# DSDE-Homework_4_Semantic_segmentation_Camseq_deeplabv3_DataInGD

ทำ semantic segmentation เพื่อหา shape ของ object ด้วย DeepLabV3 โดยใช้ PyTorch
Input data ที่ใช้ train เป็น CamSeq dataset
ทำ classification 32 classes

Setup

```
! pip install torchinfo
! nvidia-smi
```

unzip dataset Camseq

```
# unzip file (-q : quiet mode)
! unzip -FF Camseq_2007.zip
```

import libraries

```
# Pytorch framework
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
from torchinfo import summary as summary_info
import torch.optim as optim
from torch.optim import lr_scheduler
import torchvision
from torch.utils.data import Dataset, DataLoader, random_split, Subset
#utils
import os
import numpy as np
from collections import OrderedDict
from itertools import islice
#Transformation
import albumentations as A
from albumentations.pytorch import ToTensorV2
from collections import OrderedDict
#Visualization
import matplotlib.pyplot as plt
```

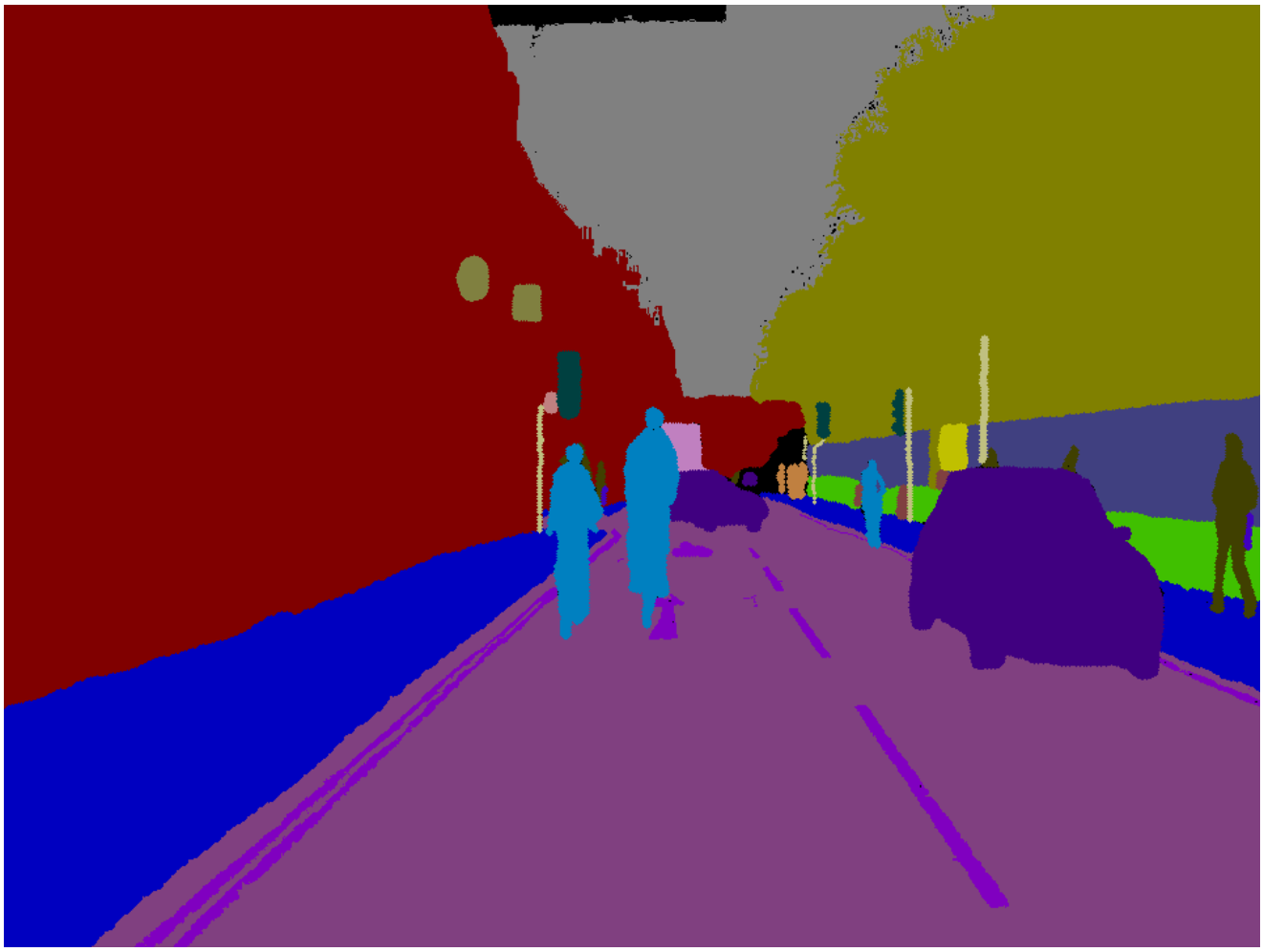
```
import cv2
from PIL import Image
#Progress bar
from tqdm.notebook import tqdm
%matplotlib inline
torch.__version__
```

