# Assignment 3: Pytorch Segmentation

For this assignment, we're goining to use Deep Learning for a new task: semantic segmentation , instead of classification we've been doing. We will also use some common techniques in Deep Learning like pretraining.

## 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 deference between semantic segmentation and classification is that for semantic segmentation, model output label for each pixel instead of a single label for the whole image

### Metrics

In semantic segmentations, we will average pixel-wise accuracy and IoU to benchmark semantic segmentation methods. Here we provide, the code for Evaluation

```
12 import numpy as np

def _hist(pred, gt, n_class):
#     mask = (label_true >= 0) & (label_true < n_class)
    hist = np.bincount(
        n_class * gt.astype(int) +
        pred, minlength=n_class ** 2
    ).reshape(n_class, n_class)
    return hist


def metrics(preds, gts, n_class):
    hist = np.zeros((n_class, n_class))
    for pred, gt in zip(preds, gts):
        hist += _hist(pred.flatten(), gt.flatten(), n_class)
    acc = np.diag(hist).sum() / hist.sum()
    iou = np.diag(hist) / (
        hist.sum(axis=1) + hist.sum(axis=0) - np.diag(hist)
    )
    mean_iou = np.nanmean(iou)
    return acc, mean_iou
```

## CMP Facade DB

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:

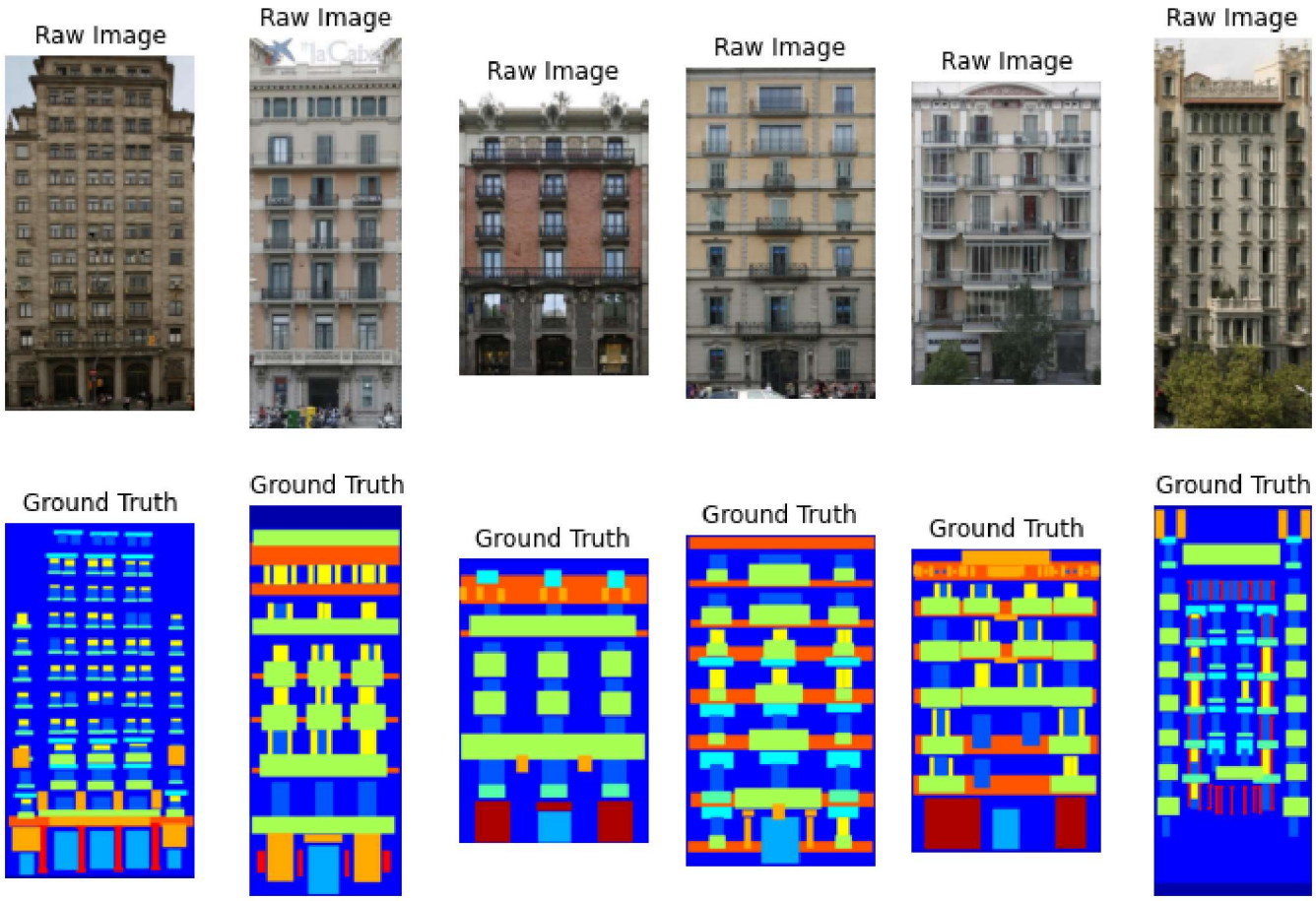
- facade
- molding
- cornice
- pillar
- window
- door
- sill
- blind
- balcony
- shop
- deco
- background

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/

## Visualize of the Dataset

```
13 import matplotlib.pyplot as plt
   import numpy as np
   # import PI

   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")
```




# Prepare

```
14 import torch
   import copy

   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)
```

```
   using device: cuda
```

