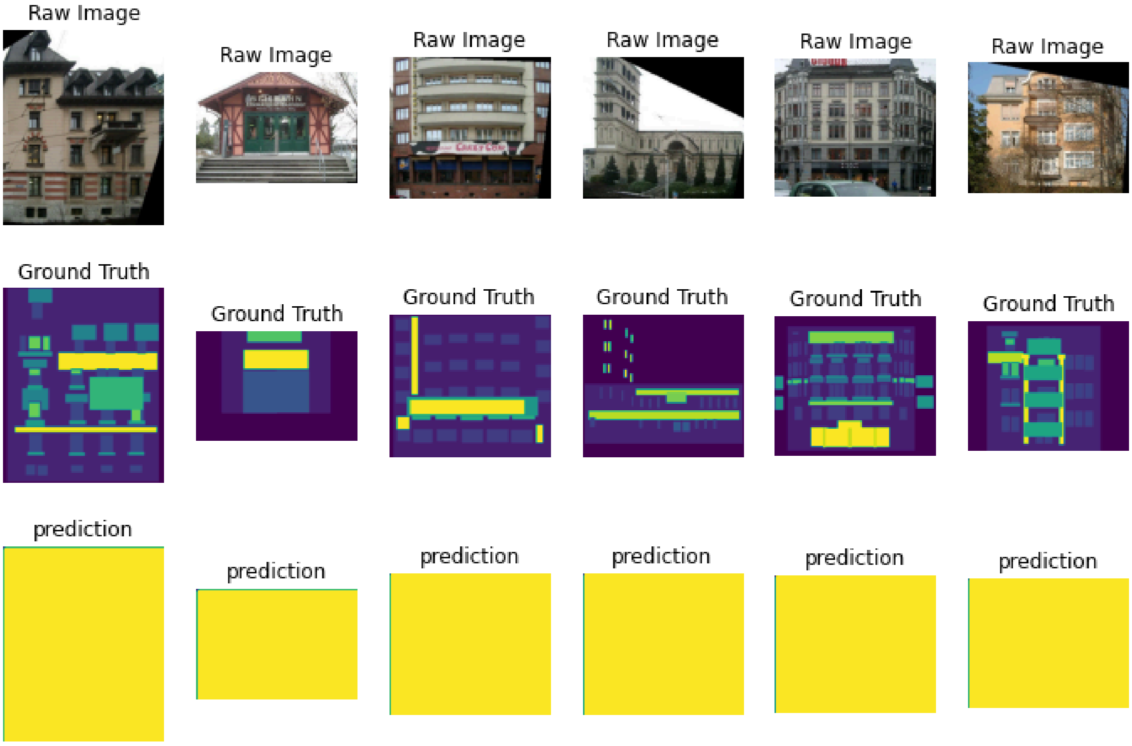
```
    val_loader,
    test_loader,
    num_epochs=10
)
```

```
Init Model
Avg Acc: 0.2307, Mean IoU: 0.01922
Epochs: 0
100 / 500, Current Avg Loss:2.48
200 / 500, Current Avg Loss:2.473
300 / 500, Current Avg Loss:2.462
400 / 500, Current Avg Loss:2.447
Epoch Loss: 2.43, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 1
100 / 500, Current Avg Loss:2.315
200 / 500, Current Avg Loss:2.288
300 / 500, Current Avg Loss:2.253
400 / 500, Current Avg Loss:2.228
Epoch Loss: 2.204, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 2
100 / 500, Current Avg Loss:2.049
200 / 500, Current Avg Loss:2.028
300 / 500, Current Avg Loss:2.018
400 / 500, Current Avg Loss:2.014
Epoch Loss: 2.01, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 3
100 / 500, Current Avg Loss:1.915
200 / 500, Current Avg Loss:1.943
300 / 500, Current Avg Loss:1.949
400 / 500, Current Avg Loss:1.947
Epoch Loss: 1.947, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 4
100 / 500, Current Avg Loss:1.899
200 / 500, Current Avg Loss:1.929
300 / 500, Current Avg Loss:1.925
400 / 500, Current Avg Loss:1.929
Epoch Loss: 1.933, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 5
100 / 500, Current Avg Loss:1.942
200 / 500, Current Avg Loss:1.937
300 / 500, Current Avg Loss:1.936
400 / 500, Current Avg Loss:1.924
Epoch Loss: 1.928, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 6
100 / 500, Current Avg Loss:1.946
200 / 500, Current Avg Loss:1.933
300 / 500, Current Avg Loss:1.939
400 / 500, Current Avg Loss:1.936
Epoch Loss: 1.925, Avg Acc: 0.3432, Mean IoU: 0.02862
Epochs: 7
100 / 500, Current Avg Loss:1.93
200 / 500, Current Avg Loss:1.933
300 / 500, Current Avg Loss:1.925
400 / 500, Current Avg Loss:1.914
Epoch Loss: 1.922, Avg Acc: 0.3433, Mean IoU: 0.02869
Epochs: 8
100 / 500, Current Avg Loss:1.908
200 / 500, Current Avg Loss:1.917
300 / 500, Current Avg Loss:1.925
400 / 500, Current Avg Loss:1.926
Epoch Loss: 1.92, Avg Acc: 0.3452, Mean IoU: 0.02944
Epochs: 9
100 / 500, Current Avg Loss:1.919
200 / 500, Current Avg Loss:1.921
300 / 500, Current Avg Loss:1.925
400 / 500, Current Avg Loss:1.918
Epoch Loss: 1.918, Avg Acc: 0.3475, Mean IoU: 0.03035
Test Acc: 0.3475, Test Mean IoU: 0.03035
```

