ECE549 Computer Vision: Assignment 4

Yutong Xie

NetID: yutongx6

Instructor: Prof. Saurabh Gupta

Apr 28, 2020

1. Improve BaseNet on FashionMNIST

In this part, I worked with the FashionMNIST dataset to improve BaseNet from different aspects. Next, I will introduce four different factors that affected the model performance.

1.1 Best Model

The best model can achieve **93%** accuracy on the test set. The architecture of my network is shown below.

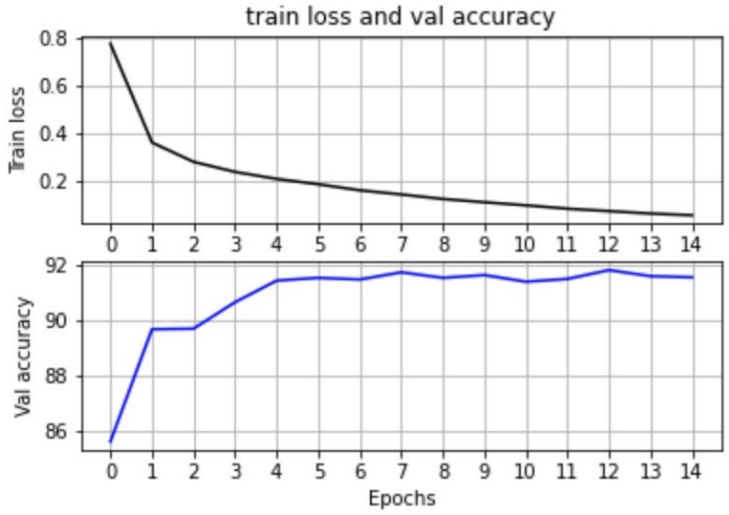
Improved BaseNet Architecture						
Layer No.	*Layer Type*	*Kernel Size*	*Input Dim*	*Output Dim*	*Input Channels*	*Output Channels*
1	conv2d	3	28	26	1	6
2	BatchNorm2d	-	26	26	6	6
3	relu	-	26	26	6	6
4	conv2d	3	26	24	6	16
5	BatchNorm2d	-	24	24	16	16
6	relu	-	24	24	16	16
7	maxpool2d	3	12	12	16	16
8	conv2d	2	12	10	16	64
9	BatchNorm2d	-	10	10	64	64
10	relu	-	10	10	64	64
11	conv2d	3	10	8	64	128
12	BatchNorm2d	-	8	8	128	128
13	relu	-	8	8	128	128
14	maxpool2d	2	4	4	128	128
15	conv2d	3	4	2	128	256
16	BatchNorm2d	-	2	2	256	256
17	relu	-	2	2	256	256
18	maxpool2d	3	1	1	256	256
19	linear	-	1	1	256	128
20	BatchNorm1d	-	1	1	128	128
21	relu	-	1	1	128	128
22	linear	-	1	1	128	5
23	BatchNorm1d	-	1	1	5	5
24	relu	-	1	1	5	5
25	linear	-	1	1	5	10

The test result on the test set is shown below.

```
[40] ################################################################################
     # 5. Try the network on test data, and report your performance in the submission r
     # ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
     ################################################################################

     # Check out why .eval() is important!
     # https://discuss.pytorch.org/t/model-train-and-model-eval-vs-model-and-model-eval
     net.eval()

     test_accuracy, test_classwise_accuracy = \
             calculate_accuracy(testloader, IS_GPU)
     print('Accuracy of the network on the test images: %d %%' % (test_accuracy))
```

```
Accuracy of the network on the test images: 93 %
```

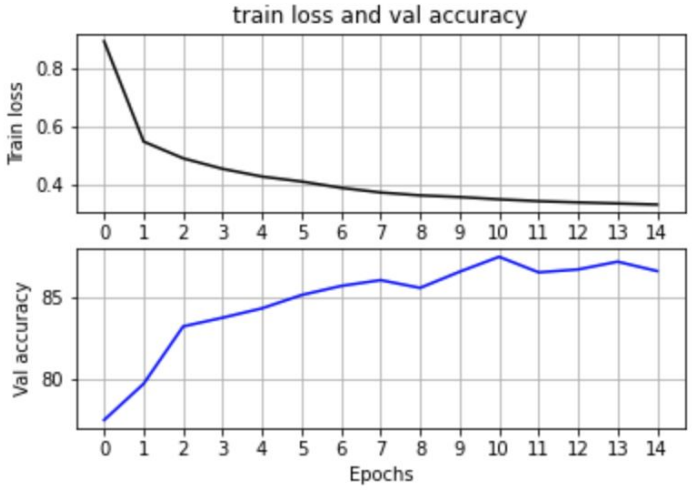
And the training plot and validation accuracy plot for final model are shown here.



