Assignment 6: Pytorch Segmentation

For this assignment, we're going to use Deep Learning for a new task: semantic segmentation.

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/

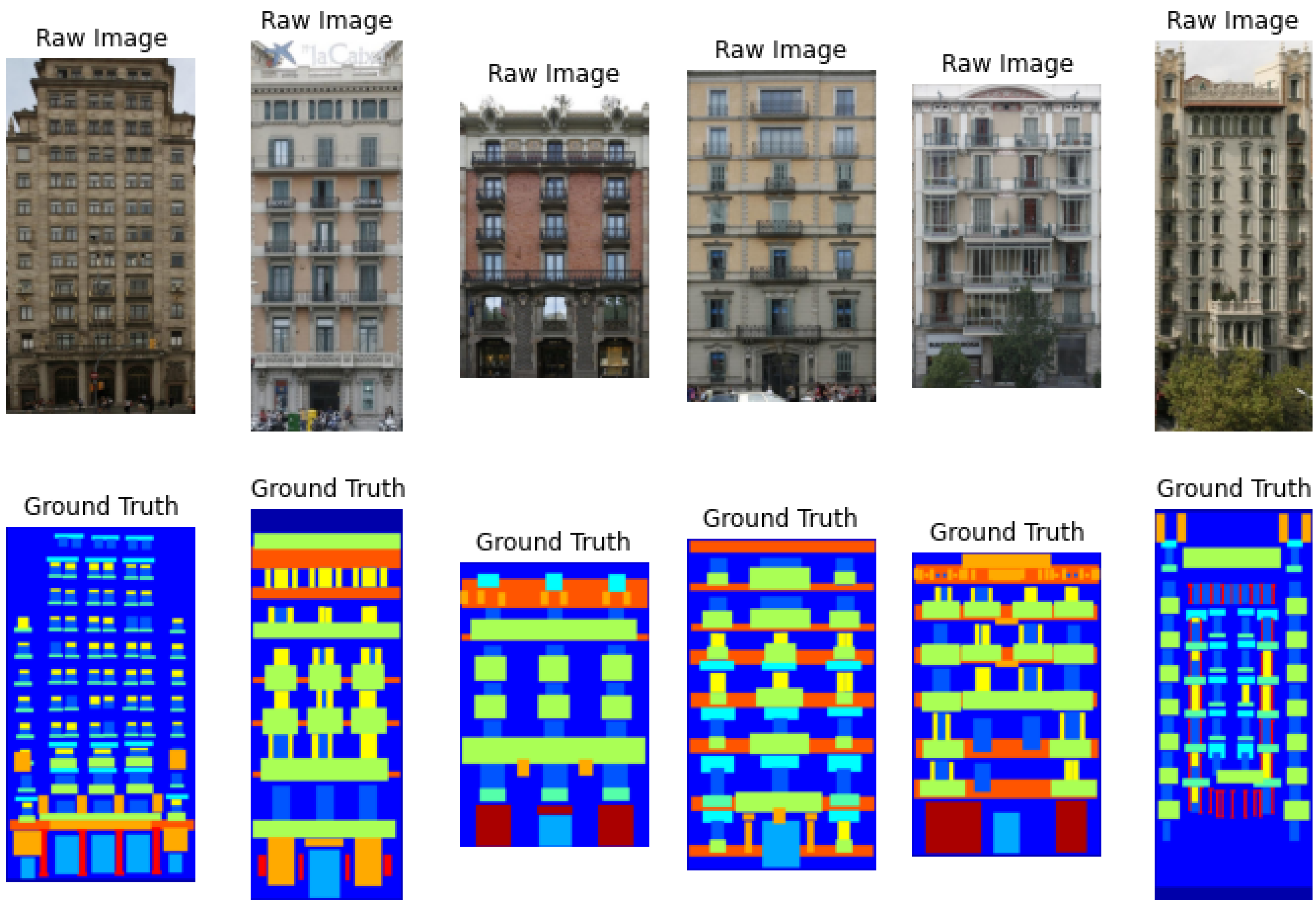
In [1]:
```
import matplotlib.pyplot as plt
import numpy as np

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")
```