In [ ]:

```
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).
```



1.3 Train FCN-32s with the pretrained VGG16 weights

In [ ]:
```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
)
```

```
/opt/conda/lib/python3.9/site-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/opt/conda/lib/python3.9/site-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing `weights=VGG16_Weights.IMAGENET1K_V1`. You can also use `weights=VGG16_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

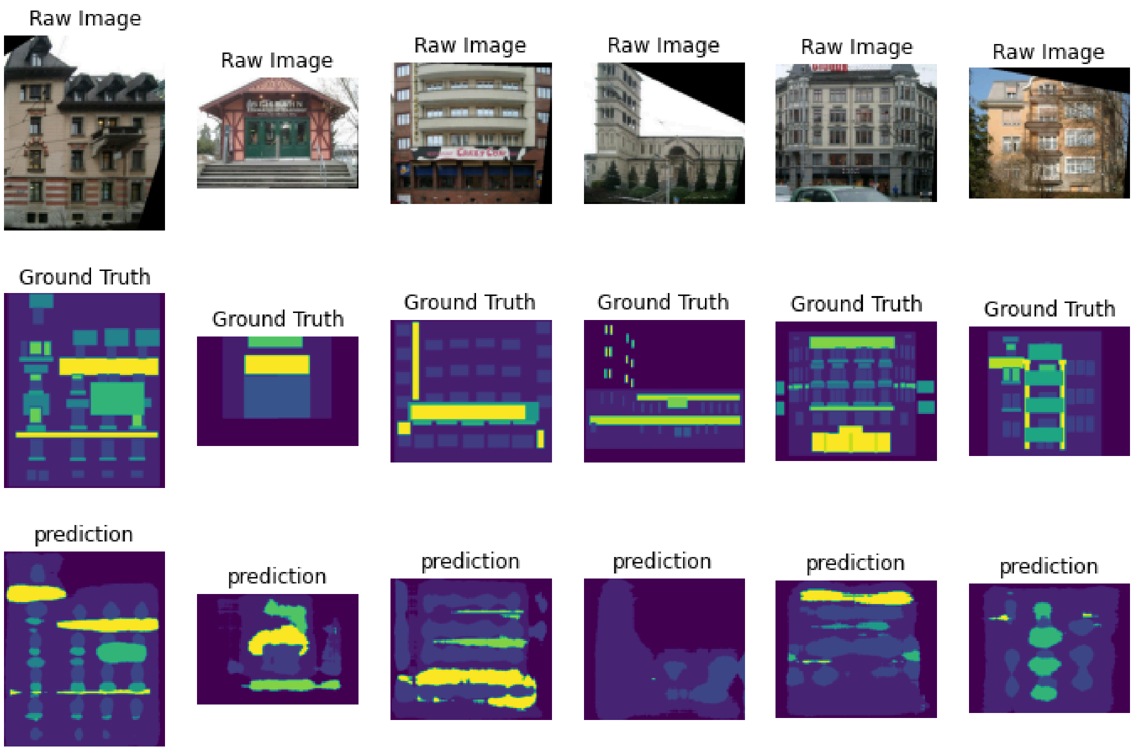
```
Init Model
Avg Acc: 0.2307, Mean IoU: 0.01922
Epochs: 0
100 / 500, Current Avg Loss:1.807
200 / 500, Current Avg Loss:1.736
300 / 500, Current Avg Loss:1.693
400 / 500, Current Avg Loss:1.649
Epoch Loss: 1.609, Avg Acc: 0.4985, Mean IoU: 0.155
Epochs: 1
100 / 500, Current Avg Loss:1.4
200 / 500, Current Avg Loss:1.383
300 / 500, Current Avg Loss:1.381
400 / 500, Current Avg Loss:1.386
Epoch Loss: 1.361, Avg Acc: 0.5593, Mean IoU: 0.182
Epochs: 2
100 / 500, Current Avg Loss:1.208
200 / 500, Current Avg Loss:1.244
300 / 500, Current Avg Loss:1.241
400 / 500, Current Avg Loss:1.244
Epoch Loss: 1.251, Avg Acc: 0.5437, Mean IoU: 0.2059
Epochs: 3
100 / 500, Current Avg Loss:1.182
200 / 500, Current Avg Loss:1.202
300 / 500, Current Avg Loss:1.172
400 / 500, Current Avg Loss:1.172
Epoch Loss: 1.165, Avg Acc: 0.5866, Mean IoU: 0.252
Epochs: 4
100 / 500, Current Avg Loss:1.057
200 / 500, Current Avg Loss:1.053
300 / 500, Current Avg Loss:1.06
400 / 500, Current Avg Loss:1.065