# Build Dataset Class in Pytorch

```
15 import torch
   import PIL
   from torch.utils.data import Dataset
   import os
```

```
import os.path as osp
import torchvision.transforms as transforms
from PIL import Image


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

class CMP_Facade_DB(Dataset):
    def __init__(
        self,
        data_list
    ):
        self.data_list = data_list

    def __len__(self):
        return len(self.data_list)

    def __getitem__(self, i):
        # input and target images
        in_name = self.data_list[i]
        gt_name = self.data_list[i].replace('.jpg','.png')

        # process the images
        normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                         std=[0.229, 0.224, 0.225])
        transf = transforms.Compose([
            transforms.ToTensor(),
            normalize
        ])
        in_image = transf(
            Image.open(in_name).convert('RGB')
        )
        gt_im = Image.open(gt_name)

        gt_label = torch.LongTensor(
            np.frombuffer(gt_im.tobytes(), dtype=np.ubyte).reshape(
                in_image.shape[1:]
            )
        ) - 1

        return in_image, gt_label

    def revert_input(self, img, label):
        img = np.transpose(img.cpu().numpy(), (1, 2, 0))
        std_img = np.array([0.229, 0.224, 0.225]).reshape((1, 1, -1))
        mean_img = np.array([0.485, 0.456, 0.406]).reshape((1, 1, -1))
        img *= std_img
        img += mean_img
```

```
        label = label.cpu().numpy()
        return img, label + 1

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))
```

```
Training Set Size: 500
Validation Set Size: 30
Test Set Size: 76
```

# Fully Convolutional Networks for Semantic Segmentation

We've seen that CNNs are powerful models to get hiereachical visual features in Deep Learning. There we are going to explore the classical work: "Fully Convolutional Networks for Semantic Segmentation"(FCN).

Though we've already used CNN models for image classifications in the previous assignemtn, those models have one major drawback: Those model take input with fixed shape and output a single vector. However, in semantic segmentation, we want the model to be able to process image with arbitary shape and predict the label map with the same shape as the input image.

In FCN, the model utilize the Transpose Convolution layers, which we've already learned during the lecture, to make it happen. For the overal 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. If you need more reference, you can check the original paper: https://arxiv.org/pdf/1411.4038.pdf and some other materials online.

Besides of transpose Convolution, there are also some difference 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.

## Naive FCN: FCN-32s (30%)

In this section, we first try to implement naive variant of FCN without skip connection: FCN-32s. Here we use FCN-32s with VGG-16 architecture for feature encoding.

Compared with VGG-16, FCN-32s only replace the fully connecteed layers with 1x1 convolution and add a Transpose Convolution at the end to output dense prediction.

FC-32s architecture:

The following Conv use kernel size = 3, padding = 1, stride =1(except conv1_1. conv1_1 should use padding = 100) The Max Pool should use "ceil_mode = True"

- [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] -> [maxpool3(2,2)]
- [conv5_1(512,512)-relu] -> [conv5_2(512,512)-relu] ->[conv5_3(512,512)-relu] -> [maxpool3(2,2)]

The following Conv use kernel size = 7, stride = 1, padding = 0

- [fc6=conv(512, 4096, 7)-relu-dropout2d]

The following Conv use kernel size = 1, stride = 1, padding = 0

- [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)]

**Note: The output of the transpose convlution 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**

**It's expected that you model perform very poor in this section. This section is mainly for you to debug your code**

**Try to name the layers use the name provide above to ensure the next section works correctly, and use a new nn.RELU() for each activation**

**The model should achieve at least 40% Mean IoU for the pretrained version.**

```
16 import torch.nn as nn

def get_upsampling_weight(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()

class FCN32s(nn.Module):
    def __init__(self, n_class=12):
        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=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((2,2), ceil_mode=True)

        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((2,2), ceil_mode=True)

        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((2,2), ceil_mode=True)

        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((2,2), ceil_mode=True)

        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((2,2), ceil_mode=True)

        self.fc6 = nn.Conv2d(512,4096, kernel_size=7, stride=1, padding=0)
        self.relu6 = nn.ReLU()
        self.dropout6 = nn.Dropout2d()

        self.fc7 = nn.Conv2d(4096, 4096, kernel_size=1, stride=1, padding=0)
        self.relu7 = nn.ReLU()
        self.dropout7 = nn.Dropout2d()
```

```
        self.out = nn.Conv2d(4096, n_class, kernel_size=1, stride=1,padding=0)
        self.transpose1 = nn.ConvTranspose2d(n_class, 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 _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 = 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, W = x.shape[-2], x.shape[-1]
        # print('H,W',H,W)
        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.pool3(x)

        x = self.conv4_1(x)
        x = self.relu4_1(x)
        x = self.conv4_2(x)
        x = self.relu4_2(x)
        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.pool5(x)

        x = self.fc6(x)
        x = self.relu6(x)
        x = self.dropout6(x)
```

```
        x = self.fc7(x)
        x = self.relu7(x)
        x = self.dropout7(x)
        x = self.out(x)
        x = self.transpose1(x)
        h = x[...,19:H+19,19:W+19]
        # print('h',h.shape)

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

17
```
# You can change it if you want
lr = 1e-4
weight_decay = 2e-5
```

18
```
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=True
    )


def Evaluate(
    val_loader,
    model,
    current_best,
    n_class=12
):
    val_loss = 0
    visualizations = []
    preds, gts = [], []

    model.eval()
    for batch_idx, (data, target) in enumerate(val_loader):
        data, target = data.to(device), target.to(device)
        with torch.no_grad():
            score = model(data)

        pred = score.max(1)[1].cpu().numpy()
        gt = target.cpu().numpy()
        preds.append(pred)
        gts.append(gt)

    avg_acc, mean_iou = metrics(
        preds, gts, n_class)

    if mean_iou > current_best["IoU"]:
        current_best["IoU"] = mean_iou
        current_best["model"] = copy.deepcopy(model)

    return avg_acc, mean_iou, current_best

def Train(
    model,
    loss_func,
    optim,
    scheduler,
    epochs,
    train_loader,
    val_lodaer,
    test_loader,
    display_interval = 100
):

    current_best = {
        "IoU": 0,
        "model": model
    }
    avg_acc, mean_iou, current_best = Evaluate(
        val_loader,
        model,
        current_best
    )

    print("Init Model")
    print("Avg Acc: {:.4}, Mean IoU: {:.4}".format(
        avg_acc, mean_iou
    ))
    for i in range(epochs):
        print("Epochs: {}".format(i))
        total_loss = 0
        model.train()
```

