Name: Nguyen Thanh Vinh

ID: 20200855

Final Project: Semantic Segmentation of Photographic Images

I. Abstract

Image segmentation is the task of partitioning a digital into multiple segments. This process helps to locate the objects and their boundaries for further data analysis. There are two main group of image segmentation: Semantic segmentation and Instance segmentation. Semantic segmentation aims to assign a class to every pixel in the image while instance segmentation identify for every pixel, which instance it belongs to (instance-aware).

This project focus on semantic segmentation task. The goal is to successfully train a convolutional neural network that can be used to label each pixel of the images to 1 of 9 provided classes (the boundary class is excluded). With 1000 original pairs of image and label, the final model with basic UNet architecture achieved 64.86% mean Intersection of Union (mIoU) on the test set.

II. Phases Development Summary

1. Baseline Model and Development Approach

My original plan was firstly achieve as high as possible training mIoU, then I would handle the overfitting by using regularization methods and fine-tuning hyper parameters. After all, it is very hard to achieve high result for test dataset if model's performance on training dataset is poor.

To gain a better understanding of how a semantic segmentation model is different from a basic image classification one, the baseline model was built with only Conv2d layers go along with ReLU activation layers. The MaxPool2d layers were removed and used padding='same' to keep the image size unchanged for the output. Other parameters are: Epochs=100; kernel size=3; batch size=10 (higher batch size caused out of memory error); Crop size=128; Optimizer=Adam (lr = 1e-3, no learning-rate scheduler); No extra transformation.

Model: [Conv2d->ReLU] * 5

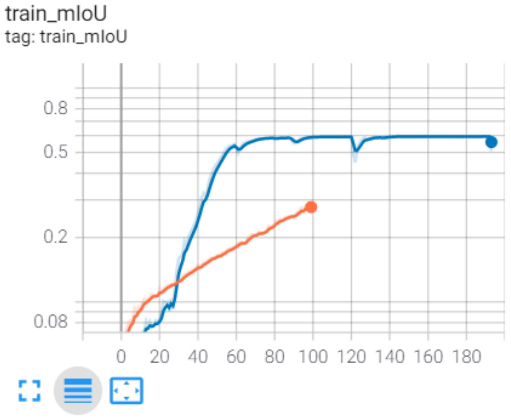
Channels: 3 -> 30 -> 90 -> 270 -> 810 -> 10

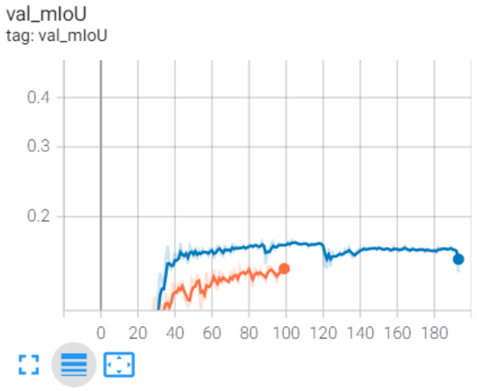
=> After 100 epochs, the training mIoU stopped at ~28% and validation mIoU was ~15%. Upon my inspection, the training loss can decreases more with longer training time, but this speed is too slow. The model hardly learn from the training dataset. Keeping image size unchanged throughout the flow limited the size and learning capacity of the model.

2. Phase 2

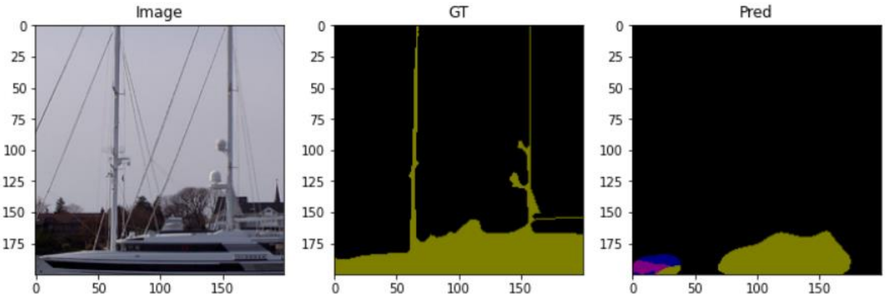
To increases the capacity of the model, the image size must be decreased during training flow, and increased after that for the output to be the same size as input. The model was divided into a downsampling part and an upsampling part, which use UpsamplingBilinear2D to upsample. In addition, the crop_size was increased to 256 to derive more information from training dataset. The downsampling method also allowed the model's batch_size to be 32, for higher accurate learning steps. At this point, the image transformations were also deployed for training input, which included: (1) Random Sharpness Enhance, (2) Random Left-Right Flip. The filter size kept=3 as last phase.