Epoch Loss: 1.058, Avg Acc: 0.6023, Mean IoU: 0.2721
Epochs: 5
100 / 500, Current Avg Loss:0.9998
200 / 500, Current Avg Loss:0.971
300 / 500, Current Avg Loss:0.9702
400 / 500, Current Avg Loss:0.9801
Epoch Loss: 0.9826, Avg Acc: 0.6013, Mean IoU: 0.2687
Epochs: 6
100 / 500, Current Avg Loss:0.9161
200 / 500, Current Avg Loss:0.919
300 / 500, Current Avg Loss:0.9067
400 / 500, Current Avg Loss:0.9103
Epoch Loss: 0.9128, Avg Acc: 0.6259, Mean IoU: 0.2838
Epochs: 7
100 / 500, Current Avg Loss:0.8393
200 / 500, Current Avg Loss:0.843
300 / 500, Current Avg Loss:0.8553
400 / 500, Current Avg Loss:0.8609
Epoch Loss: 0.8497, Avg Acc: 0.6225, Mean IoU: 0.3103
Epochs: 8
100 / 500, Current Avg Loss:0.8299
200 / 500, Current Avg Loss:0.8212
300 / 500, Current Avg Loss:0.8013
400 / 500, Current Avg Loss:0.7865
Epoch Loss: 0.7923, Avg Acc: 0.6, Mean IoU: 0.3186
Epochs: 9
100 / 500, Current Avg Loss:0.7129
200 / 500, Current Avg Loss:0.7315
300 / 500, Current Avg Loss:0.7494
400 / 500, Current Avg Loss:0.7528
Epoch Loss: 0.7505, Avg Acc: 0.643, Mean IoU: 0.3315
Epochs: 10
100 / 500, Current Avg Loss:0.7473
200 / 500, Current Avg Loss:0.7178
300 / 500, Current Avg Loss:0.7084
400 / 500, Current Avg Loss:0.7079
Epoch Loss: 0.705, Avg Acc: 0.6371, Mean IoU: 0.3239
Epochs: 11
100 / 500, Current Avg Loss:0.6303
200 / 500, Current Avg Loss:0.6364
300 / 500, Current Avg Loss:0.6476
400 / 500, Current Avg Loss:0.6598
Epoch Loss: 0.6644, Avg Acc: 0.6506, Mean IoU: 0.3522
Epochs: 12
100 / 500, Current Avg Loss:0.6261
```