1.2. Factors that helped improved model performance

In this part, I tried to analyze different factors that affected the model, I will compare the performance of them with the basic BaseNet. For the original BaseNet, the accuracy on validation and test set are shown below.

```
[1] loss: 0.890
Accuracy of the network on the val images: 77 %
[2] loss: 0.547
Accuracy of the network on the val images: 79 %
[3] loss: 0.490
Accuracy of the network on the val images: 83 %
[4] loss: 0.454
Accuracy of the network on the val images: 83 %
[5] loss: 0.427
Accuracy of the network on the val images: 84 %
[6] loss: 0.410
Accuracy of the network on the val images: 85 %
[7] loss: 0.389
Accuracy of the network on the val images: 85 %
[8] loss: 0.373
Accuracy of the network on the val images: 86 %
[9] loss: 0.363
Accuracy of the network on the val images: 85 %
[10] loss: 0.357
Accuracy of the network on the val images: 86 %
[11] loss: 0.349
Accuracy of the network on the val images: 87 %
[12] loss: 0.343
Accuracy of the network on the val images: 86 %
[13] loss: 0.339
Accuracy of the network on the val images: 86 %
[14] loss: 0.335
Accuracy of the network on the val images: 87 %
[15] loss: 0.332
Accuracy of the network on the val images: 86 %

Finished Training
```




```
Accuracy of the network on the test images: 87 %
```

1.2.1 Data normalization

For this part, I tried to normalize input data to make training easier and more robust. Using transforms.Normalize() can help us to normalize the input data to zero mean and fixed standard deviation. After applying data normalization, all the images have similar

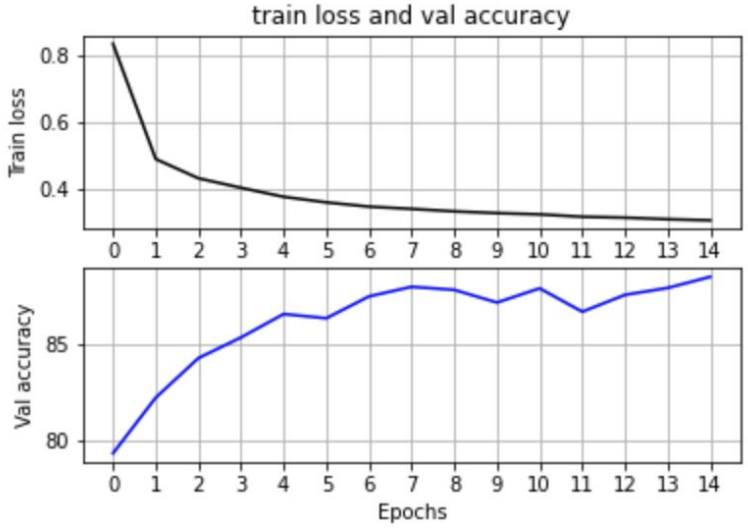
distribution, which will make the model converge fast. I made all the data to become zero mean and 1 sigma using following instructions.

```
train_transform = transforms.Compose(
    [
     transforms.ToTensor(),
     transforms.Normalize(mean = (0.5, ), std = (0.5, ))
     ])
test_transform = transforms.Compose(
    [
     transforms.ToTensor(),
     transforms.Normalize(mean = (0.5, ), std = (0.5, ))
    ])
```

Because, we need to test the model performance on test set, we also need to normalize the test set.

After training the new model, I find that the performance was improved from 87% accuracy to 88% accuracy, which means that data normalization is helpful.

```
Accuracy of the network on the test images: 88 %
```



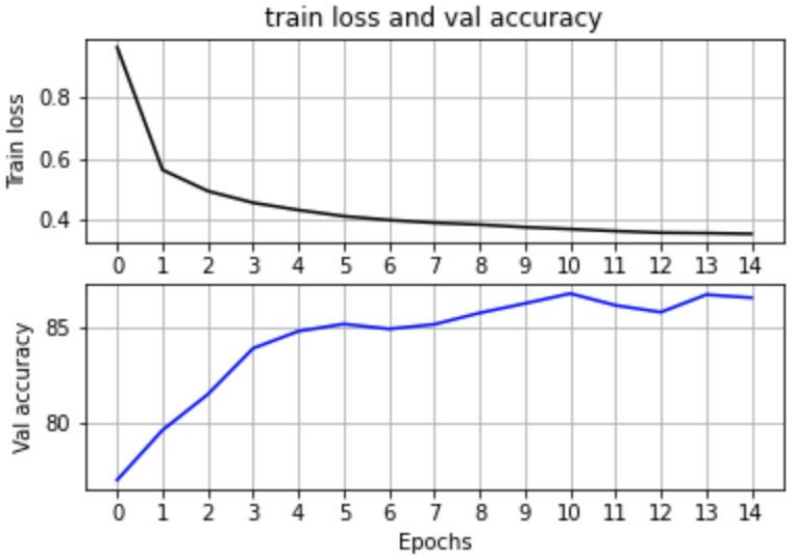
1.2.2 Data Augmentation

In the part, I augment the training data using horizontal flips. The convolutional neural network are very prone to overfitting problems. Thorugh data augmentation, there will generate more “new” samples to solve the problem.