Model: [Conv2d->ReLU->MaxPooling]*5 -> [Conv2d->ReLU->UpsamplingBilinear2D]*5





The model stopped before training completed because of Colab GPU limits. The training mIoU had much improvement, the model started fitting the training dataset more closely, object patterns appeared:

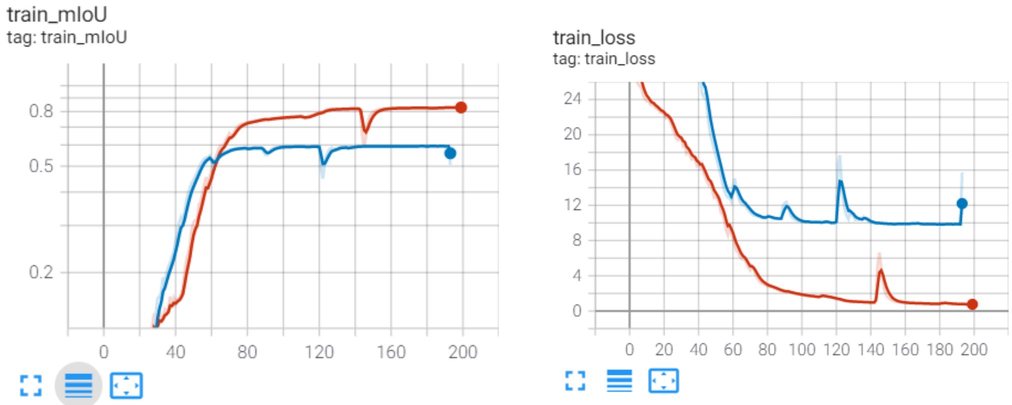


<training result>

3. Phase 3

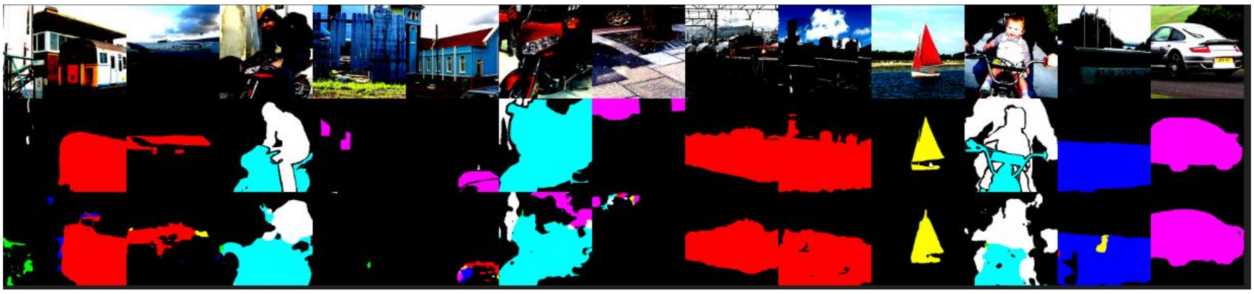
Continue maximizing the training mIoU, the batch size was increased to 50, and number of convolutional block increased to 6-6 encoder-decoder (increase depth). After good results from some trials, I replaced the UpsamplingBilinear2d layers with ConvTranspose2d layers for learnable kernels. Other parameters are the same as last phase.