```
200 / 500, Current Avg Loss:0.6209
300 / 500, Current Avg Loss:0.6273
400 / 500, Current Avg Loss:0.6359
Epoch Loss: 0.6373, Avg Acc: 0.656, Mean IoU: 0.3518
Epochs: 13
100 / 500, Current Avg Loss:0.6113
200 / 500, Current Avg Loss:0.6116
300 / 500, Current Avg Loss:0.61
400 / 500, Current Avg Loss:0.6136
Epoch Loss: 0.6158, Avg Acc: 0.6619, Mean IoU: 0.3492
Epochs: 14
100 / 500, Current Avg Loss:0.6182
200 / 500, Current Avg Loss:0.6198
300 / 500, Current Avg Loss:0.6132
400 / 500, Current Avg Loss:0.6026
Epoch Loss: 0.6, Avg Acc: 0.6611, Mean IoU: 0.3525
Epochs: 15
100 / 500, Current Avg Loss:0.5851
200 / 500, Current Avg Loss:0.5779
300 / 500, Current Avg Loss:0.5827
400 / 500, Current Avg Loss:0.5843
Epoch Loss: 0.5788, Avg Acc: 0.6673, Mean IoU: 0.3659
Epochs: 16
100 / 500, Current Avg Loss:0.5447
200 / 500, Current Avg Loss:0.5433
300 / 500, Current Avg Loss:0.5551
400 / 500, Current Avg Loss:0.5565
Epoch Loss: 0.5564, Avg Acc: 0.6648, Mean IoU: 0.3622
Epochs: 17
100 / 500, Current Avg Loss:0.5288
200 / 500, Current Avg Loss:0.5418
300 / 500, Current Avg Loss:0.5377
400 / 500, Current Avg Loss:0.5352
Epoch Loss: 0.5331, Avg Acc: 0.6663, Mean IoU: 0.369
Epochs: 18
100 / 500, Current Avg Loss:0.5079
200 / 500, Current Avg Loss:0.5107
300 / 500, Current Avg Loss:0.5277
400 / 500, Current Avg Loss:0.5251
Epoch Loss: 0.5191, Avg Acc: 0.6744, Mean IoU: 0.3658
Epochs: 19
100 / 500, Current Avg Loss:0.5001
200 / 500, Current Avg Loss:0.5056
300 / 500, Current Avg Loss:0.5036
400 / 500, Current Avg Loss:0.5105
Epoch Loss: 0.5088, Avg Acc: 0.6747, Mean IoU: 0.3697
Test Acc: 0.6747, Test Mean IoU: 0.3697
```

In [ ]:

```python
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).
```



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.

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

FCN-8s architecture:

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

As you can see, the structure of this part is the same as FCN-32s

- [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 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.

"Architecture Graph" "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]

In [ ]:

```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.n_class = n_class
        self.conv1_1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, stride=1, padding=100)
        self.relu1_1 = nn.ReLU()
        self.conv1_2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1)
        self.relu1_2 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2, stride=2, ceil_mode=True)
        self.conv2_1 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1)
        self.relu2_1 = nn.ReLU()
        self.conv2_2 = nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, stride=1, padding=1)
        self.relu2_2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2, stride=2, ceil_mode=True)
        self.conv3_1 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=1, padding=1)
        self.relu3_1 = nn.ReLU()
        self.conv3_2 = nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, stride=1, padding=1)
        self.relu3_2 = nn.ReLU()
        self.conv3_3 = nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, stride=1, padding=1)
        self.relu3_3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(2, stride=2, ceil_mode=True)
```