The code is here.

After training the new model, I find that the performance keeps 87% accuracy, which means that data augmentation is not very helpful.

```
Accuracy of the network on the test images: 87 %
```



1.2.3 Deeper Network

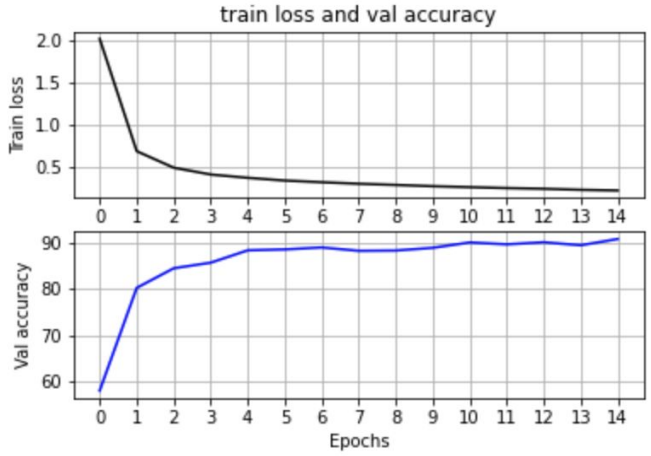
In this part, I tried to make the network deeper by adding more convolutional and fully connected layers. Multiple stacked CONV layers can develop more complex features of the input volume before the destructive pooling operation. The modified network architecture is shown below.

`INPUT -> [CONV -> RELU -> CONV -> RELU -> POOL]*2 -> [CONV -> RELU -> POOL] -> [FC -> RELU]*2 -> FC`

In this part, I use CONV layers with 3x3 kernel size, because the input images are only 28x28.

After training the new model, I find that the performance was improved from 87% accuracy to 91% accuracy, which means that creating deeper network is helpful.

```
Accuracy of the network on the test images: 91 %
```



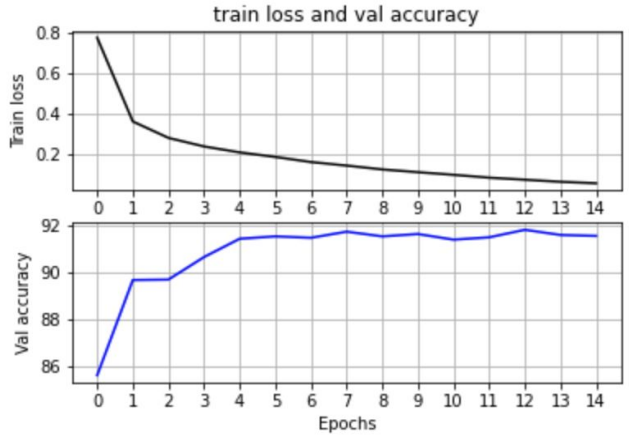
1.2.4 Normalization layers

In this part, I tried to add normalization layers after conv layers and linear layers. Normalization layers can reduce overfitting and improve training of the model. BatchNorm2d is needed after Conv layers and BatchNorm1d can be used after fully connected layers. The channel number of normalization layers should corresponds to the previous conv layers or linear layers. After experiment, I found that there is no great difference between inserting normalization layers before or after ReLu layers. Hence, I inserted the normalization layers before ReLu layers and the network architecture can be modified from the last part.

```
INPUT -> [CONV -> BatchNorm -> RELU -> CONV -> BatchNorm -> RELU -> POOL]*2 -> [CONV -> BatchNorm -> RELU -> POOL] -> [FC -> BatchNorm -> RELU]*2 -> FC
```

After modified the model, the accuracy increased to 93%, which means that normalization layers are useful.

```
Accuracy of the network on the test images: 93 %
```