Build Dataloader and Set Up Device

In [2]:

```
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(
```

```python
            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 (30%)

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 [7]:
```
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, stride=1, padding=100)
        self.relu1_1 = nn.ReLU(inplace=True)
        self.conv1_2 = nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1)
        self.relu1_2 = nn.ReLU(inplace=True)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2, ceil_mode=True)

        self.conv2_1 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.relu2_1 = nn.ReLU(inplace=True)
        self.conv2_2 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1)
        self.relu2_2 = nn.ReLU(inplace=True)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2, ceil_mode=True)


        self.conv3_1 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)
        self.relu3_1 = nn.ReLU(inplace=True)
        self.conv3_2 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)
        self.relu3_2 = nn.ReLU(inplace=True)
        self.conv3_3 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)
        self.relu3_3 = nn.ReLU(inplace=True)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2, ceil_mode=True)


        self.conv4_1 = nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1)
        self.relu4_1 = nn.ReLU(inplace=True)
        self.conv4_2 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.relu4_2 = nn.ReLU(inplace=True)
        self.conv4_3 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.relu4_3 = nn.ReLU(inplace=True)
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=2, ceil_mode=True)


        self.conv5_1 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.relu5_1 = nn.ReLU(inplace=True)
        self.conv5_2 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.relu5_2 = nn.ReLU(inplace=True)
        self.conv5_3 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.relu5_3 = nn.ReLU(inplace=True)
        self.pool5 = nn.MaxPool2d(kernel_size=2, stride=2, ceil_mode=True)


        self.fc6 = nn.Conv2d(512, 4096, kernel_size=7, stride=1, padding=0)
        self.relu_fc6 = nn.ReLU(inplace=True)
        self.dropout_fc6 = nn.Dropout2d(0.5)
        self.fc7 = nn.Conv2d(4096, 4096, kernel_size=1, stride=1, padding=0)
        self.relu_fc7 = nn.ReLU(inplace=True)
        self.dropout_fc7 = nn.Dropout2d(0.5)
        self.score = nn.Conv2d(4096, n_class, kernel_size=1, stride=1, padding=0)