ตัวอย่าง input image

```
img = Image.open('./Camseq_2007/0016E5_07961.png')
img
```


ตัวที่มี segmentation จะมีชื่อไฟล์ลงท้ายด้วย _L

```
img = Image.open('./Camseq_2007/0016E5_07961_L.png')
```



load สีที่ใช้ label

```
# load colormap from label_colors.txt
colormap = OrderedDict()
with open("./Camseq_2007/label_colors.txt",'r') as f:
        for line in f:
                r,g,b,cls = line.split()
                colormap[cls] = [int(e) for e in [r,g,b]]
                list(islice(colormap.items(),8))
```

```
'Animal': [64, 128, 64],
'Archway': [192, 0, 128],
'Bicyclist': [0, 128, 192],
'Bridge': [0, 128, 64],
'Building': [128, 0, 0],
'Car': [64, 0, 128],
'CartLuggagePram': [64, 0, 192],
'Child': [192, 128, 64],
```

สร้าง DataSet Class

```
class CamSeqDataset(Dataset):

    def __init__(self,
                 img_dir,
                 colormap=colormap,
                 transforms=None):

        super().__init__()
        # sort order of frame from video sequence
        self.images = sorted([os.path.join(img_dir, e)
                              for e in os.listdir(img_dir)
                              if not e.split('.')[0].endswith('_L')])
        # remove text files
        self.images = [e for e in self.images if not e.endswith('.txt')]
        self.masks = sorted([os.path.join(img_dir, e)
                              for e in os.listdir(img_dir)
                              if e.split('.')[0].endswith('_L')])
        self.colormap = colormap
        self.num_classes = len(self.colormap) # 32 classes
        self.transforms = transforms

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

    def __getitem__(self, idx):

        img = Image.open(self.images[idx])
        mask = Image.open(self.masks[idx])

        if img.mode != 'RGB':
            img = img.convert('RGB')
        if mask.mode != 'RGB':
            mask = mask.convert('RGB')

        img = np.asarray(img) # change from image to array
        mask = np.asarray(mask)
        mask_channels = np.zeros(
            (mask.shape[0], mask.shape[1]), dtype=np.int64)

        # convert RGB mask to class-pixel mask ; (R,G,B) -> (Class)
        for i, cls in enumerate(self.colormap.keys()):
            color = self.colormap[cls]
            sub_mask = np.all(mask==color, axis=-1)*i
            mask_channels += sub_mask #*i

        # transforms such as normalization
        if self.transforms is not None:
            transformed = self.transforms(image=img, masks=mask_channels)
            img = transformed['image']
```

```
            mask_channels = transformed['masks']

        mask_channels = mask_channels.astype(np.float32)
        img = img.astype(np.float32) #/255

        instance = {'image': torch.from_numpy(img.transpose(2,0,1)),
                    'mask': torch.from_numpy(mask_channels)}

        return instance

    def ___first__(self):
        return self.__getitem__[0]
```

สร้าง transform

```
# simple transform (using ImageNet norm and std) "Albumentation ==
torchvision.transforms for segmentation"

transform = A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224,
0.225))
```