```
        for batch_idx, (data, target) in enumerate(train_loader):
            data, target = data.to("cuda:0"), target.to("cuda:0")
            optim.zero_grad()

            score = model(data)
            loss = loss_func(score, target.squeeze(1))
            loss_data = loss.item()
            if np.isnan(loss_data):
                raise ValueError('loss is nan while training')
            loss.backward()
            optim.step()
            total_loss += loss.item()
            if batch_idx % display_interval == 0 and batch_idx != 0:
                print("{} / {}, Current Avg Loss:{:.4}".format(
                    batch_idx, len(train_loader), total_loss / (batch_idx + 1)
                ))


        total_loss /= len(train_loader)
        model.eval()
        avg_acc, mean_iou, current_best = Evaluate(
            val_loader,
            model,
            current_best
        )
        scheduler.step(total_loss)
        print("Avg Loss: {:.4}, Avg Acc: {:.4}, Mean IoU: {:.4}".format(
            total_loss, avg_acc, mean_iou
        ))

    test_acc, test_iou, current_best = Evaluate(
        val_loader,
        current_best["model"],
        current_best
    )
    print("Test Acc: {:.4}, Test Mean IoU: {:.4}".format(
        test_acc, test_iou
    ))
    return current_best["model"]



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

optim = torch.optim.Adam(
    model.parameters(),
    lr=lr,
    weight_decay=weight_decay,
)
from torch.optim.lr_scheduler import ReduceLROnPlateau
scheduler = ReduceLROnPlateau(
    optim, 'min', patience=3,
    min_lr=1e-10, verbose=True
)

# Choose the right loss function in torch.nn
loss_func = nn.CrossEntropyLoss()
best_model = Train(
    model,
    loss_func,
    optim,
    scheduler,
    5,
    train_loader,
```

```
    val_loader,
    test_loader
)
```

```
<ipython-input-4-4da76993cd6a>:56: UserWarning: The given NumPy array is not writeable, and PyTorch does
  gt_label = torch.LongTensor(

Init Model
Avg Acc: 0.02884, Mean IoU: 0.002403
Epochs: 0
100 / 500, Current Avg Loss:1.996
200 / 500, Current Avg Loss:1.97
300 / 500, Current Avg Loss:1.943
400 / 500, Current Avg Loss:1.917
Avg Loss: 1.892, Avg Acc: 0.3675, Mean IoU: 0.04041
Epochs: 1
100 / 500, Current Avg Loss:1.779
200 / 500, Current Avg Loss:1.772
300 / 500, Current Avg Loss:1.763
400 / 500, Current Avg Loss:1.763
Avg Loss: 1.757, Avg Acc: 0.4375, Mean IoU: 0.06889
Epochs: 2
100 / 500, Current Avg Loss:1.752
200 / 500, Current Avg Loss:1.742
300 / 500, Current Avg Loss:1.746
400 / 500, Current Avg Loss:1.734
Avg Loss: 1.723, Avg Acc: 0.4471, Mean IoU: 0.07259
Epochs: 3
100 / 500, Current Avg Loss:1.671
200 / 500, Current Avg Loss:1.68
300 / 500, Current Avg Loss:1.682
400 / 500, Current Avg Loss:1.683
Avg Loss: 1.686, Avg Acc: 0.4428, Mean IoU: 0.07453
Epochs: 4
100 / 500, Current Avg Loss:1.668
200 / 500, Current Avg Loss:1.66
300 / 500, Current Avg Loss:1.657
400 / 500, Current Avg Loss:1.634
Avg Loss: 1.616, Avg Acc: 0.5105, Mean IoU: 0.1307
Test Acc: 0.5105, Test Mean IoU: 0.1307
```

## Visualize Output

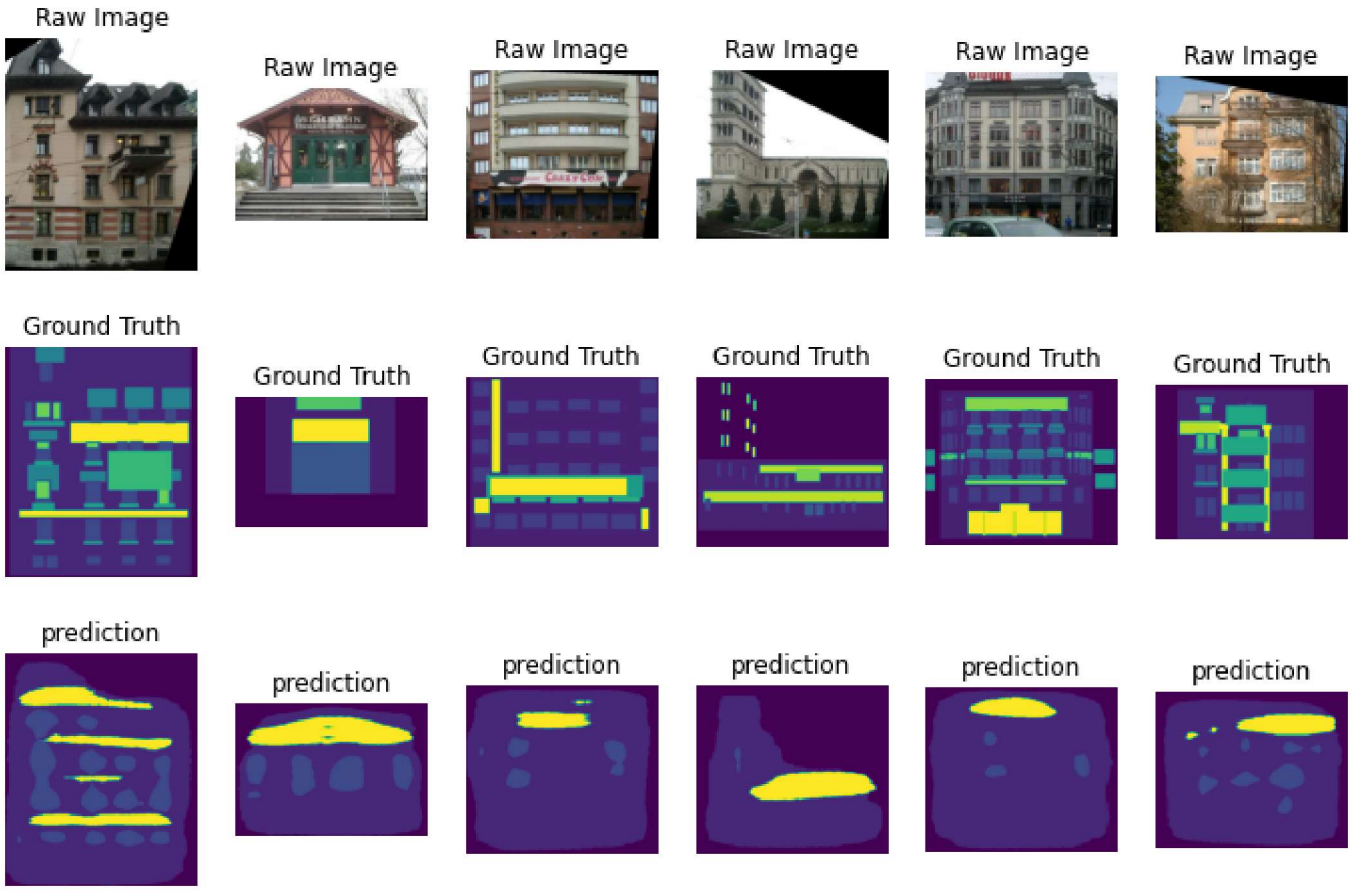
In this section, we visualize several model outputs to see how our model actually perform.