2. Semantic Segmentation

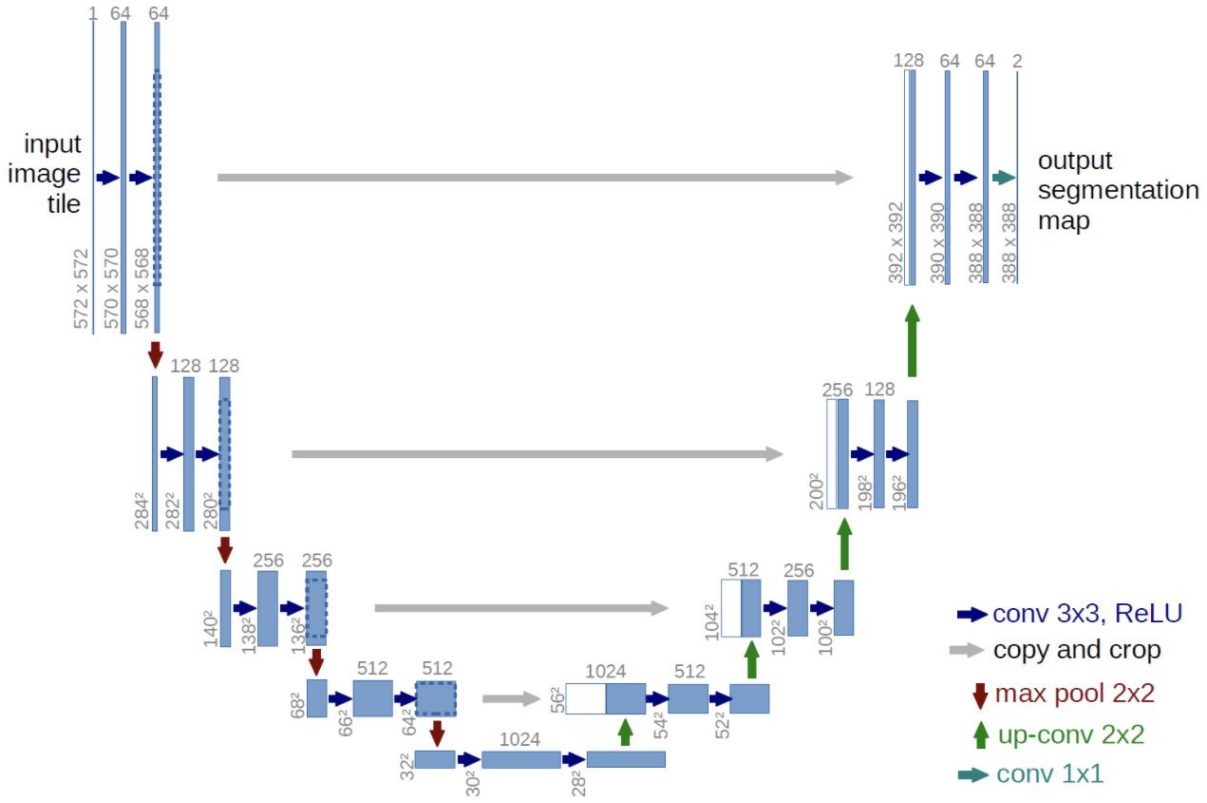
In this part, I build my own semantic segmentation model on the Stanford Background Dataset. I need to classify image pixels into 9 different categories.

2.1 Develop your model

Before designing my model, I implement training cycle and save the best model based on the loss function.

```
        if running_loss < best_loss:
          print("saving best model \n")
          best_loss = running_loss
          best_model_wts = copy.deepcopy(model.state_dict())


    model.load_state_dict(best_model_wts)
```

After implementing the training cycle, I create a model based on U-Net proposed in 2015 by Ronneberger. The architecture of U-Net is shown below. The network first downsample the inputs and conduct the convolution, then upsample for outputs.



In the original network, the output size is different with input size, because the skip connection part includes image cropping. However, in my implementation, I keep them same

for easy comparison with ground truth images. Also, I add batch normalization layers after each convolution layer to reduce overfitting.

Before each max pooling layer, the image will be processed by two convolutional layers, batch normalization layers and ReLu layers, so I define a Conv function to conduct the operation.

```
def Conv(Cin,Cout):
    down = nn.Sequential(
        nn.Conv2d(Cin,Cout,3,padding=1,stride=1),
        nn.BatchNorm2d(Cout),
        nn.ReLU(inplace=True),
        nn.Conv2d(Cout,Cout,3,padding=1,stride=1),
        nn.BatchNorm2d(Cout),
        nn.ReLU(inplace=True),
    )
    return down
```

The total network architecture is shown below. (I used CONV instead of six layers shown above for simplicity)