แบ่ง train val test

```
dataset = CamSeqDataset(img_dir='./Camseq_2007',
colormap=colormap,transforms=transform)
# we split train/val/test -> 70/15/15
train_size = int(len(dataset)*0.7)
val_size = (len(dataset)-train_size)//2
# train_set, rest = random_split(dataset, [train_size, len(dataset)-
train_size])
# val_set, test_set = random_split(rest, [val_size, len(rest)-val_size])
# We do not use random split because the dataset is extracted from a video
sequence, so nearly every frame looks the same.
train_set = Subset(dataset, range(train_size))
val_set = Subset(dataset, range(train_size, train_size + val_size))
test_set = Subset(dataset, range(train_size + val_size, len(dataset)))

batch_size = 2

train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_set, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_set, batch_size=batch_size, shuffle=True)
```

```
# loader size : 101 images (train/val/test) -> (70/16/15) -> batch size 2
(35/8/8)
```

```
len(train_loader), len(val_loader), len(test_loader)
```

```
(35, 8, 8)
```

function สำหรับ create model, และ train model

```
def create_model(out_channels=32):
    model =
torchvision.models.segmentation.deeplabv3_resnet50(pretrained=True)
    # decoder head
    model.classifier =
torchvision.models.segmentation.deeplabv3.DeepLabHead(
        2048, num_classes=out_channels)

    model.train()
    return model
```

```
def train_model(model,
                train_loader,
                val_loader,
                criterion= nn.CrossEntropyLoss(),
                num_epochs=1,
                device=torch.device('cpu'),
                lr=0.0002,
                model_path="./model.pth"
                ):

    model.to(device)
    optimizer = optim.Adam(model.parameters(), lr=lr)
    scheduler = lr_scheduler.StepLR(optimizer, step_size=4, gamma=0.5)
    metrics = {
          "train_losses" : [],
          "val_losses" : [],
          "train_acc" : [],
          "val_acc" : []
          }
    min_val_loss = 1e10
    for epoch in range(1, num_epochs + 1):
        tr_loss = []
        val_loss = []
        tr_acc = []
        val_acc = []
        model.train()
        print('Epoch {}/{}'.format(epoch, num_epochs))
        for sample in tqdm(train_loader):
            if sample['image'].shape[0]==1:
```

```
                break
            inputs = sample['image'].to(device)
            masks = sample['mask'].to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            y_pred = outputs['out']
            y_true = masks
            loss = criterion(y_pred.float(), y_true.long())
            acc = (torch.argmax(y_pred, 1) == y_true).float().mean()
            loss.backward()
            tr_loss.append(loss)
            tr_acc.append(acc)
            optimizer.step()
        scheduler.step()
        optimizer.zero_grad()
        avg_tr_loss = torch.mean(torch.Tensor(tr_loss))
        metrics["train_losses"].append(avg_tr_loss)
        avg_tr_acc = torch.mean(torch.Tensor(tr_acc))
        metrics["train_acc"].append(avg_tr_acc)
        print(f'Train loss: {avg_tr_loss}')
        print(f'Train acc: {avg_tr_acc}')

        # Validation phrase
        for sample in tqdm(val_loader):
            if sample['image'].shape[0]==1:
                break
            inputs = sample['image'].to(device)
            masks = sample['mask'].to(device)
            model.eval()
            with torch.no_grad():
                outputs = model(inputs)
            y_pred = outputs['out']
            y_true = masks
            loss = criterion(y_pred.float(), y_true.long())
            # acc using pixel accuracy. (it is easy to understand,but no
way the best metric)
            # learn more https://towardsdatascience.com/metrics-to-
evaluate-your-semantic-segmentation-model-6bcb99639aa2
            acc = (torch.argmax(y_pred, 1) == y_true).float().mean()
            val_loss.append(loss)
            val_acc.append(acc)
        avg_val_loss = torch.mean(torch.Tensor(val_loss))
        metrics["val_losses"].append(avg_val_loss)
        avg_val_acc = torch.mean(torch.Tensor(val_acc))
        metrics["val_acc"].append(avg_val_acc)
        print(f'val loss: {avg_val_loss}')
        print(f'val acc: {avg_val_acc}')
        # save model state that have best val loss
```

```
        if avg_val_loss < min_val_loss:
            torch.save(model.state_dict(), model_path)
            min_val_loss = avg_val_loss
    return model,metrics
```

สร้าง model

```
# set device
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# Assuming that we are on a CUDA machine, this should print a CUDA device:
lr =0.0002
print(device)
from torchinfo import summary as summary_info
model = create_model(out_channels=dataset.num_classes)
# model architecture
model
```