```
def visualize(model, test_loader):
    idxs = [1, 2, 5, 6, 7, 8]
    fig, axes = plt.subplots(nrows=3, ncols=6, figsize=(12, 8))
    model.eval()
    for i, idx in enumerate(idxs):
        img, label = test_loader.dataset[idx]

        pred = model(img.unsqueeze(0).to(device))
        pred = (pred.max(1)[1] + 1).squeeze(0).cpu().numpy()

        img, label = test_loader.dataset.revert_input(img, label)

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

        axes[1][i].imshow(label)
        axes[1][i].axis('off')
```

```
        axes[1][i].set_title("Ground Truth")

        axes[2][i].imshow(pred)
        axes[2][i].axis('off')
        axes[2][i].set_title("prediction")
```

11
```
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 intege
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
```




## Utilize the pretrain features

In the previous section, we use the random initalized weights to train FCN-32S from scrath. We can see that it perform poorly. In this section, we utilize the feature from pretrained model(In our case, we use VGG-16) to help us get a better result.

14
```
import torchvision
vgg16 = torchvision.models.vgg16(pretrained=True)

model = FCN32s(n_class=12)
model.copy_params_from_vgg16(vgg16)
```

```
model.to(device)
```

```
optim = torch.optim.Adam(
    model.parameters(),
    lr=lr,
    weight_decay=weight_decay,
)
from torch.optim.lr_scheduler import ReduceLROnPlateau
scheduler = ReduceLROnPlateau(
    optim, 'min', patience=3,
    min_lr=1e-10, verbose=True
)

loss_func = nn.CrossEntropyLoss()
best_model_pretrain = Train(
    model,
    loss_func,
    optim,
    scheduler,
    25,
    train_loader,
    val_loader,
    test_loader
)
```

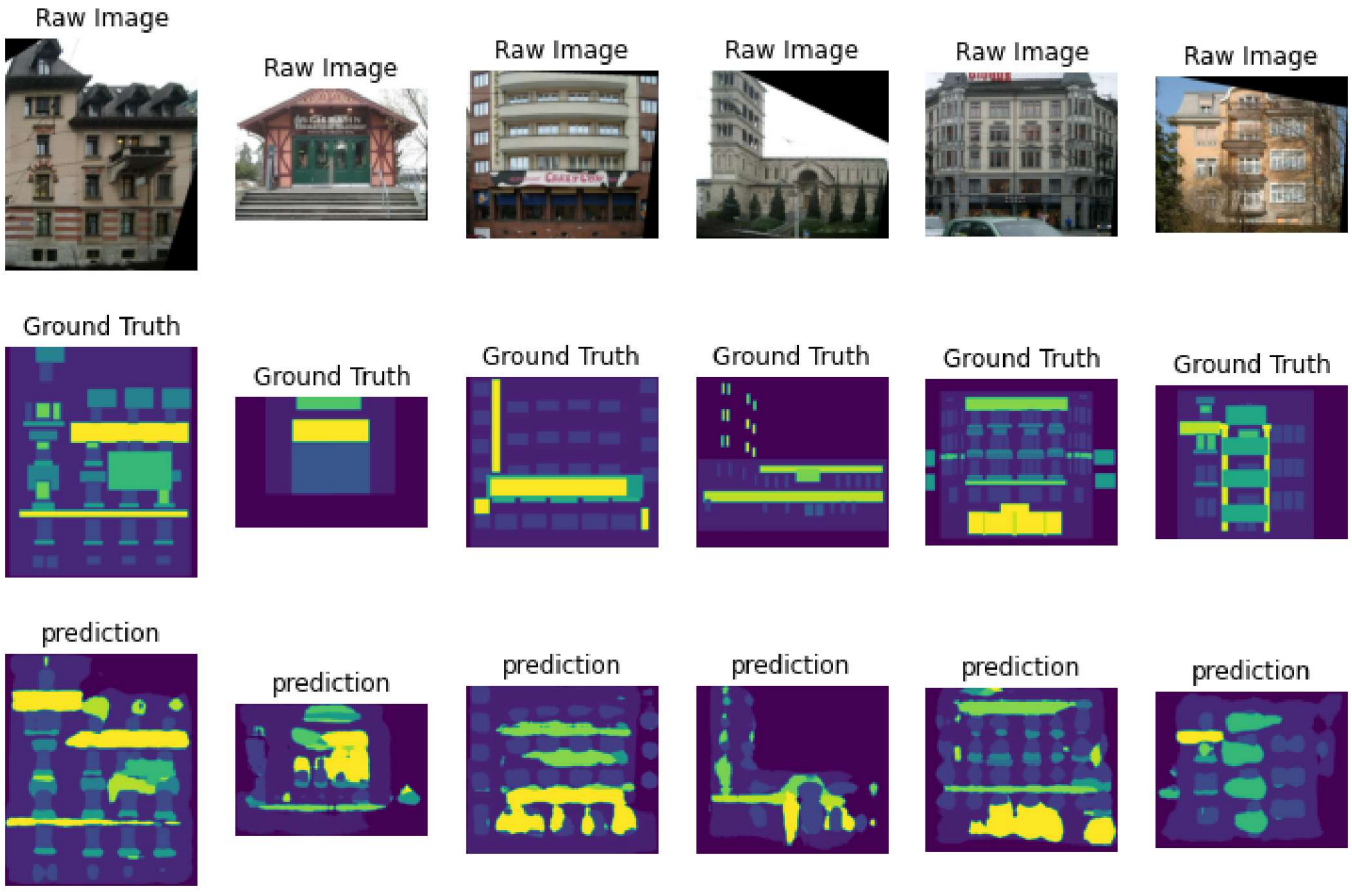
```
<ipython-input-6-4da76993cd6a>:56: UserWarning: The given NumPy array is not writeable, and PyTorch does
  gt_label = torch.LongTensor(