        self.transpose_conv_1 = nn.ConvTranspose2d(n_class, n_class, kernel_size=64, str


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

```python
        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)*****
        input_height, input_width = x.size()[2], x.size()[3]

        x = self.conv1_1(x)
        x = self.relu1_1(x)
        x = self.conv1_2(x)
        x = self.relu1_2(x)
        x = self.pool1(x)


        x = self.conv2_1(x)
        x = self.relu2_1(x)
        x = self.conv2_2(x)
        x = self.relu2_2(x)
        x = self.pool2(x)


        x = self.conv3_1(x)
        x = self.relu3_1(x)
        x = self.conv3_2(x)
        x = self.relu3_2(x)
        x = self.conv3_3(x)
        x = self.relu3_3(x)
        x = self.pool3(x)


        x = self.conv4_1(x)
        x = self.relu4_1(x)
        x = self.conv4_2(x)
        x = self.relu4_2(x)
        x = self.conv4_3(x)
        x = self.relu4_3(x)
```

```python
        x = self.pool4(x)


        x = self.conv5_1(x)
        x = self.relu5_1(x)
        x = self.conv5_2(x)
        x = self.relu5_2(x)
        x = self.conv5_3(x)
        x = self.relu5_3(x)
        x = self.pool5(x)


        x = self.fc6(x)
        x = self.relu_fc6(x)
        x = self.dropout_fc6(x)
        x = self.fc7(x)
        x = self.relu_fc7(x)
        x = self.dropout_fc7(x)
        x = self.score(x)


        x = self.transpose_conv_1(x)

        crop_height = (x.size()[2] - input_height) // 2
        crop_width = (x.size()[3] - input_width) // 2

        x = x[:, :, crop_height:crop_height + input_height, crop_width:crop_width + inpu


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

        return x


    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

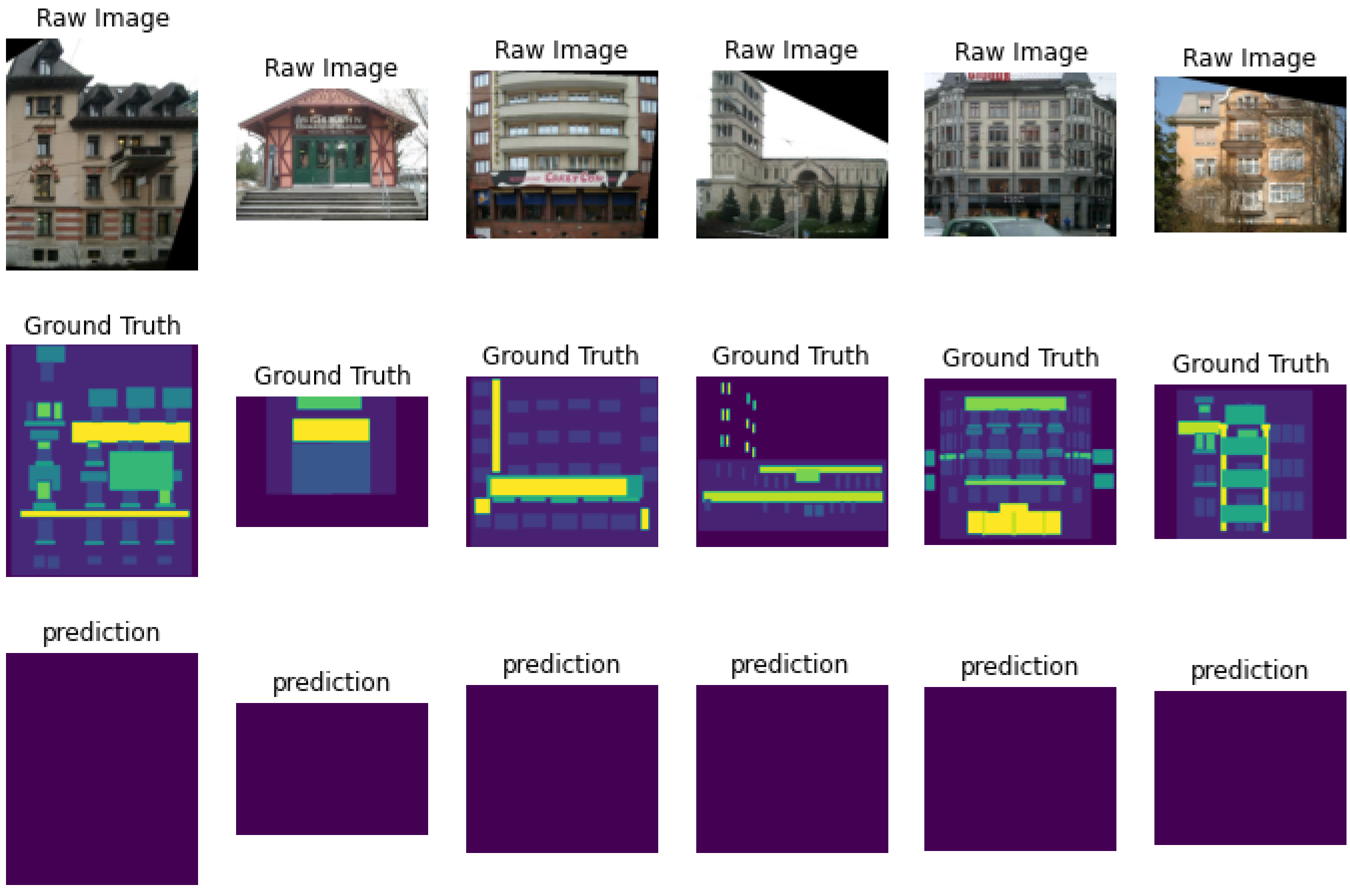
In [8]:
```
from FCN.trainer import Trainer