ดู summary ของ network

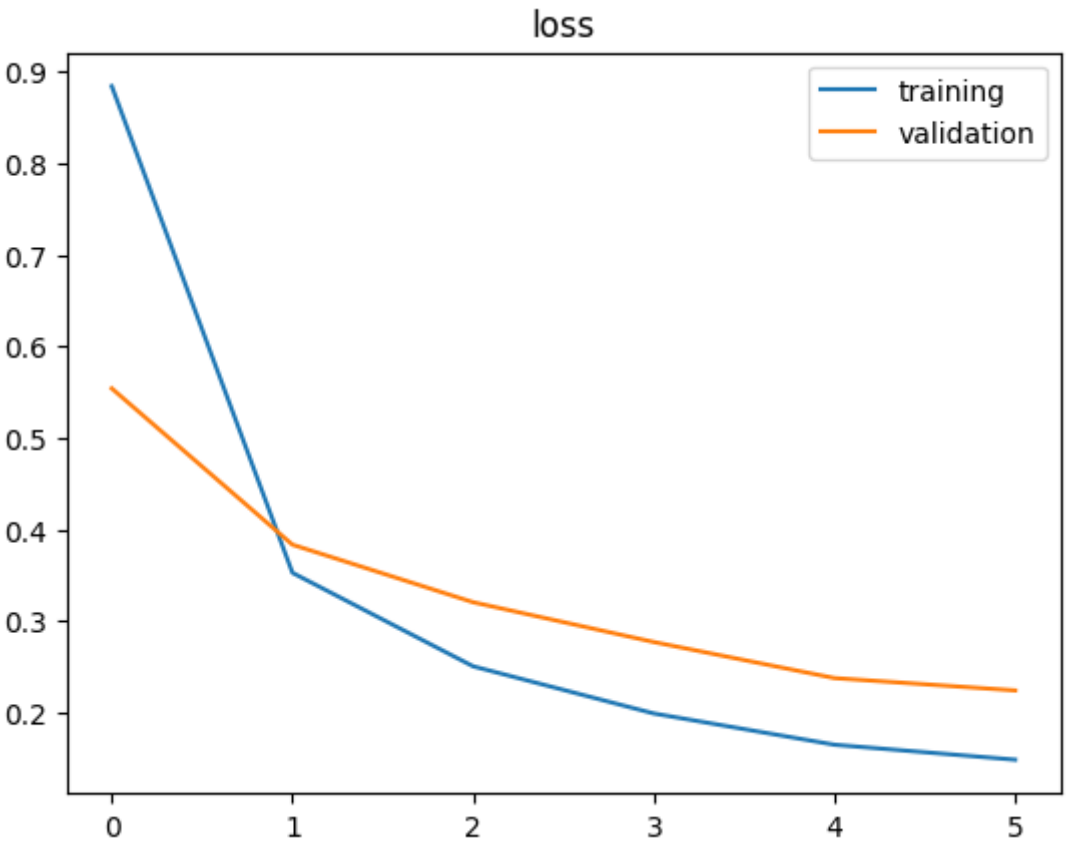
```
model.to(device)
# model summary that can pass dummy input to show output shape
# input shape desc : (batch, channel, H, W)
summary_info(model, input_size = (2, 3, 720, 960))
```

```
==========================================================================
==========================
Total params: 42,006,901
Trainable params: 42,006,901
Non-trainable params: 0
Total mult-adds (G): 913.76
==========================================================================
==========================
Input size (MB): 16.59
Forward/backward pass size (MB): 11731.92
Params size (MB): 168.03
Estimated Total Size (MB): 11916.54
==========================================================================
==========================
```