Init Model
Avg Acc: 0.004432, Mean IoU: 0.001011
Epochs: 0
100 / 500, Current Avg Loss:1.911
200 / 500, Current Avg Loss:1.779
300 / 500, Current Avg Loss:1.709
400 / 500, Current Avg Loss:1.675
Avg Loss: 1.621, Avg Acc: 0.5524, Mean IoU: 0.1842
Epochs: 1
100 / 500, Current Avg Loss:1.358
200 / 500, Current Avg Loss:1.334
300 / 500, Current Avg Loss:1.315
400 / 500, Current Avg Loss:1.311
Avg Loss: 1.299, Avg Acc: 0.5582, Mean IoU: 0.1938
Epochs: 2
100 / 500, Current Avg Loss:1.239
200 / 500, Current Avg Loss:1.226
300 / 500, Current Avg Loss:1.225
400 / 500, Current Avg Loss:1.191
Avg Loss: 1.184, Avg Acc: 0.6043, Mean IoU: 0.2566
Epochs: 3
100 / 500, Current Avg Loss:1.061
200 / 500, Current Avg Loss:1.068
300 / 500, Current Avg Loss:1.08
400 / 500, Current Avg Loss:1.079
Avg Loss: 1.068, Avg Acc: 0.598, Mean IoU: 0.2826
Epochs: 4
100 / 500, Current Avg Loss:0.9962
200 / 500, Current Avg Loss:0.9851
300 / 500, Current Avg Loss:1.001
400 / 500, Current Avg Loss:0.9915
Avg Loss: 0.9846, Avg Acc: 0.6279, Mean IoU: 0.3288
Epochs: 5
100 / 500, Current Avg Loss:0.9425
```

```
200 / 500, Current Avg Loss:0.922
300 / 500, Current Avg Loss:0.9163
400 / 500, Current Avg Loss:0.9222
Avg Loss: 0.914, Avg Acc: 0.6493, Mean IoU: 0.3399
Epochs: 6
100 / 500, Current Avg Loss:0.8561
200 / 500, Current Avg Loss:0.867
300 / 500, Current Avg Loss:0.8526
400 / 500, Current Avg Loss:0.8583
Avg Loss: 0.8556, Avg Acc: 0.6664, Mean IoU: 0.3549
Epochs: 7
100 / 500, Current Avg Loss:0.7939
200 / 500, Current Avg Loss:0.7879
300 / 500, Current Avg Loss:0.7983
400 / 500, Current Avg Loss:0.7914
Avg Loss: 0.7949, Avg Acc: 0.6752, Mean IoU: 0.3691
Epochs: 8
100 / 500, Current Avg Loss:0.6917
200 / 500, Current Avg Loss:0.7315
300 / 500, Current Avg Loss:0.7394
400 / 500, Current Avg Loss:0.7508
Avg Loss: 0.7503, Avg Acc: 0.6609, Mean IoU: 0.3541
Epochs: 9
100 / 500, Current Avg Loss:0.6725
200 / 500, Current Avg Loss:0.6808
300 / 500, Current Avg Loss:0.6778
400 / 500, Current Avg Loss:0.6899
Avg Loss: 0.688, Avg Acc: 0.6873, Mean IoU: 0.4101
Epochs: 10
100 / 500, Current Avg Loss:0.6101
200 / 500, Current Avg Loss:0.6018
300 / 500, Current Avg Loss:0.6226
400 / 500, Current Avg Loss:0.634
Avg Loss: 0.6308, Avg Acc: 0.6921, Mean IoU: 0.4037
Epochs: 11
100 / 500, Current Avg Loss:0.5687
200 / 500, Current Avg Loss:0.5712
300 / 500, Current Avg Loss:0.5859
400 / 500, Current Avg Loss:0.6006
Avg Loss: 0.6013, Avg Acc: 0.6699, Mean IoU: 0.3975
Epochs: 12
100 / 500, Current Avg Loss:0.5955
200 / 500, Current Avg Loss:0.5875
300 / 500, Current Avg Loss:0.5747
400 / 500, Current Avg Loss:0.5738
Avg Loss: 0.5665, Avg Acc: 0.6867, Mean IoU: 0.3991
Epochs: 13
100 / 500, Current Avg Loss:0.5069
200 / 500, Current Avg Loss:0.5112
300 / 500, Current Avg Loss:0.503
400 / 500, Current Avg Loss:0.4987
Avg Loss: 0.5022, Avg Acc: 0.6993, Mean IoU: 0.4203
Epochs: 14
100 / 500, Current Avg Loss:0.4451
200 / 500, Current Avg Loss:0.4607
300 / 500, Current Avg Loss:0.4597
400 / 500, Current Avg Loss:0.4633
Avg Loss: 0.466, Avg Acc: 0.6947, Mean IoU: 0.409
Epochs: 15
100 / 500, Current Avg Loss:0.4655
200 / 500, Current Avg Loss:0.4495
300 / 500, Current Avg Loss:0.4464
400 / 500, Current Avg Loss:0.4384
Avg Loss: 0.4376, Avg Acc: 0.7074, Mean IoU: 0.4302
Epochs: 16
```