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
Epoch Loss: 2.428, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 1
Epoch Loss: 2.195, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 2
Epoch Loss: 1.999, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 3
Epoch Loss: 1.938, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 4
Epoch Loss: 1.926, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 5
Epoch Loss: 1.923, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 6
Epoch Loss: 1.921, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 7
Epoch Loss: 1.919, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 8
Epoch Loss: 1.918, Avg Acc: 0.3431, Mean IoU: 0.02859
Epochs: 9
Epoch Loss: 1.917, Avg Acc: 0.3431, Mean IoU: 0.02859
Test Acc: 0.3431, Test Mean IoU: 0.02859
```

In [5]:
```
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 [8]:
```
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
)
```

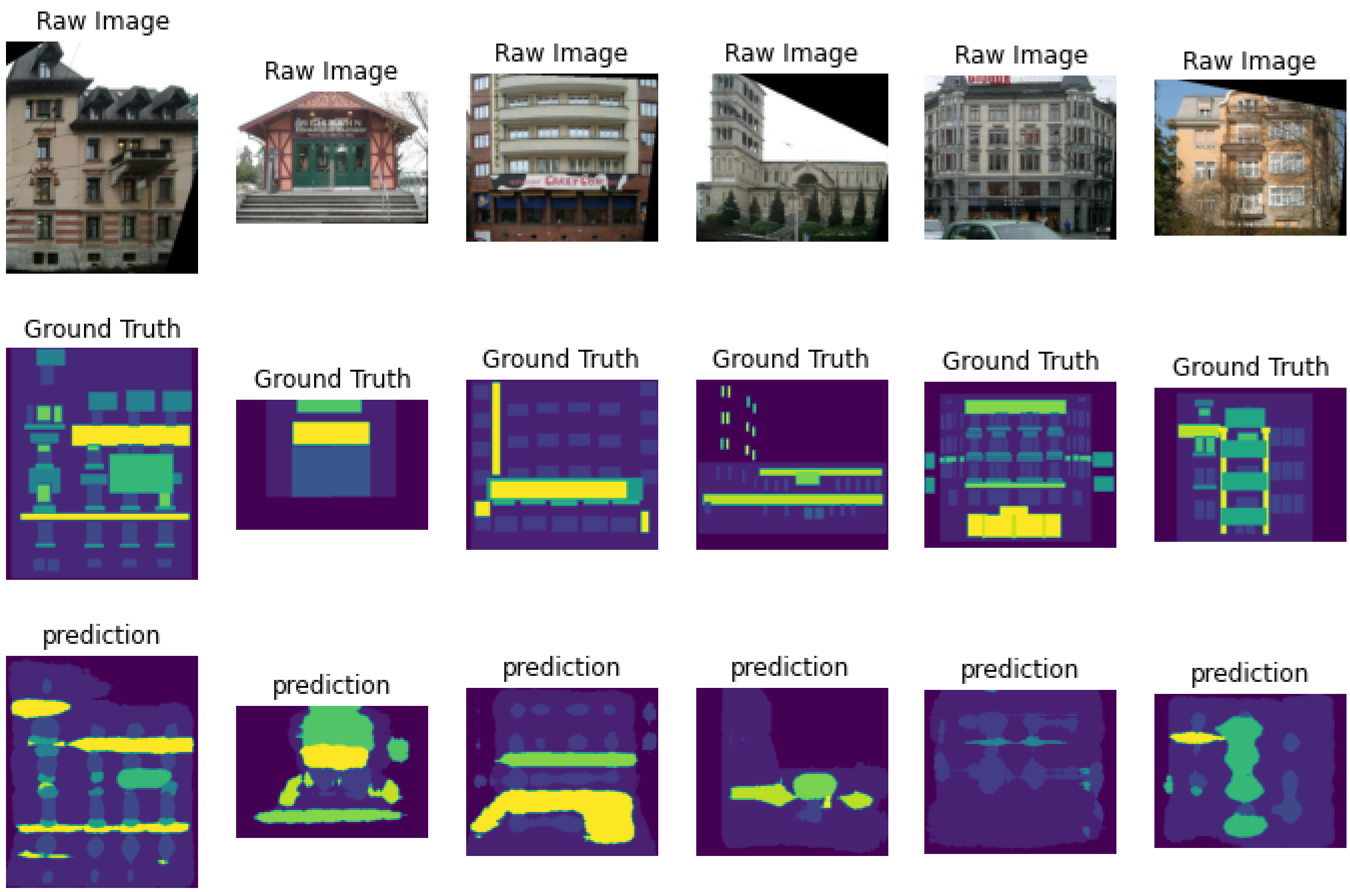
```
/opt/conda/lib/python3.9/site-packages/torchvision/models/_utils.py:208: UserWarning: Th
e parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, plea
se use 'weights' instead.
  warnings.warn(
/opt/conda/lib/python3.9/site-packages/torchvision/models/_utils.py:223: UserWarning: Ar
guments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and m
ay be removed in the future. The current behavior is equivalent to passing `weights=VGG1
6_Weights.IMAGENET1K_V1`. You can also use `weights=VGG16_Weights.DEFAULT` to get the mo
st up-to-date weights.
  warnings.warn(msg)
Init Model
Avg Acc: 0.2307, Mean IoU: 0.01922
Epochs: 0
Epoch Loss: 1.618, Avg Acc: 0.4962, Mean IoU: 0.1315
Epochs: 1
```

```
Epoch Loss: 1.383, Avg Acc: 0.5274, Mean IoU: 0.185
Epochs: 2