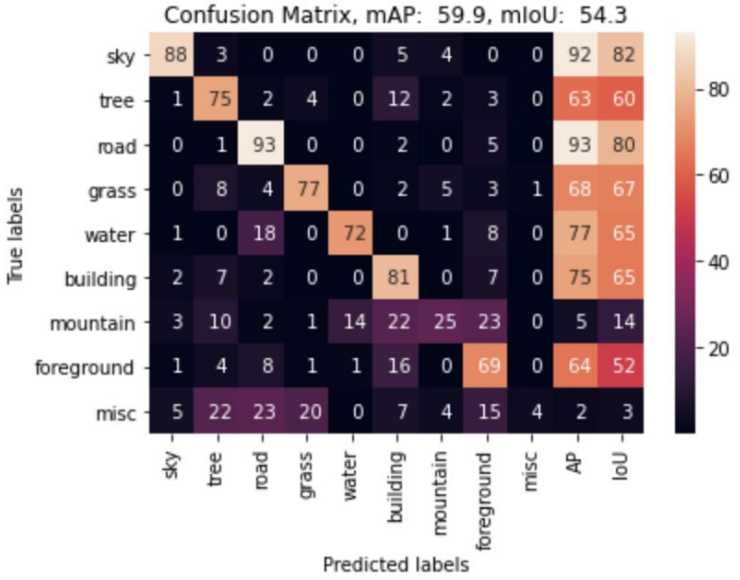
Modified U-Net Architecture						
Layer No.	*Layer Type*	*Kernel Size*	*Input Dim*	*Output Dim*	*Input Channels*	*Output Channels*
1	Conv	3	288x224	288x224	3	64
2	maxpool2d	2	288x224	144x112	64	64
3	Conv	3	144x112	144x112	64	128
4	maxpool2d	2	144x112	72x56	128	128
5	Conv	3	72x56	72x56	128	256
6	maxpool2d	2	72x56	36x28	256	256
7	Conv	3	36x28	36x28	256	512
8	maxpool2d	2	36x28	18x14	512	512
9	Conv	3	18x14	18x14	512	1024
10	ConvTranspose2d	3	18x14	36x28	1024	512
	concatenate outputs	-	36x28	36x28	512	1024
11	Conv	3	36x28	36x28	1024	512
12	ConvTranspose2d	3	36x28	72x56	512	256
	concatenate outputs	-	72x56	72x56	256	512
13	Conv	3	72x56	72x56	512	256
14	ConvTranspose2d	3	72x56	144x112	256	128
	concatenate outputs	-	144x112	144x112	128	256
15	Conv	3	144x112	144x112	256	128
16	ConvTranspose2d	3	144x112	288x224	128	64
	concatenate outputs	-	288x224	288x224	64	128
17	Conv	3	288x224	288x224	128	64
18	conv2d	1	288x224	288x224	64	9

Also, for easier visulization, I use torch.summary to demonstrate the network architecture.