```
100 / 500, Current Avg Loss:0.4243
200 / 500, Current Avg Loss:0.4107
300 / 500, Current Avg Loss:0.4063
400 / 500, Current Avg Loss:0.4028
Avg Loss: 0.4078, Avg Acc: 0.7007, Mean IoU: 0.4225
Epochs: 17
100 / 500, Current Avg Loss:0.3956
200 / 500, Current Avg Loss:0.3867
300 / 500, Current Avg Loss:0.385
400 / 500, Current Avg Loss:0.3971
Avg Loss: 0.3977, Avg Acc: 0.7085, Mean IoU: 0.4161
Epochs: 18
100 / 500, Current Avg Loss:0.4289
200 / 500, Current Avg Loss:0.4035
300 / 500, Current Avg Loss:0.3967
400 / 500, Current Avg Loss:0.3878
Avg Loss: 0.3867, Avg Acc: 0.7067, Mean IoU: 0.4297
Epochs: 19
100 / 500, Current Avg Loss:0.3774
200 / 500, Current Avg Loss:0.3639
300 / 500, Current Avg Loss:0.3628
400 / 500, Current Avg Loss:0.3577
Avg Loss: 0.3552, Avg Acc: 0.7007, Mean IoU: 0.4274
Epochs: 20
100 / 500, Current Avg Loss:0.3367
200 / 500, Current Avg Loss:0.3273
300 / 500, Current Avg Loss:0.3301
400 / 500, Current Avg Loss:0.3307
Avg Loss: 0.3314, Avg Acc: 0.7037, Mean IoU: 0.4381
Epochs: 21
100 / 500, Current Avg Loss:0.3363
200 / 500, Current Avg Loss:0.3302
300 / 500, Current Avg Loss:0.3285
400 / 500, Current Avg Loss:0.3323
Avg Loss: 0.3318, Avg Acc: 0.7164, Mean IoU: 0.4216
Epochs: 22
100 / 500, Current Avg Loss:0.3093
200 / 500, Current Avg Loss:0.3105
300 / 500, Current Avg Loss:0.3133
400 / 500, Current Avg Loss:0.3212
Avg Loss: 0.3225, Avg Acc: 0.7053, Mean IoU: 0.4291
Epochs: 23
100 / 500, Current Avg Loss:0.307
200 / 500, Current Avg Loss:0.3114
300 / 500, Current Avg Loss:0.3042
400 / 500, Current Avg Loss:0.3035
Avg Loss: 0.301, Avg Acc: 0.7104, Mean IoU: 0.4397
Epochs: 24
100 / 500, Current Avg Loss:0.3074
200 / 500, Current Avg Loss:0.3009
300 / 500, Current Avg Loss:0.292
400 / 500, Current Avg Loss:0.2895
Avg Loss: 0.29, Avg Acc: 0.7131, Mean IoU: 0.4358
Test Acc: 0.7104, Test Mean IoU: 0.4397
```

15
```
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 intege
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
```




## Skip Connection: FCN-8s(40%)

Though we've get a prety good result using FCN-32s with VGG-16 pretrain. We can actully do better with another technique introduced in FCN paper: Skip Connection.

With skip connection, we are supposed to get a better performance especially for some details.

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 conv1_1. conv1_1 should use padding = 100) The Max Pool should use "ceil_mode = True"

**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] -> [maxpool3(2,2)]
- [conv5_1(512,512)-relu] -> [conv5_2(512,512)-relu] ->[conv5_3(512,512)-relu] -> [maxpool3(2,2)]

The following Conv use kernel size = 7, stride = 1, padding = 0

- [fc6=conv(512, 4096, 7)-relu-dropout2d]

The following Conv use kernel size = 1, stride = 1, padding = 0

- [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: kernel size = 4, stride = 2, bias = False

- [upscore1 = transpose_conv(n_class, n_class)]

The transpose convolution: kernel size = 4, stride = 2, bias = False

- [upscore2 = transpose_conv(n_class, n_class)]

The transpose convolution: kernel 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]

**The model should achieve at least 40% Mean IoU**

```
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((2,2), ceil_mode=True)

        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((2,2), ceil_mode=True)

        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((2,2), ceil_mode=True)

        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((2,2), ceil_mode=True)

        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((2,2), ceil_mode=True)

        self.fc6 = nn.Conv2d(512,4096, kernel_size=7, stride=1, padding=0)
        self.relu6 = nn.ReLU()
        self.dropout6 = nn.Dropout2d()

        self.fc7 = nn.Conv2d(4096, 4096, kernel_size=1, stride=1, padding=0)
        self.relu7 = nn.ReLU()
        self.dropout7 = nn.Dropout2d()

        self.transpose1 = nn.ConvTranspose2d(n_class, n_class, 64, stride=32, bias=False)

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


        self._initialize_weights()

    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 = 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, W = x.shape[-2], x.shape[-1]
        # print('H,W',H,W)
        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.pool3(x)
        score_pool_3 = self.score_pool_3(x)


        x = self.conv4_1(x)
        x = self.relu4_1(x)
        x = self.conv4_2(x)
        x = self.relu4_2(x)
        x = self.pool4(x)
        score_pool_4 = self.score_pool_4(x)

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


        x = self.fc6(x)
        x = self.relu6(x)
        x = self.dropout6(x)

        x = self.fc7(x)
        x = self.relu7(x)
        x = self.dropout7(x)
        x = self.score(x)

        upscore1 = self.upscore1(x)
        upscore1_h, upscore1_w = upscore1.shape[-2], upscore1.shape[-1]
        upscore2 = self.upscore2(score_pool_4[...,5:5+upscore1_h, 5:5+upscore1_w] + upscore1)
        upscore2_h, upscore2_w = upscore2.shape[-2], upscore2.shape[-1]
        upscore3 = self.upscore3(score_pool_3[...,9:9+upscore2_h, 9:9+upscore2_w] +
                                 upscore2)[...,31:31+H,31:31+W]

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


        return upscore3

    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()))
```

```
*   import torchvision
    vgg16 = torchvision.models.vgg16(pretrained=True)