Epoch Loss: 1.266, Avg Acc: 0.5397, Mean IoU: 0.2097
Epochs: 3
Epoch Loss: 1.173, Avg Acc: 0.566, Mean IoU: 0.2625
Epochs: 4
Epoch Loss: 1.103, Avg Acc: 0.5761, Mean IoU: 0.2476
Epochs: 5
Epoch Loss: 1.01, Avg Acc: 0.5774, Mean IoU: 0.2624
Epochs: 6
Epoch Loss: 0.9324, Avg Acc: 0.6001, Mean IoU: 0.2851
Epochs: 7
Epoch Loss: 0.8832, Avg Acc: 0.5999, Mean IoU: 0.2917
Epochs: 8
Epoch Loss: 0.8344, Avg Acc: 0.5861, Mean IoU: 0.2773
Epochs: 9
Epoch Loss: 0.7936, Avg Acc: 0.5994, Mean IoU: 0.302
Epochs: 10
Epoch Loss: 0.7533, Avg Acc: 0.6046, Mean IoU: 0.3052
Epochs: 11
Epoch Loss: 0.7186, Avg Acc: 0.6292, Mean IoU: 0.3087
Epochs: 12
Epoch Loss: 0.6922, Avg Acc: 0.616, Mean IoU: 0.3061
Epochs: 13
Epoch Loss: 0.6536, Avg Acc: 0.6293, Mean IoU: 0.3289
Epochs: 14
Epoch Loss: 0.6427, Avg Acc: 0.6298, Mean IoU: 0.3142
Epochs: 15
Epoch Loss: 0.6135, Avg Acc: 0.6301, Mean IoU: 0.3327
Epochs: 16
Epoch Loss: 0.593, Avg Acc: 0.6419, Mean IoU: 0.336
Epochs: 17
Epoch Loss: 0.5676, Avg Acc: 0.6474, Mean IoU: 0.3524
Epochs: 18
Epoch Loss: 0.5507, Avg Acc: 0.6436, Mean IoU: 0.3379
Epochs: 19
Epoch Loss: 0.5344, Avg Acc: 0.6493, Mean IoU: 0.342
Test Acc: 0.6474, Test Mean IoU: 0.3524
```

In [10]:

```
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(40%)

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 [3]:

```
import torch.nn as nn

def crop(tensor_to_crop, tensor_to_match, final=False):
```

```python
    cropy = tensor_to_crop.size()[2] - tensor_to_match.size()[2]
    cropx = tensor_to_crop.size()[3] - tensor_to_match.size()[3]
    if final:
        startx = cropx // 2 - 31
        starty = cropy // 2 - 31
    else:
        startx = cropx // 2
        starty = cropy // 2
    return tensor_to_crop[:, :, starty:starty + tensor_to_match.size()[2], startx:startx