loss graph

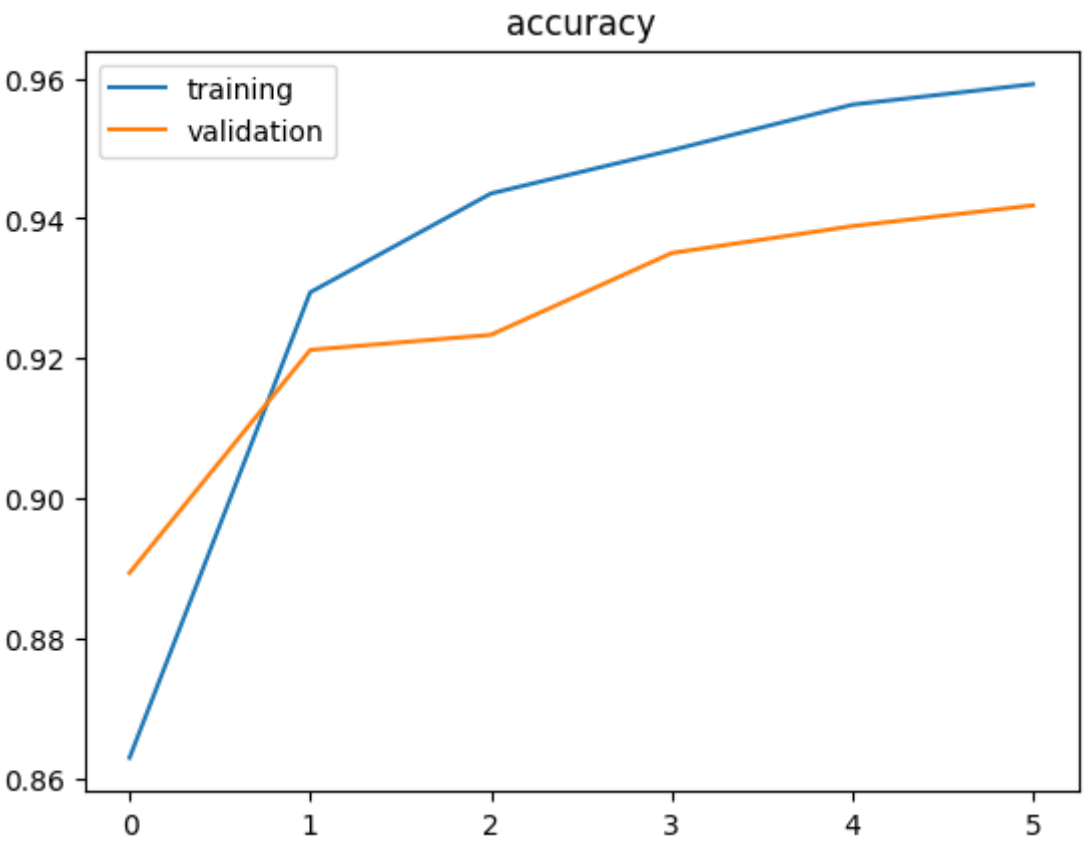
```
# loss graph
#overfitting because small dataset (only 70 image)
```

```
plt.plot(metrics["train_losses"], label = 'training')
plt.plot(metrics["val_losses"], label = 'validation')
plt.title("loss")
plt.legend()
```




plot pixel accuracy

```
# Pixel accuracy
plt.plot(metrics["train_acc"], label = 'training')
plt.plot(metrics["val_acc"], label = 'validation')
plt.title("accuracy")
plt.legend()
```




โหลด model ที่ save มาทำ evaluation

```
# load best val state

PATH = './model.pth'

model.load_state_dict(torch.load(PATH))
```

clear cache และ garbage collect

```
import gc
with torch.no_grad():
torch.cuda.empty_cache()
gc.collect()
```

```
model.eval()
accuracy = []
with torch.no_grad():
    for sample in tqdm(test_loader):
        if sample['image'].shape[0]==1:
            break
        inputs = sample['image'].to(device)
        masks = sample['mask'].to(device)

        outputs = model(inputs)
        y_pred = outputs['out']
```

```
        y_true = masks
        # acc using pixel accuracy. (it is easy to understand,but not the
best metric)
        acc = (torch.argmax(y_pred, 1) == y_true).float().mean()
        accuracy.append(acc)
accuracy = torch.mean(torch.Tensor(accuracy))
print(f'accuracy: {accuracy}')
```

```
>> accuracy: 0.9269694089889526
```

ควรใช้ IoU ในการวัด performance ของ segmentation task

function สำหรับ compute IoU

```
def compute_iou_batch(predictions, ground_truths, num_classes):
    """
    Compute Intersection over Union (IoU) for semantic segmentation over a
batch of images.

    Parameters:
    - predictions: A 4D torch tensor of shape (batch, class, height,
width) with predicted class scores
    - ground_truths: A 3D torch tensor of shape (batch, height, width)
with ground truth class labels
    - num_classes: Total number of classes in the segmentation task

    Returns:
    - A torch tensor of IoU values for each class
    """
    batch_size = predictions.size(0)
    iou_per_class = torch.zeros(num_classes, dtype=torch.float32)

    # Iterate over each class
    for cls in range(num_classes):
        intersection = 0
        union = 0

        # Compute IoU per image in the batch
        for i in range(batch_size):
            # Predicted mask for the current class
            pred_mask = (predictions[i, cls] > 0.5)  # Apply threshold for
binary mask
            # Ground truth mask for the current class
            gt_mask = (ground_truths[i] == cls)

            # Compute Intersection and Union
            intersection += torch.sum(pred_mask & gt_mask).item()
            union += torch.sum(pred_mask | gt_mask).item()
```

```
        # Compute IoU for the current class, avoid division by zero
        if union == 0:
            iou_per_class[cls] = float('nan')  # or 0 if you prefer
        else:
            iou_per_class[cls] = intersection / union

    return iou_per_class
```

Eval

```
iou_per_class_total = torch.zeros(num_classes, dtype=torch.float32)
num_batches = 0
num_classes = 32
model.eval()
with torch.no_grad():
    for sample in tqdm(test_loader):
        if sample['image'].shape[0]==1:
            break
        inputs = sample['image'].to(device)
        masks = sample['mask'].to(device)

        outputs = model(inputs)
        y_pred = outputs['out']
        y_true = masks
        # predictions, ground_truths = batch
        # predictions = torch.softmax(predictions, dim=1)  # Convert
logits to probabilities
        batch_iou = compute_iou_batch(y_pred, y_true, num_classes)
        iou_per_class_total += batch_iou
        num_batches += 1

average_iou_per_class = iou_per_class_total / num_batches
overall_average_iou = torch.nanmean(average_iou_per_class)
print(f'average_iou_per_class: {average_iou_per_class}')
print(f'overall_average_iou: {overall_average_iou}')
```

```
>> average_iou_per_class: tensor([ nan, 0.0000, 0.6856, nan, 0.7293,
0.0067, nan, 0.2761, 0.0370, 0.7496, 0.1374, nan, 0.0000, nan, 0.2079,
nan, 0.0309, 0.8530, nan, 0.4549, 0.0000, 0.7450, nan, nan, 0.5498, nan,
0.6805, 0.5452, nan, 0.0181, 0.0618, 0.6146]) overall_average_iou:
0.3515826165676117
```

```
model.eval()
img = next(iter(val_loader))
output = model(img['image'].to(device))['out']
```

function invTransofrom ใช้สำหรับแปลงรูปที่ normalized แล้ว กลับมาเป็นรูป original

```
# inverse transform (normalized image -> original)
def invTransform(img):
    img = img*torch.tensor([0.229, 0.224, 0.225]).mean() +
torch.tensor([0.485, 0.456, 0.406]).mean()     # unnormalize
    npimg = img.numpy().clip(0,255)
    return npimg
def imshow(img):
    npimg = invTransform(img)
    plt.figure(figsize=(16,16))
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

```
image = img['image'][0]

imshow(image)
```


```
fig, axes = plt.subplots(nrows=4, ncols=8, sharex=True, sharey=True,
figsize=(16,20))
```

```
axes_list = [item for sublist in axes for item in sublist]

thresh=0.3
res = output[0].detach().cpu().numpy()
for i, mask in enumerate(res):
    ax = axes_list.pop(0)
    ax.imshow(np.where(mask>thresh, 255, 0), cmap='gray')
    ax.set_title(list(colormap.keys())[i])

for ax in axes_list:
    ax.remove()

plt.tight_layout()
```


นำ output ผ่าน argmax

```
seg = torch.argmax(output[0], 0)
seg = seg.cpu().detach().numpy()
```

```
plt.figure(figsize=(16,16))
plt.imshow(seg)  # display image
plt.show()
```

```
oiginal_image = invTransform(img['image'][0])
oiginal_image = np.transpose(oiginal_image, (1, 2, 0))
f, axarr = plt.subplots(1,2,figsize= (16,16))
axarr[0].imshow(seg)
axarr[1].imshow(oiginal_image)
plt.show()
```