```python
        self.conv4_1 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, stride=1, padding=1)
        self.relu4_1 = nn.ReLU()
        self.conv4_2 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, stride=1, padding=1)
        self.relu4_2 = nn.ReLU()
        self.conv4_3 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, stride=1, padding=1)
        self.relu4_3 = nn.ReLU()
        self.pool4 = nn.MaxPool2d(2, stride=2, ceil_mode=True)
        self.conv5_1 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, stride=1, padding=1)
        self.relu5_1 = nn.ReLU()
        self.conv5_2 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, stride=1, padding=1)
        self.relu5_2 = nn.ReLU()
        self.conv5_3 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, stride=1, padding=1)
        self.relu5_3 = nn.ReLU()
        self.pool5 = nn.MaxPool2d(2, stride=2, ceil_mode=True)
        self.fc6 = nn.Conv2d(in_channels=512, out_channels=4096, kernel_size=7, stride=1, padding=0)
        self.relu6 = nn.ReLU()
        self.dropout6 = nn.Dropout2d(p=0.5)
        self.fc7 = nn.Conv2d(in_channels=4096, out_channels=4096, kernel_size=1, stride=1, padding=0)
        self.relu7 = nn.ReLU()
        self.dropout7 = nn.Dropout2d(p=0.5)
        self.score = nn.Conv2d(in_channels=4096, out_channels=self.n_class, kernel_size=1, stride=1, padding=0)

        # additional score pool
        self.scorePool3 = nn.Conv2d(in_channels=256, out_channels=self.n_class, kernel_size=1, stride=1, padding=0
        self.scorePool4 = nn.Conv2d(in_channels=512, out_channels=self.n_class, kernel_size=1, stride=1, padding=0
        # transpose conv
        self.transConv1 = nn.ConvTranspose2d(self.n_class, self.n_class, kernel_size=4, stride=2, bias=False)
        self.transConv2 = nn.ConvTranspose2d(self.n_class, self.n_class, kernel_size=4, stride=2, bias=False)
        self.transConv3 = nn.ConvTranspose2d(self.n_class, self.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)*****
        input_image_width, input_image_height = x.shape[3], x.shape[2]

        x = self.relu1_1(self.conv1_1(x))
        x = self.relu1_2(self.conv1_2(x))
        x = self.pool1(x)
        x = self.relu2_1(self.conv2_1(x))
        x = self.relu2_2(self.conv2_2(x))
```

```python
        x = self.pool2(x)
        x = self.relu3_1(self.conv3_1(x))
        x = self.relu3_2(self.conv3_2(x))
        x = self.relu3_3(self.conv3_3(x))
        x = self.pool3(x)
        p3_score = x.clone()
        p3_score = self.scorePool3(p3_score)
        x = self.relu4_1(self.conv4_1(x))
        x = self.relu4_2(self.conv4_2(x))
        x = self.relu4_3(self.conv4_3(x))
        x = self.pool4(x)
        p4_score = x.clone()
        p4_score = self.scorePool4(p4_score)
        x = self.relu5_1(self.conv5_1(x))
        x = self.relu5_2(self.conv5_2(x))
        x = self.relu5_3(self.conv5_3(x))
        x = self.pool5(x)
        x = self.relu6(self.fc6(x))
        x = self.dropout6(x)
        x = self.relu7(self.fc7(x))
        x = self.dropout7(x)
        x = self.score(x)

        # upscore 1
        x = self.transConv1(x)

        # upscore 2
        upscore1_width, upscore1_height = x.shape[3], x.shape[2]
        p4_score = p4_score[..., 5: 5 + upscore1_height, 5: 5 + upscore1_width]
        x = self.transConv2(x + p4_score)

        # upscore 3
        upscore2_width, upscore2_height = x.shape[3], x.shape[2]
        p3_score = p3_score[..., 9: 9 + upscore2_height, 9: 9 + upscore2_width]
#         print("x.shape, p3_score.shape")
#         print(x.shape, p3_score.shape)
        x = self.transConv3(x + p3_score) # upscore3
#         print("cropping")
#         print(x.shape, input_image_height, input_image_width)
        h = x[..., 31: 31 + input_image_height, 31: 31 + input_image_width]