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, stride=1, padding=100)
        self.relu1_1 = nn.ReLU(inplace=True)
        self.conv1_2 = nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1)
        self.relu1_2 = nn.ReLU(inplace=True)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2, ceil_mode=True)


        self.conv2_1 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.relu2_1 = nn.ReLU(inplace=True)
        self.conv2_2 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1)
        self.relu2_2 = nn.ReLU(inplace=True)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2, ceil_mode=True)



        self.conv3_1 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)
        self.relu3_1 = nn.ReLU(inplace=True)
        self.conv3_2 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)
        self.relu3_2 = nn.ReLU(inplace=True)
        self.conv3_3 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)
        self.relu3_3 = nn.ReLU(inplace=True)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2, ceil_mode=True)



        self.conv4_1 = nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1)
        self.relu4_1 = nn.ReLU(inplace=True)
        self.conv4_2 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.relu4_2 = nn.ReLU(inplace=True)
        self.conv4_3 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.relu4_3 = nn.ReLU(inplace=True)
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=2, ceil_mode=True)



        self.conv5_1 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.relu5_1 = nn.ReLU(inplace=True)
        self.conv5_2 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.relu5_2 = nn.ReLU(inplace=True)
        self.conv5_3 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1)
        self.relu5_3 = nn.ReLU(inplace=True)
        self.pool5 = nn.MaxPool2d(kernel_size=2, stride=2, ceil_mode=True)
```

```python
        self.fc6 = nn.Conv2d(512, 4096, kernel_size=7, stride=1, padding=0)
        self.relu_fc6 = nn.ReLU(inplace=True)
        self.dropout_fc6 = nn.Dropout2d(0.5)
        self.fc7 = nn.Conv2d(4096, 4096, kernel_size=1, stride=1, padding=0)
        self.relu_fc7 = nn.ReLU(inplace=True)
        self.dropout_fc7 = nn.Dropout2d(0.5)
        self.score = nn.Conv2d(4096, n_class, kernel_size=1, stride=1, padding=0)


        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, bi
        self.upscore2 = nn.ConvTranspose2d(n_class, n_class, kernel_size=4, stride=2, bi
        self.upscore3 = nn.ConvTranspose2d(n_class, n_class, kernel_size=16, stride=8, b


        # *****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)*****

        conv1 = self.relu1_2(self.conv1_2(self.relu1_1(self.conv1_1(x))))
        pool1 = self.pool1(conv1)
```

```
        conv2 = self.relu2_2(self.conv2_2(self.relu2_1(self.conv2_1(pool1))))
        pool2 = self.pool2(conv2)

        conv3 = self.relu3_3(self.conv3_3(self.relu3_2(self.conv3_2(self.relu3_1(self.co
        pool3 = self.pool3(conv3)

        conv4 = self.relu4_3(self.conv4_3(self.relu4_2(self.conv4_2(self.relu4_1(self.co
        pool4 = self.pool4(conv4)

        conv5 = self.relu5_3(self.conv5_3(self.relu5_2(self.conv5_2(self.relu5_1(self.co
        pool5 = self.pool5(conv5)


        fc6 = self.dropout_fc6(self.relu_fc6(self.fc6(pool5)))
        fc7 = self.dropout_fc7(self.relu_fc7(self.fc7(fc6)))
        score = self.score(fc7)


        upscore1 = self.upscore1(score)


        score_pool_4 = self.score_pool_4(pool4)
        score_pool_4c = crop(score_pool_4, upscore1)
        fuse1 = upscore1 + score_pool_4c


        upscore2 = self.upscore2(fuse1)


        score_pool_3 = self.score_pool_3(pool3)
        score_pool_3c = crop(score_pool_3, upscore2)
        fuse2 = upscore2 + score_pool_3c


        upscore3 = self.upscore3(fuse2)
        h = crop(upscore3, x, final=True)


        # *****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 [4]:
```
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
)
```

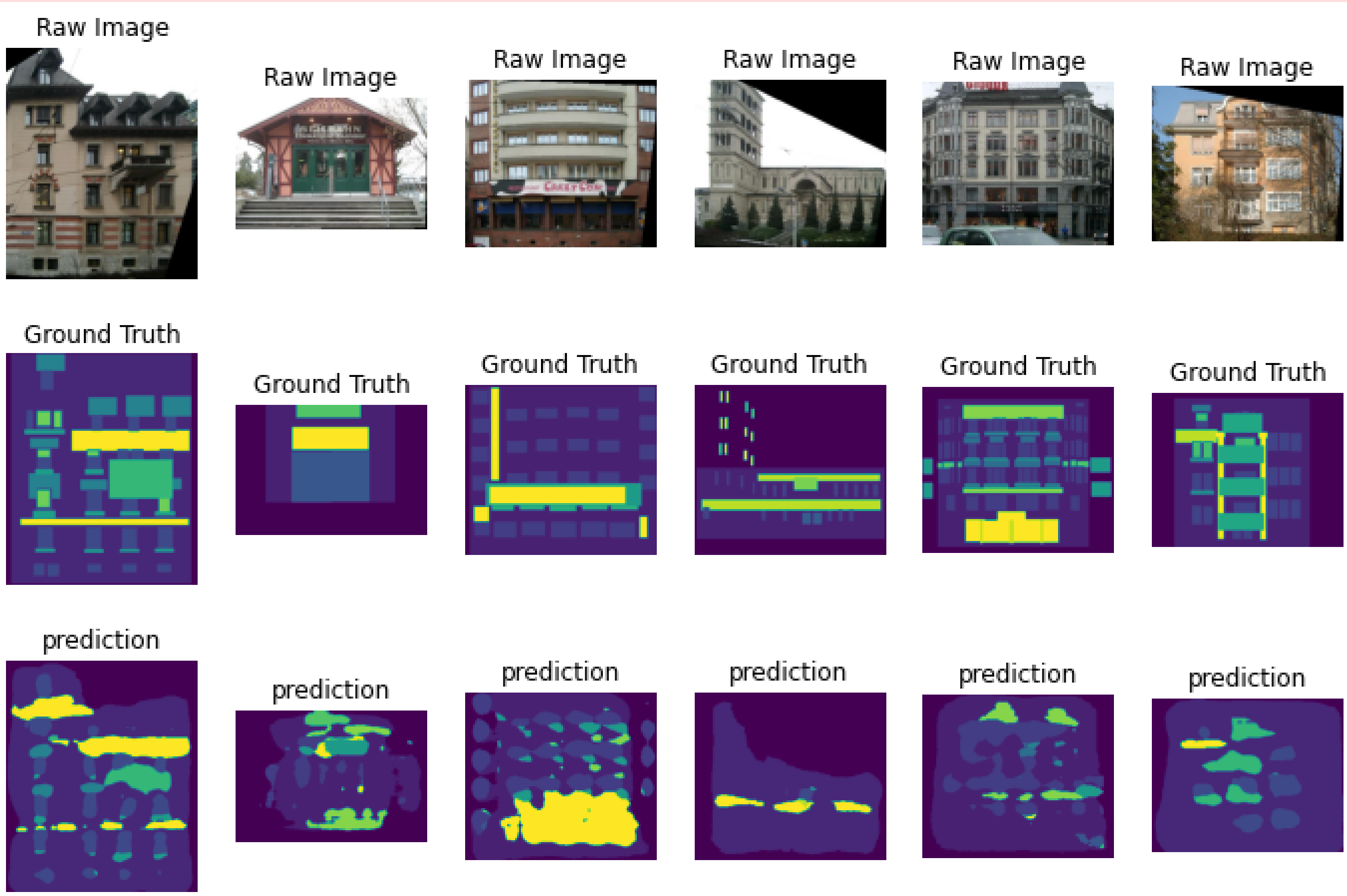
```
/opt/conda/lib/python3.9/site-packages/torchvision/models/_utils.py:208: UserWarning: Th
e parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, plea
se use 'weights' instead.
  warnings.warn(
/opt/conda/lib/python3.9/site-packages/torchvision/models/_utils.py:223: UserWarning: Ar
guments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and m
ay be removed in the future. The current behavior is equivalent to passing `weights=VGG1
6_Weights.IMAGENET1K_V1`. You can also use `weights=VGG16_Weights.DEFAULT` to get the mo
st up-to-date weights.
  warnings.warn(msg)
Init Model
Avg Acc: 0.2307, Mean IoU: 0.01922
Epochs: 0
Epoch Loss: 1.581, Avg Acc: 0.5378, Mean IoU: 0.1705
Epochs: 1
Epoch Loss: 1.312, Avg Acc: 0.5656, Mean IoU: 0.1701
Epochs: 2
Epoch Loss: 1.188, Avg Acc: 0.5847, Mean IoU: 0.2376
Epochs: 3
Epoch Loss: 1.112, Avg Acc: 0.6061, Mean IoU: 0.2622
Epochs: 4
Epoch Loss: 1.037, Avg Acc: 0.6064, Mean IoU: 0.2478
Epochs: 5
Epoch Loss: 0.9723, Avg Acc: 0.6287, Mean IoU: 0.2926
Epochs: 6
Epoch Loss: 0.9262, Avg Acc: 0.6279, Mean IoU: 0.2968
Epochs: 7
Epoch Loss: 0.8643, Avg Acc: 0.6412, Mean IoU: 0.3251
Epochs: 8
Epoch Loss: 0.8055, Avg Acc: 0.6489, Mean IoU: 0.3207
Epochs: 9
Epoch Loss: 0.748, Avg Acc: 0.6499, Mean IoU: 0.3201
Epochs: 10
```

```
Epoch Loss: 0.7311, Avg Acc: 0.652, Mean IoU: 0.3355
Epochs: 11
Epoch Loss: 0.6733, Avg Acc: 0.6478, Mean IoU: 0.3313
Epochs: 12
Epoch Loss: 0.6547, Avg Acc: 0.6605, Mean IoU: 0.3372
Epochs: 13
Epoch Loss: 0.6149, Avg Acc: 0.6733, Mean IoU: 0.3508
Epochs: 14
Epoch Loss: 0.5834, Avg Acc: 0.6665, Mean IoU: 0.3546
Epochs: 15
Epoch Loss: 0.5604, Avg Acc: 0.6661, Mean IoU: 0.3548
Epochs: 16
Epoch Loss: 0.5301, Avg Acc: 0.6769, Mean IoU: 0.3715
Epochs: 17
Epoch Loss: 0.5178, Avg Acc: 0.6785, Mean IoU: 0.3545
Epochs: 18
Epoch Loss: 0.4874, Avg Acc: 0.6789, Mean IoU: 0.367
Epochs: 19
Epoch Loss: 0.4724, Avg Acc: 0.6765, Mean IoU: 0.3566
Test Acc: 0.6769, Test Mean IoU: 0.3715
```

In [9]:

```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).
```



Part 3: Questions (29%):

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 pretrain weights perform a lot better than the FCN-32s from scratch both in accuracy and classifying pixels to correspoind class. The FCN-32s from scratch classify all the pixel to one class while the FCN-32s with pretrained weights can classify some pixel to the correspoding class. The pretrain weights help a lot on classifying pixel to different class because the pretrained model has already learned a set of useful features from a large and diverse dataset, and the FCN-32s model with pretrain weight can converge faster than the FCN-32s model from scratch.

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: The FCN-8s with pretrain weights perform better than the FCN-32s with pretrain weights both in accuracy and classifying pixels to correspoind class. The FCN-32s with pretrained weights can classify some pixel to the correspoding class while the FCN-8s with pretrain weight is more accuratly classify the pixel to its coresponding class. The FCN-8s with pretrain weights perform better because the skip connection strategy from the FCN-8s is able to bring high resolution details from early convolutional layers to the final layers which will improve the accuracy.

Survey (1%)

Question:

How many hours did you spend on this assignment?

Your Answer: 30 hours

```
In [ ]:
```