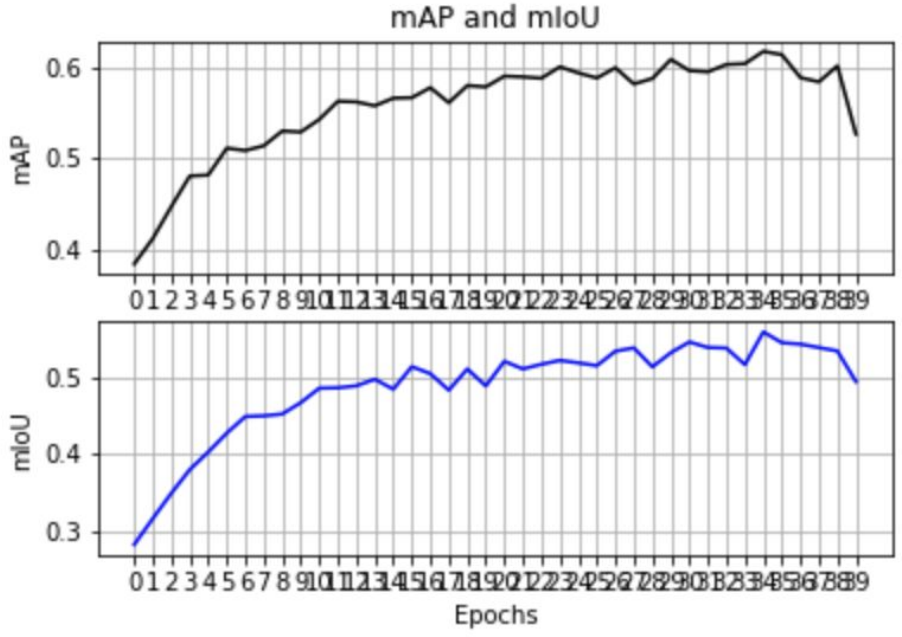
Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 288, 224]	1,792
BatchNorm2d-2	[-1, 64, 288, 224]	128
ReLU-3	[-1, 64, 288, 224]	0
Conv2d-4	[-1, 64, 288, 224]	36,928
BatchNorm2d-5	[-1, 64, 288, 224]	128
ReLU-6	[-1, 64, 288, 224]	0
MaxPool2d-7	[-1, 64, 144, 112]	0
Conv2d-8	[-1, 128, 144, 112]	73,856
BatchNorm2d-9	[-1, 128, 144, 112]	256
ReLU-10	[-1, 128, 144, 112]	0
Conv2d-11	[-1, 128, 144, 112]	147,584
BatchNorm2d-12	[-1, 128, 144, 112]	256
ReLU-13	[-1, 128, 144, 112]	0
MaxPool2d-14	[-1, 128, 72, 56]	0
Conv2d-15	[-1, 256, 72, 56]	295,168
BatchNorm2d-16	[-1, 256, 72, 56]	512
ReLU-17	[-1, 256, 72, 56]	0
Conv2d-18	[-1, 256, 72, 56]	590,080
BatchNorm2d-19	[-1, 256, 72, 56]	512
ReLU-20	[-1, 256, 72, 56]	0
MaxPool2d-21	[-1, 256, 36, 28]	0
Conv2d-22	[-1, 512, 36, 28]	1,180,160
BatchNorm2d-23	[-1, 512, 36, 28]	1,024
ReLU-24	[-1, 512, 36, 28]	0
Conv2d-25	[-1, 512, 36, 28]	2,359,808
BatchNorm2d-26	[-1, 512, 36, 28]	1,024
ReLU-27	[-1, 512, 36, 28]	0
MaxPool2d-28	[-1, 512, 18, 14]	0
Conv2d-29	[-1, 1024, 18, 14]	4,719,616
BatchNorm2d-30	[-1, 1024, 18, 14]	2,048
ReLU-31	[-1, 1024, 18, 14]	0
Conv2d-32	[-1, 1024, 18, 14]	9,438,208
BatchNorm2d-33	[-1, 1024, 18, 14]	2,048
ReLU-34	[-1, 1024, 18, 14]	0
ConvTranspose2d-35	[-1, 512, 36, 28]	4,719,104
Conv2d-36	[-1, 512, 36, 28]	4,719,104
BatchNorm2d-37	[-1, 512, 36, 28]	1,024
ReLU-38	[-1, 512, 36, 28]	0
Conv2d-39	[-1, 512, 36, 28]	2,359,808
BatchNorm2d-40	[-1, 512, 36, 28]	1,024
ReLU-41	[-1, 512, 36, 28]	0
ConvTranspose2d-42	[-1, 256, 72, 56]	1,179,904
Conv2d-43	[-1, 256, 72, 56]	1,179,904
BatchNorm2d-44	[-1, 256, 72, 56]	512
ReLU-45	[-1, 256, 72, 56]	0
Conv2d-46	[-1, 256, 72, 56]	590,080
BatchNorm2d-47	[-1, 256, 72, 56]	512
ReLU-48	[-1, 256, 72, 56]	0
ConvTranspose2d-49	[-1, 128, 144, 112]	295,040
Conv2d-50	[-1, 128, 144, 112]	295,040
BatchNorm2d-51	[-1, 128, 144, 112]	256
ReLU-52	[-1, 128, 144, 112]	0
Conv2d-53	[-1, 128, 144, 112]	147,584
BatchNorm2d-54	[-1, 128, 144, 112]	256
ReLU-55	[-1, 128, 144, 112]	0
ConvTranspose2d-56	[-1, 64, 288, 224]	73,792
Conv2d-57	[-1, 64, 288, 224]	73,792
BatchNorm2d-58	[-1, 64, 288, 224]	128
ReLU-59	[-1, 64, 288, 224]	0
Conv2d-60	[-1, 64, 288, 224]	36,928
BatchNorm2d-61	[-1, 64, 288, 224]	128
ReLU-62	[-1, 64, 288, 224]	0
Conv2d-63	[-1, 9, 288, 224]	585

After implementing the model, I trained for 40 epochs and saved the best model. The test result on test set is shown below, which is satisfied the basic need.

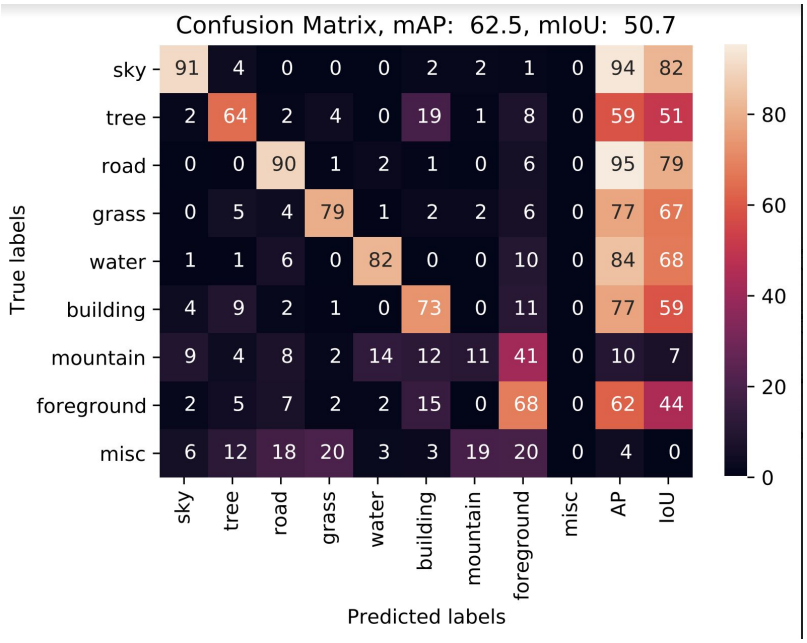
```
       sky: AP: 0.92, IoU: 0.82
      tree: AP: 0.63, IoU: 0.60
      road: AP: 0.93, IoU: 0.80
     grass: AP: 0.68, IoU: 0.67
     water: AP: 0.77, IoU: 0.65
  building: AP: 0.75, IoU: 0.65
  mountain: AP: 0.05, IoU: 0.14
foreground: AP: 0.64, IoU: 0.52
      misc: AP: 0.02, IoU: 0.03
      mean: AP: 0.60, IoU: 0.54
```