#         print(h.shape, input_image_height, input_image_width)


        # *****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()))
```

In [ ]:

```
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
)
```

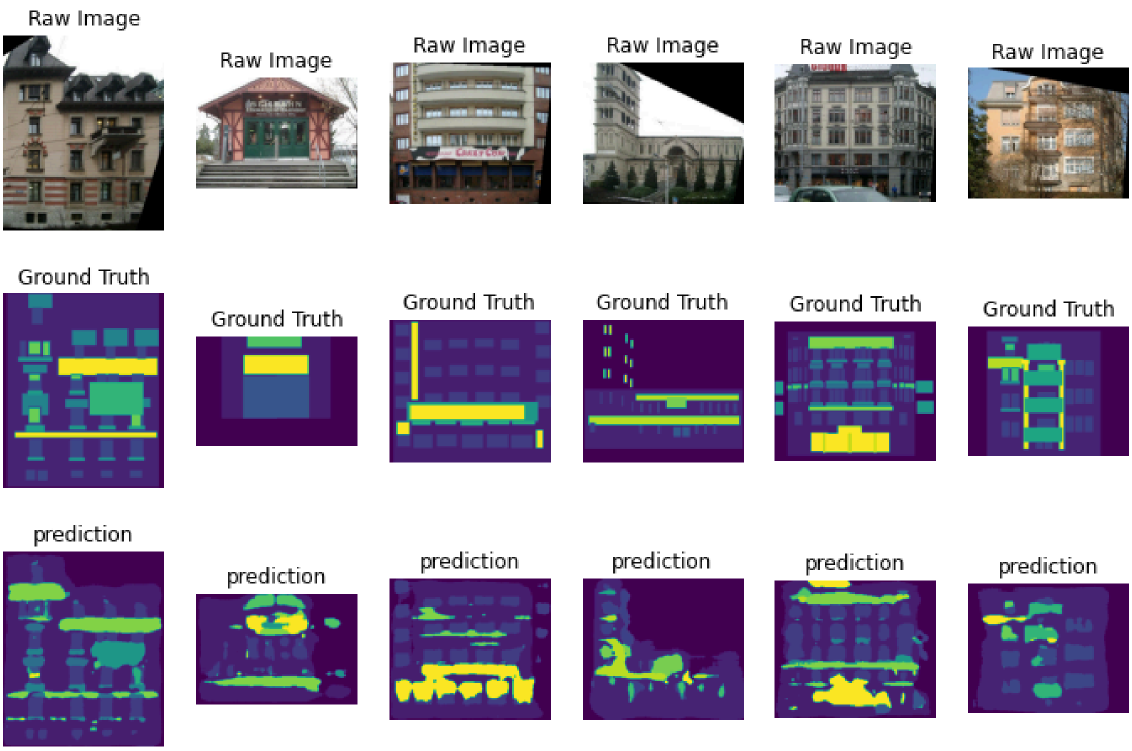
```
Init Model
Avg Acc: 0.2307, Mean IoU: 0.01922
Epochs: 0
100 / 500, Current Avg Loss:1.633
200 / 500, Current Avg Loss:1.418
300 / 500, Current Avg Loss:1.316
400 / 500, Current Avg Loss:1.259
Epoch Loss: 1.208, Avg Acc: 0.6182, Mean IoU: 0.3217
Epochs: 1
100 / 500, Current Avg Loss:0.9355
200 / 500, Current Avg Loss:0.9476
300 / 500, Current Avg Loss:0.9347
400 / 500, Current Avg Loss:0.9365
Epoch Loss: 0.9432, Avg Acc: 0.6591, Mean IoU: 0.3524
Epochs: 2
100 / 500, Current Avg Loss:0.8212
200 / 500, Current Avg Loss:0.8533
300 / 500, Current Avg Loss:0.8519
400 / 500, Current Avg Loss:0.8429
Epoch Loss: 0.8603, Avg Acc: 0.6592, Mean IoU: 0.3631
Epochs: 3
100 / 500, Current Avg Loss:0.8373
200 / 500, Current Avg Loss:0.8263
300 / 500, Current Avg Loss:0.8051
400 / 500, Current Avg Loss:0.7997
Epoch Loss: 0.7946, Avg Acc: 0.6941, Mean IoU: 0.3907
Epochs: 4
100 / 500, Current Avg Loss:0.7116
200 / 500, Current Avg Loss:0.7067
300 / 500, Current Avg Loss:0.7111
400 / 500, Current Avg Loss:0.7123
Epoch Loss: 0.7183, Avg Acc: 0.6924, Mean IoU: 0.4056
Epochs: 5
100 / 500, Current Avg Loss:0.6748
200 / 500, Current Avg Loss:0.6753
300 / 500, Current Avg Loss:0.657
400 / 500, Current Avg Loss:0.6529
Epoch Loss: 0.6466, Avg Acc: 0.6974, Mean IoU: 0.4148
Epochs: 6
100 / 500, Current Avg Loss:0.6009
200 / 500, Current Avg Loss:0.5915
300 / 500, Current Avg Loss:0.5868
400 / 500, Current Avg Loss:0.6042
Epoch Loss: 0.6055, Avg Acc: 0.7018, Mean IoU: 0.432
Epochs: 7
100 / 500, Current Avg Loss:0.5509
200 / 500, Current Avg Loss:0.5574
300 / 500, Current Avg Loss:0.5501
400 / 500, Current Avg Loss:0.5516
Epoch Loss: 0.5516, Avg Acc: 0.7182, Mean IoU: 0.4418
Epochs: 8
100 / 500, Current Avg Loss:0.4993
200 / 500, Current Avg Loss:0.485
300 / 500, Current Avg Loss:0.4887
400 / 500, Current Avg Loss:0.4922
Epoch Loss: 0.4941, Avg Acc: 0.7081, Mean IoU: 0.4423
Epochs: 9
100 / 500, Current Avg Loss:0.5137
200 / 500, Current Avg Loss:0.4978
300 / 500, Current Avg Loss:0.4777
400 / 500, Current Avg Loss:0.4897
Epoch Loss: 0.4887, Avg Acc: 0.726, Mean IoU: 0.4616
Epochs: 10
100 / 500, Current Avg Loss:0.4093
200 / 500, Current Avg Loss:0.4236
300 / 500, Current Avg Loss:0.4334
400 / 500, Current Avg Loss:0.4342
Epoch Loss: 0.4362, Avg Acc: 0.7323, Mean IoU: 0.4485
Epochs: 11
100 / 500, Current Avg Loss:0.4095
200 / 500, Current Avg Loss:0.4126
300 / 500, Current Avg Loss:0.4137
400 / 500, Current Avg Loss:0.4135
Epoch Loss: 0.4103, Avg Acc: 0.7311, Mean IoU: 0.4556
Epochs: 12
100 / 500, Current Avg Loss:0.3896
```