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

    optim = torch.optim.Adam(
        model.parameters(),
        lr=lr,
        weight_decay=weight_decay,
    #     momentum=momentum,
    )

    from torch.optim.lr_scheduler import ReduceLROnPlateau
    scheduler = ReduceLROnPlateau(
        optim, 'min', patience=3,
        min_lr=1e-10, verbose=True
    )
    loss_func = nn.CrossEntropyLoss()
    best_model_fcn8s = Train(
        model,
        loss_func,
        optim,
        scheduler,
        25,
        train_loader,
        val_loader,
        test_loader
    )
```

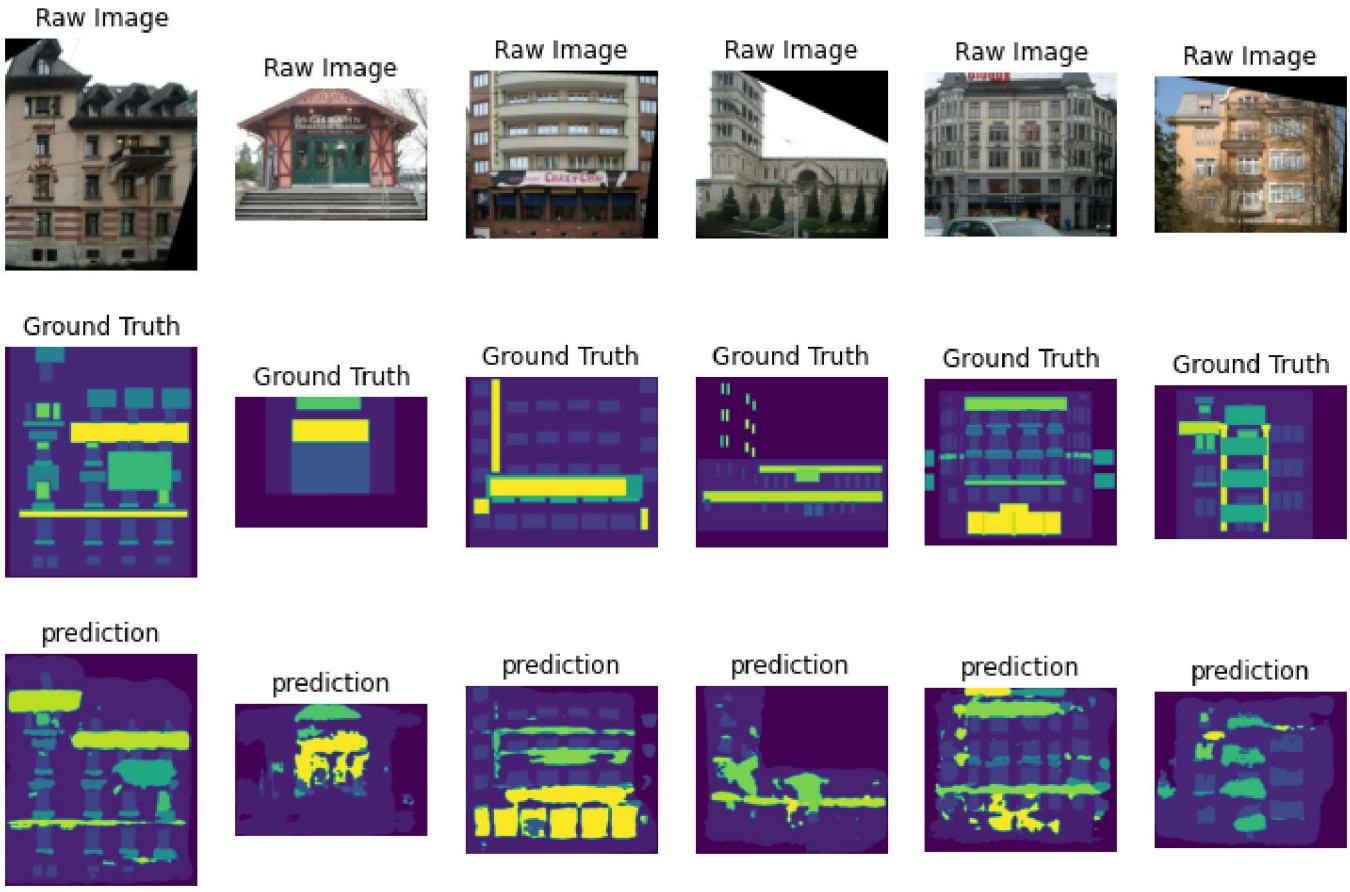
```
    <ipython-input-4-4da76993cd6a>:56: UserWarning: The given NumPy array is not writeable, and PyTorch does
      gt_label = torch.LongTensor(

    Init Model
    Avg Acc: 0.06052, Mean IoU: 0.01811
    Epochs: 0
    100 / 500, Current Avg Loss:1.741
    200 / 500, Current Avg Loss:1.576
    300 / 500, Current Avg Loss:1.478
    400 / 500, Current Avg Loss:1.403
    Avg Loss: 1.361, Avg Acc: 0.5749, Mean IoU: 0.2668
    Epochs: 1
```