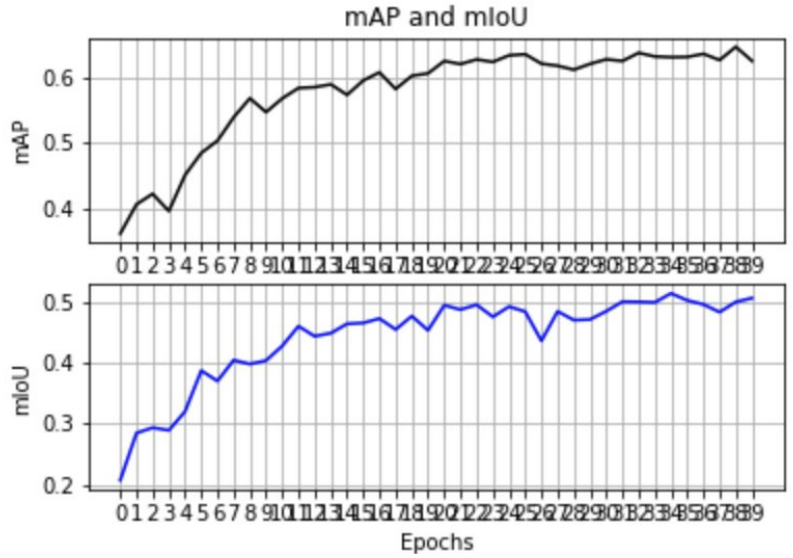


I also plot the mAP and mIoU for each epoch tested on validation set.



I tried different optimizer and compare their performance, which includes SGD and Adam. I found that the SGD approach will provide higher mIoU, but Adam approach will provide higher mAP. The following figures are result of approach Adam.





2.2 Build on top of ImageNet pre-trained Model

In this part, I build the model based on pre-trained ResNet18 model. First, I removed the last two layers of ResNet including classifier and global average pooling layers. The ResNet can be seen as the downsample section of U-Net, so that I stack the ResNet with the upsample part of the U-Net for semantic segmentation. For the purpose of assignment, I kept the part of the model derived from ResNet model fixed using the following code.

```
resnet18 = models.resnet18(pretrained=True)

for param in resnet18.parameters():
  param.requires_grad = False
```

I set the requires_grad properties of parameters as False, so that they will not be updated.

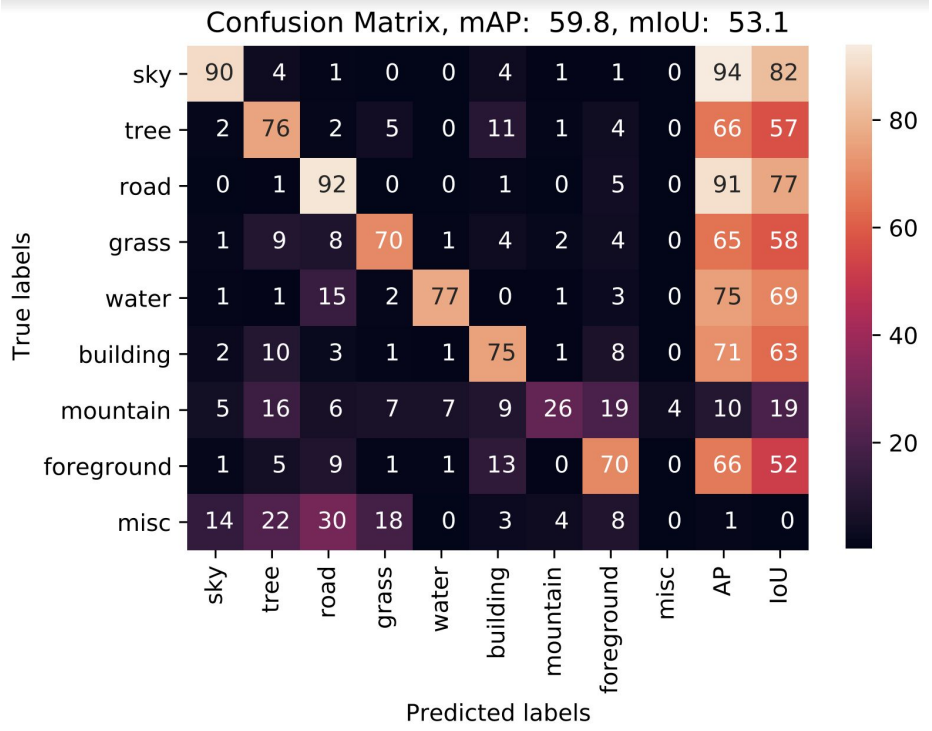
Then, I stack the resnet except for the last two layers on the upsample network defined below.

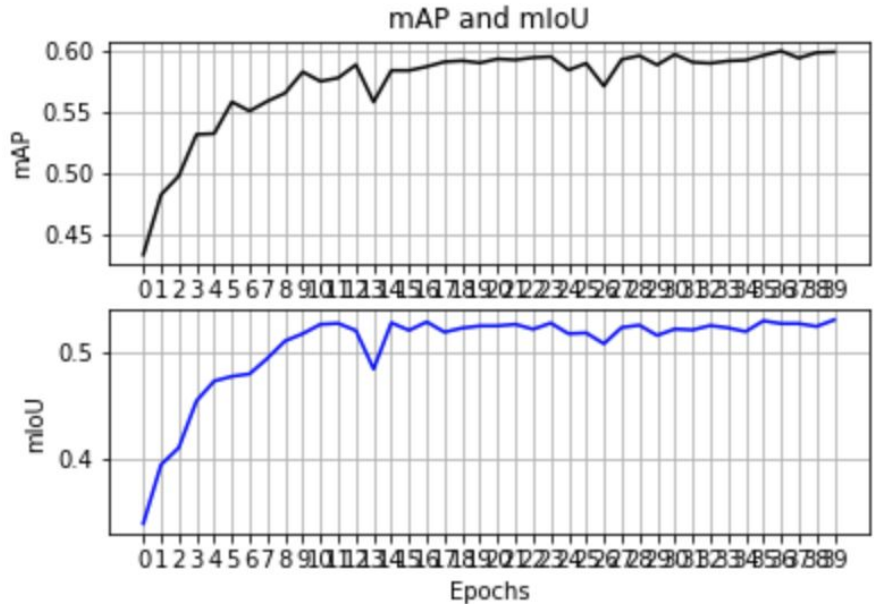
```
    resnet18 = list(resnet18.children())
    self.preprocess = Conv(3, 64)
    self.down0 = nn.Sequential(*resnet18[:3])
    self.down1 = nn.Sequential(*resnet18[3:5])
    self.down2 = resnet18[5]
    self.down3 = resnet18[6]
    self.down4 = resnet18[7]

    self.upsample = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)

    self.conv1 = Conv(256 + 512, 512)
    self.conv2 = Conv(128 + 512, 256)
    self.conv3 = Conv(64 + 256, 256)
    self.conv4 = Conv(64 + 256, 128)
    self.conv5 = Conv(64 + 128, 64)
    self.conv6 = nn.Conv2d(64, 9, 1)

def forward(self, x):
    x_raw = self.preprocess(x)

    x0 = self.down0(x)
    x1 = self.down1(x0)
    x2 = self.down2(x1)
    x3 = self.down3(x2)
    x4 = self.down4(x3)

    x = self.upsample(x4)
    x = torch.cat((x, x3), dim=1)
    x = self.conv1(x)

    x = self.upsample(x)
    x = torch.cat((x, x2), dim=1)
    x = self.conv2(x)

    x = self.upsample(x)
    x = torch.cat((x, x1), dim=1)
    x = self.conv3(x)

    x = self.upsample(x)
    x = torch.cat((x, x0), dim=1)
    x = self.conv4(x)

    x = self.upsample(x)
    x = torch.cat((x, x_raw), dim=1)
    x = self.conv5(x)

    x = self.conv6(x)

    return x
```

From the chart below, we can see that ResNet18 will downsample the input to the 1/32 size of the original one. Hence, We need 5 deconvolution layers to recover the size to 288x224.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

After implementing the model, I trained them for 40 epochs. The model can achieved the least requirement on validation set and test set. The best model can achieved 0.60 mAP and 0.53 mIoU. The performance is shown below.




mAP and mIoU
mAP
0.60
0.55
0.50
0.45
mIoU
0.5
0.4
Epochs