```
200 / 500, Current Avg Loss:0.3771
300 / 500, Current Avg Loss:0.3805
400 / 500, Current Avg Loss:0.3769
Epoch Loss: 0.3804, Avg Acc: 0.731, Mean IoU: 0.4572
Epochs: 13
100 / 500, Current Avg Loss:0.376
200 / 500, Current Avg Loss:0.3541
300 / 500, Current Avg Loss:0.365
400 / 500, Current Avg Loss:0.3662
Epoch Loss: 0.3806, Avg Acc: 0.726, Mean IoU: 0.4442
Epochs: 14
100 / 500, Current Avg Loss:0.4873
200 / 500, Current Avg Loss:0.4358
300 / 500, Current Avg Loss:0.4138
400 / 500, Current Avg Loss:0.3987
Epoch Loss: 0.3907, Avg Acc: 0.7356, Mean IoU: 0.4693
Epochs: 15
100 / 500, Current Avg Loss:0.3143
200 / 500, Current Avg Loss:0.3372
300 / 500, Current Avg Loss:0.3393
400 / 500, Current Avg Loss:0.3344
Epoch Loss: 0.3291, Avg Acc: 0.7328, Mean IoU: 0.4582
Epochs: 16
100 / 500, Current Avg Loss:0.2924
200 / 500, Current Avg Loss:0.3049
300 / 500, Current Avg Loss:0.3058
400 / 500, Current Avg Loss:0.3085
Epoch Loss: 0.3113, Avg Acc: 0.7383, Mean IoU: 0.472
Epochs: 17
100 / 500, Current Avg Loss:0.2983
200 / 500, Current Avg Loss:0.2857
300 / 500, Current Avg Loss:0.2957
400 / 500, Current Avg Loss:0.2978
Epoch Loss: 0.314, Avg Acc: 0.7237, Mean IoU: 0.4536
Epochs: 18
100 / 500, Current Avg Loss:0.301
200 / 500, Current Avg Loss:0.3018
300 / 500, Current Avg Loss:0.2977
400 / 500, Current Avg Loss:0.2962
Epoch Loss: 0.2965, Avg Acc: 0.7425, Mean IoU: 0.4852
Epochs: 19
100 / 500, Current Avg Loss:0.2905
200 / 500, Current Avg Loss:0.2779
300 / 500, Current Avg Loss:0.2855
400 / 500, Current Avg Loss:0.2786
Epoch Loss: 0.2817, Avg Acc: 0.7436, Mean IoU: 0.4722
Test Acc: 0.7425, Test Mean IoU: 0.4852
```

In [ ]:

```
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).
```



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.

Your Answer:

The FCN-32s with pretrained weights performs much better than the one training from scratch in that it converges faster and achieves a higher IOU. Pretrained weights help a lot. Potential reasons include:

- The pretrained weights expedite convergence by providing an initialization that is closed to a better local optimum (or even the global optimum).
- The pretrained weights can boost generalization of long-tail samples by leveraging learned features from a large dataset like ImageNet.
- The pretrained weights can regularize the weights, mitigating overfitting risks by enabling the model to excel across diverse datasets and more real-world scenarios.

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.

Your Answer:

FCN-8s is much better than FCN-32s in terms of the evaluation IOU performance. A potential reason can be: The incorporation of skip connections in FCN-8s merges the predictions from the 16-times and 8-times down-sampled features, and hence enabling it to capture both low-level and high-level features from different network depths. Those merged features lead to more fine-grained segmentation results.

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_paper.pdf) 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).

In [ ]:
```
import torch

# b=torch.rand([[1,2],[3,2]])
a=torch.Tensor([[1,2],[3,2]])
a.size()
a=a.unsqueeze(0)
a.size()
```

Out[ ]:
```
torch.Size([1, 2, 2])
```

In [ ]:
```
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)*****
    def softmax(x):
        x_exp = torch.exp(x)
        return x_exp / torch.sum(x_exp)
    logit_sm = softmax(logit)
    sorted_sm, indices = torch.sort(logit_sm, dim=1, descending=True)
#     print(sorted_sm, indices, sorted_sm.shape, indices.shape)
    top1_prob = sorted_sm[0, 0]
    top1_idx = indices[0, 0]

    # *****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)
```

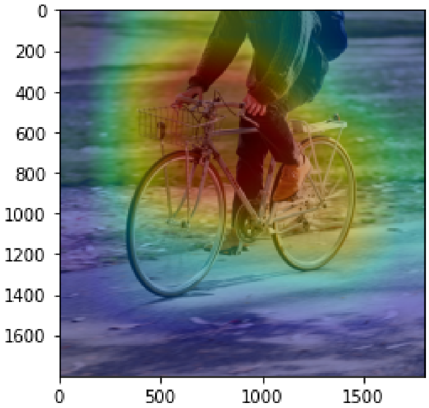
```
    bz, nc, h, w = feature_conv.shape

    ################################################################################
    # 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)*****
    weight_sel = weight_fc[idx]
    class_map = weight_sel[None, ...] @ feature_conv.reshape((nc, h * w))
    class_map = class_map.reshape(h, w)

    # Normalize the class activation map
    class_map -= np.min(class_map)
    class_map /= np.max(class_map)

    # Convert to cam_img of shape (7,7) and value range [0, 255]
    cam_img = (class_map * 255).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
```

In [ ]:

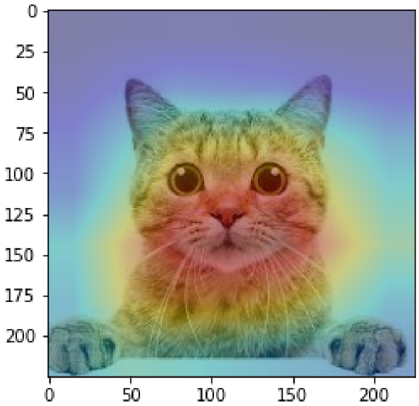
```
import io
from PIL import Image
from torchvision import transforms
from torch.nn import functional as F
import numpy as np
import cv2
import json
from CAM.resnet import resnet18
import matplotlib.pyplot as plt


# 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
    ################################################################################
```

```python
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    weight_fc = net.fc.weight.data # weight_fc is of shape (1000, 512)
    weight_fc = weight_fc.detach().cpu().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: 0.443 -> mountain bike
```




```
top1 prediction: 0.368 -> tabby cat
```



In [ ]: