

# Deep Learning for Computer Vision HW2 Report

資工所碩二 R08922086 蔡承運

## Problem 1: Image Classification

### (1) Print the network architecture of my model

```
ResNet18(  
  (backbone): ResNet(  
    (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)  
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
    (relu): ReLU(inplace=True)  
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)  
    (layer1): Sequential(  
      (0): BasicBlock(  
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        (relu): ReLU(inplace=True)  
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
      )  
      (1): BasicBlock(  
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        (relu): ReLU(inplace=True)  
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
      )  
    )  
  )  
)
```

```
    (layer2): Sequential(  
      (0): BasicBlock(  
        (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)  
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        (relu): ReLU(inplace=True)  
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        (downsample): Sequential(  
          (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)  
          (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        )  
      )  
      (1): BasicBlock(  
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        (relu): ReLU(inplace=True)  
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
      )  
    )  
  )  
)
```

```
    (layer3): Sequential(  
      (0): BasicBlock(  
        (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)  
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        (relu): ReLU(inplace=True)  
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        (downsample): Sequential(  
          (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)  
          (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        )  
      )  
      (1): BasicBlock(  
        (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
        (relu): ReLU(inplace=True)  
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)  
      )  
    )  
  )  
)
```

```
(layer4): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
(avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
(fc): Linear(in_features=512, out_features=50, bias=True)
)
```

The backbone is exactly the same as standard ResNet18. I only modified the last layer (fc) to change the output dimension to the desired one (50).

For more detailed information, please refer to “src/models/p1\_model.txt”

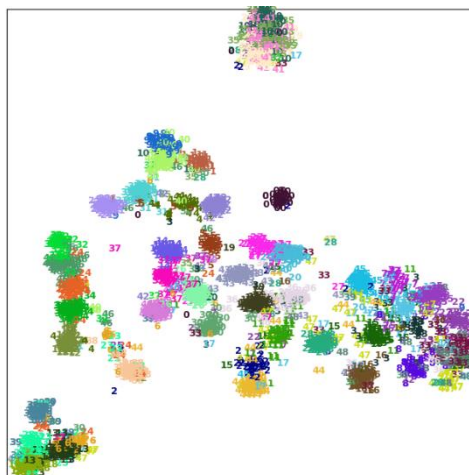
## (2) Report accuracy of the model on the validation set

ResNet18 Accuracy on the validation set: **81.76%** (loss=0.596576)

Remarks: other models tried

ResNet34:	loss=0.574860	Acc=82.84%
ResNeXt101:	loss=0.398426	Acc=89.01%
VGG16:	loss=0.67792	Acc=79.76%

## (3) Visualize classification results on validation set using t-SNE on the output features of the 2<sup>nd</sup> last layer and briefly explain the result.



As shown by the figure, data are split into different clusters, with data in clusters being the same class. Only some of the data remain unsplit on the top of the figure. I guess those data is the reason for imperfect accuracy.

## Problem 2: Semantic Segmentation

### (1) Print the model architecture of VGG16-FCN32s

```
FCN32s(
  (backbone): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv_block): Sequential(
    (0): Conv2d(512, 4096, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Dropout2d(p=0.1, inplace=False)
    (3): Conv2d(4096, 4096, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): Dropout2d(p=0.1, inplace=False)
  )
  (classifier): Sequential(
    (0): Conv2d(4096, 7, kernel_size=(1, 1), stride=(1, 1))
    (1): ReLU(inplace=True)
  )
  (upsample): ConvTranspose2d(7, 7, kernel_size=(32, 32), stride=(32, 32))
)
```

The backbone part is the same as the convolution part in VGG16. I only remove the fully-connected layers after the convolution layers (layers behind layer 30.) Instead, I append three convolution layers with much more channels for further feature extractions (“conv\_block” and “classifier”). Finally, features are upsampled 32 times via transpose convolution (“upsample”). The hyper-parameters and the architectures for the appended layers follows the original paper (FCN32s in “Fully Convolutional Networks for Semantic Segmentation”). More detailed information can be found in “src/models/ p2\_model\_baseline.txt

(2) Show the predicted mask of image id: 0010, 0097, 0107 during the early, middle, and the final stage of the training of VGG16-FCN32s

Image ID	Early stage (epoch 1)	Middle stage (epoch 6)	Final stage (epoch 15)	Ground Truth
0010				
0097				
0107				

(3) Print the improved model architecture (better than VGG16-FCN32s)

```
FCN8s(
  (features3): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
)
```

```
(features4): Sequential(
  (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace=True)
  (4): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (5): ReLU(inplace=True)
  (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
```



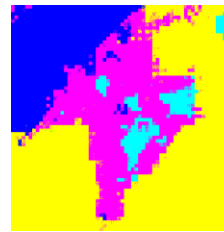
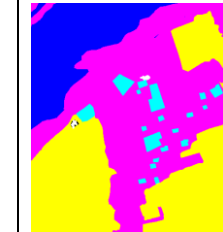
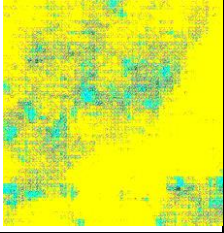
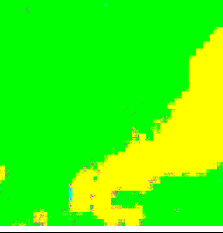
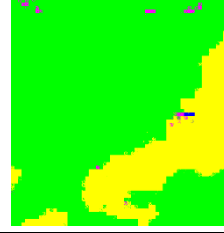
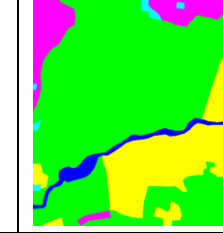
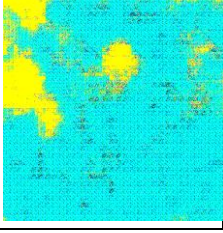

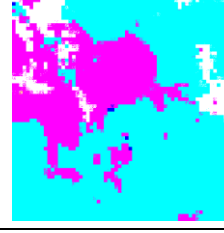
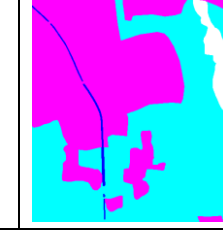
```

(features5): Sequential(
  (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace=True)
  (4): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (5): ReLU(inplace=True)
  (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
(fcn): Sequential(
  (0): Conv2d(512, 4096, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): Dropout2d(p=0.1, inplace=False)
  (3): Conv2d(4096, 4096, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (4): ReLU(inplace=True)
  (5): Dropout2d(p=0.1, inplace=False)
)

(score5): Conv2d(4096, 7, kernel_size=(1, 1), stride=(1, 1))
(upsample_5): ConvTranspose2d(7, 7, kernel_size=(2, 2), stride=(2, 2))
(score4): Conv2d(512, 7, kernel_size=(1, 1), stride=(1, 1))
(upsample_4): ConvTranspose2d(7, 7, kernel_size=(2, 2), stride=(2, 2))
(score3): Conv2d(256, 7, kernel_size=(1, 1), stride=(1, 1))
(upsample_3): ConvTranspose2d(7, 7, kernel_size=(8, 8), stride=(8, 8))
)

```

(4) Show the predicted mask of image id: 0010, 0097, 0107 during the early, middle, and the final stage of the training of the improved model

Image ID	Early stage (epoch 1)	Middle stage (epoch 6)	Final stage (epoch 15)	Ground Truth
0010				
0097				
0107				



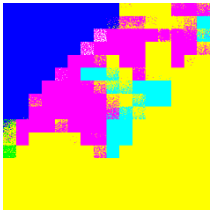
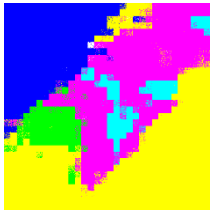
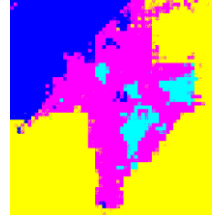
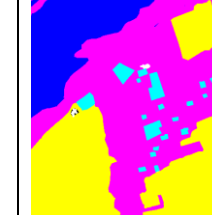
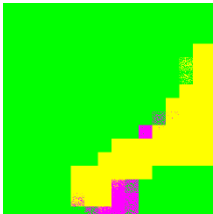
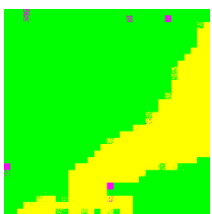
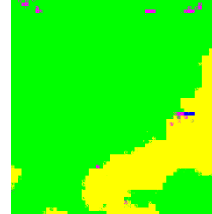
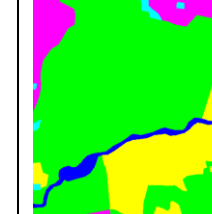
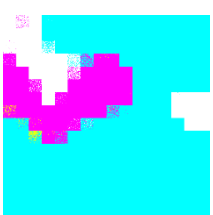

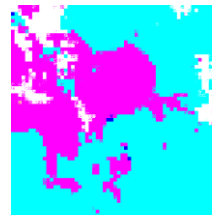

- (5) Report mIOU score of both models on the validation set. Discuss why the improved model performs better by conducting experiments to support it.

Model	mIOU score	Loss
Baseline: <u>VGG16-FCN32s</u>	<u>0.685135</u>	0.370613
Improved: <u>VGG16-FCN8s</u>	<u>0.697265</u>	0.340346

#### Discussion

As channels increases and feature map sizes (height, width) decrease, spatial information is also loss owing to saturated size. In (VGG16)FCN32s, model is trying to reconstruct the whole picture using only  $1/32 * 1/32 = 1/1024$  of the size. Although shrinking the size of feature maps forces the model to learn the most essential information, however, the size is too small and **it's incredibly hard for the model to learn subtle information in local areas purely from such small feature maps.**

In contrast, (VGG16)FCN8s improves the defect of the above problem. It provides shortcuts from layers of lower-level, when reconstructing the pixel-wise predictions. Therefore, **subtle information can be passed and restored** by the model. In the following table, I plot the different prediction results for (VGG16)FCN32s, (VGG16)FCN16s, (VGG16)FCN8s to see the difference. As evidenced by the results, FCN do **capture much finer** results, which corroborates the claims above.

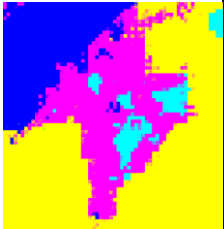
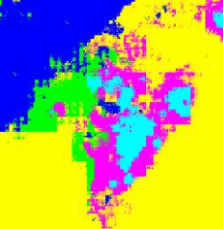
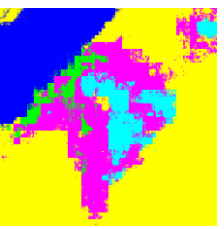

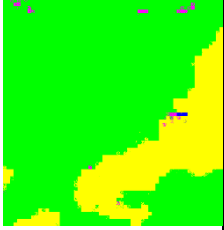
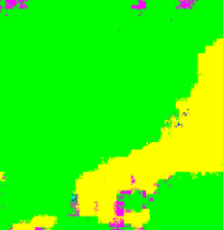

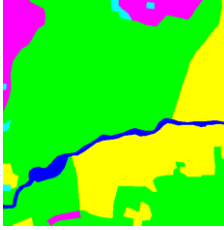
Types	FCN32s	FCN16s	FCN8s	Ground Truth
0010				
0097				
0107				

Remarks: other models tried

Model	Loss	mIOU score
FCN16s	0.345898	0.686301
FCN8s	0.340346	0.697265
FCN8s, fine-tuned with Lovasz loss	0.357549	0.705292
FCN4s	0.442913	0.632384
FCN2s	0.387345	0.672525
UNet16	0.460651	0.634736
SegNet	0.562292	0.549803
DeepLabV3-ResNet101	0.346967	0.728019
DeepLabV3-ResNet101, fine-tuned with Lovasz loss	<b>0.339132</b>	<b>0.736061</b>

Remarks: other models tried

Following the same method to transform from FCN32s to FCN16s or FCN8s, I also tried FCN4s and FCN2s. However, the results are not as good as FCN8s. This is probably due to the fact that the low-level features are not formed properly enough to provide useful information when the model “decodes.” In these examples, the additional information even harms the final results.

Types	FCN8s	FCN4s	FCN2s	Ground Truth
mIOU	0.697265	0.681951	0.672525	1
0010				
0097				
0107	