```
100 / 500, Current Avg Loss:1.138
200 / 500, Current Avg Loss:1.131
300 / 500, Current Avg Loss:1.107
400 / 500, Current Avg Loss:1.109
Avg Loss: 1.099, Avg Acc: 0.6194, Mean IoU: 0.2866
Epochs: 2
100 / 500, Current Avg Loss:1.028
200 / 500, Current Avg Loss:1.029
300 / 500, Current Avg Loss:1.005
400 / 500, Current Avg Loss:1.007
Avg Loss: 0.9933, Avg Acc: 0.6546, Mean IoU: 0.3441
Epochs: 3
100 / 500, Current Avg Loss:0.9098
200 / 500, Current Avg Loss:0.9341
300 / 500, Current Avg Loss:0.9185
400 / 500, Current Avg Loss:0.9218
Avg Loss: 0.9175, Avg Acc: 0.6549, Mean IoU: 0.3671
Epochs: 4
100 / 500, Current Avg Loss:0.8508
200 / 500, Current Avg Loss:0.8366
300 / 500, Current Avg Loss:0.8561
400 / 500, Current Avg Loss:0.8537
Avg Loss: 0.8469, Avg Acc: 0.66, Mean IoU: 0.376
Epochs: 5
100 / 500, Current Avg Loss:0.7703
200 / 500, Current Avg Loss:0.7786
300 / 500, Current Avg Loss:0.7908
400 / 500, Current Avg Loss:0.7876
Avg Loss: 0.7884, Avg Acc: 0.6807, Mean IoU: 0.4011
Epochs: 6
100 / 500, Current Avg Loss:0.7235
200 / 500, Current Avg Loss:0.7412
300 / 500, Current Avg Loss:0.7312
400 / 500, Current Avg Loss:0.7409
Avg Loss: 0.7435, Avg Acc: 0.6806, Mean IoU: 0.4135
Epochs: 7
100 / 500, Current Avg Loss:0.6738
200 / 500, Current Avg Loss:0.6978
300 / 500, Current Avg Loss:0.7001
400 / 500, Current Avg Loss:0.7062
Avg Loss: 0.7043, Avg Acc: 0.6857, Mean IoU: 0.4062
Epochs: 8
100 / 500, Current Avg Loss:0.6642
200 / 500, Current Avg Loss:0.6491
300 / 500, Current Avg Loss:0.646
400 / 500, Current Avg Loss:0.6425
Avg Loss: 0.646, Avg Acc: 0.6849, Mean IoU: 0.4113
Epochs: 9
100 / 500, Current Avg Loss:0.5545
200 / 500, Current Avg Loss:0.5787
300 / 500, Current Avg Loss:0.5963
400 / 500, Current Avg Loss:0.6036
Avg Loss: 0.6069, Avg Acc: 0.6942, Mean IoU: 0.4142
Epochs: 10
100 / 500, Current Avg Loss:0.5534
200 / 500, Current Avg Loss:0.5741
300 / 500, Current Avg Loss:0.5634
400 / 500, Current Avg Loss:0.5632
Avg Loss: 0.5624, Avg Acc: 0.7109, Mean IoU: 0.4601
Epochs: 11
100 / 500, Current Avg Loss:0.51
200 / 500, Current Avg Loss:0.5014
300 / 500, Current Avg Loss:0.5068
400 / 500, Current Avg Loss:0.5161
Avg Loss: 0.5156, Avg Acc: 0.7152, Mean IoU: 0.4574
```

```
Epochs: 12
100 / 500, Current Avg Loss:0.452
200 / 500, Current Avg Loss:0.4645
300 / 500, Current Avg Loss:0.4644
400 / 500, Current Avg Loss:0.4661
Avg Loss: 0.4726, Avg Acc: 0.7184, Mean IoU: 0.4602
Epochs: 13
100 / 500, Current Avg Loss:0.4132
200 / 500, Current Avg Loss:0.4209
300 / 500, Current Avg Loss:0.4463
400 / 500, Current Avg Loss:0.4448
Avg Loss: 0.4475, Avg Acc: 0.7135, Mean IoU: 0.4692
Epochs: 14
100 / 500, Current Avg Loss:0.4146
200 / 500, Current Avg Loss:0.4234
300 / 500, Current Avg Loss:0.4238
400 / 500, Current Avg Loss:0.4199
Avg Loss: 0.4199, Avg Acc: 0.7237, Mean IoU: 0.4723
Epochs: 15
100 / 500, Current Avg Loss:0.3691
200 / 500, Current Avg Loss:0.3836
300 / 500, Current Avg Loss:0.3973
400 / 500, Current Avg Loss:0.387
Avg Loss: 0.3847, Avg Acc: 0.7283, Mean IoU: 0.4758
Epochs: 16
100 / 500, Current Avg Loss:0.3464
200 / 500, Current Avg Loss:0.3499
300 / 500, Current Avg Loss:0.35
400 / 500, Current Avg Loss:0.351
Avg Loss: 0.3547, Avg Acc: 0.7233, Mean IoU: 0.4784
Epochs: 17
100 / 500, Current Avg Loss:0.3347
200 / 500, Current Avg Loss:0.3332
300 / 500, Current Avg Loss:0.3406
400 / 500, Current Avg Loss:0.3361
Avg Loss: 0.3389, Avg Acc: 0.7228, Mean IoU: 0.4844
Epochs: 18
100 / 500, Current Avg Loss:0.3189
200 / 500, Current Avg Loss:0.3141
300 / 500, Current Avg Loss:0.3173
400 / 500, Current Avg Loss:0.3194
Avg Loss: 0.3191, Avg Acc: 0.7261, Mean IoU: 0.4928
Epochs: 19
100 / 500, Current Avg Loss:0.3148
200 / 500, Current Avg Loss:0.3084
```

22
```
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 intege
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
```




## Inline Questions(30%):

**Inline Question 1: Why using pretrained model to initialized our model(FCN-32s) helps a lot? Please give at least two specific reasons**

### Your Answer:

1. The main reason is that training from scratch is not feasible according to the experiment. From the first section in this assignment and the experiment by authors(Long et al.), we can see the mean IU is increasing very slow. This shows training from scratch is not working.

2. The underlying reason is that imagenet pretrain initialization helps initialization, because there is a lot of similarity between image classification task and our semantic segmentation task. The similarity is that both tasks are for classification. The only difference is that the former is for image level classification, while the latter is for pixel level segmentation. The former only need the global information, while the latter also need the local information. Overall, these two tasks are very similar. Thus, imagenet classification weights can easily transfer to the semantic segmentaion task.

3. The similarity in weights lies in that, for the shallow layers, the two tasks share similar embeddings. Thus, only the weights in the deep layers would go through large change. That

is why the pre-trained network trains faster.

**Inline Question 2: Compare the performance and visualization of FCN-32s and FCN-8s. Please state the differnece, and provide some explanation. You can visualize more images than we provide, if it's necessary for you to see the difference.**

## Your Answer:

The difference is that FCN-8s has higher mean IU and average accuracy than FCN-32s. The visualization shows that FCN-8s retains more details.

The reason for such difference is that, FCN-8s has coarse-to-fine 'skip' architecture. If we view the network as an encoder-decoder architecture, the 'skip' connection means we have a multi-resolution decoder. (The deep layer decoder takes the output of the previous layer decoder.) Compared to the plain network, the multi-resolution decoder can obtain both the local and global information. It gets feature from different layers from the encoder, which are responsible for local and global information respectively. The shallow layer retrieves the local information, while the deep layer gets the global information. Contrastly, the FCN-32s only has global information from the deepest encoded feature map. This is why its result is vague.