Channels: 3 -> 30 -> 90 -> 270 -> 810 -> 810*3 -> 810 * 3 -> 810 -> 270 -> 90 -> 30 -> 10



<red=phase3; blue=phase2>

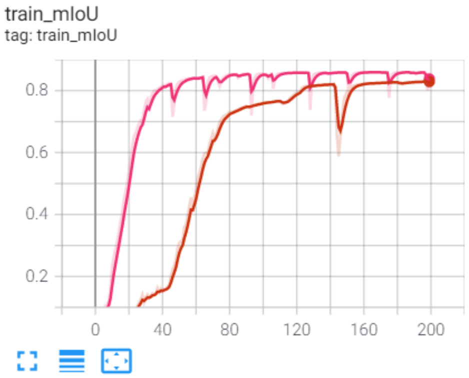
The training loss decreased faster, the training mIoU reached over 80% at the end which is greatly improved. The model started fitting the training data very closely:

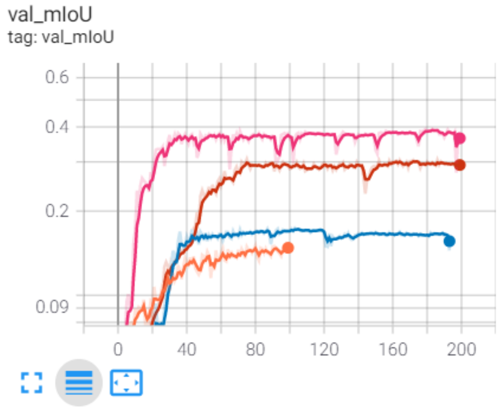


<predicted images from training dataset>

4. Phase 4

Continued improving from training mIoU of ~80%, number of epoch was increased but the training process fluctuate greatly after epoch ~200 and yielded no noticeable result. The model depth and width also seemed maximized, so I turned into improving the dataset. Originally, there are 1000 pairs of image and label, and this is clearly too small for a convolutional neural network to work well. I tried to directly randomly transform the images during training by random rotate and random cropping. However, they costed so much training time (over 120s / epochs). That made me generate images directly on my PC and reupload the dataset [1]. Instead of only center crop, I cropped 4 corner and 1 center crop with size = 256. Besides, I randomly rotate the image with degree <= 30 and center crop again (to eliminate black border). However, 5400 images and labels are too much for my current model (over 100sec/epoch and especially my Colab account started slowing down), so I remove the random rotate, took the 4500 for main processing. Images were then normalized with pixel values mean and standard deviation calculated manually from new dataset [2]. Other parameters were the same as last phase.

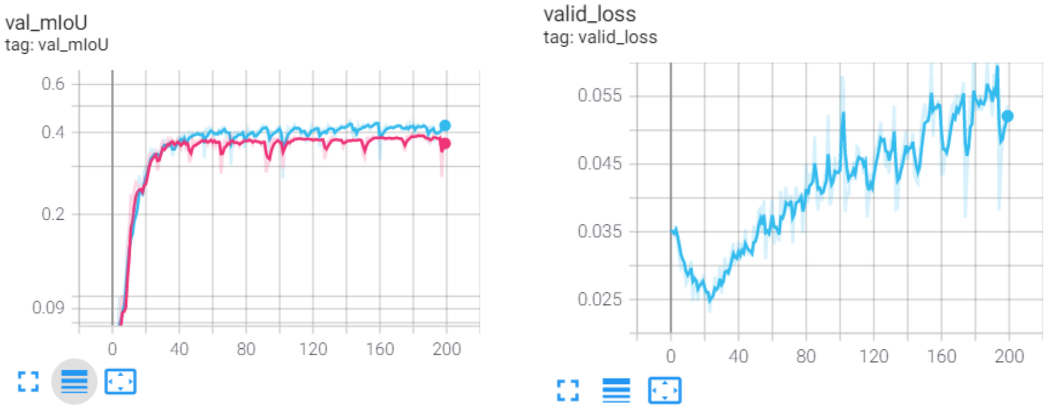




There were slight improvement in training mIoU, and much improvement in the validation loss and validation mIoU in compare with previous phases. This means the increase of data variance did help the model learning and generalizing better.

5. Phase 5

From the improvement in phase 4, regularization methods were deployed to handle the overfitting. The Batnorm2D layers were placed after each Conv2D layer, Dropout were deployed after encoding part and after decoding part with p=0.5. Because of heavy overfitting problem, I started with high weight decay of 1e-2 for Adam optimizer, then gradually decreasing with factor=10, and got the most optimal value at 1e-6. Other parameters are the same as last phase.

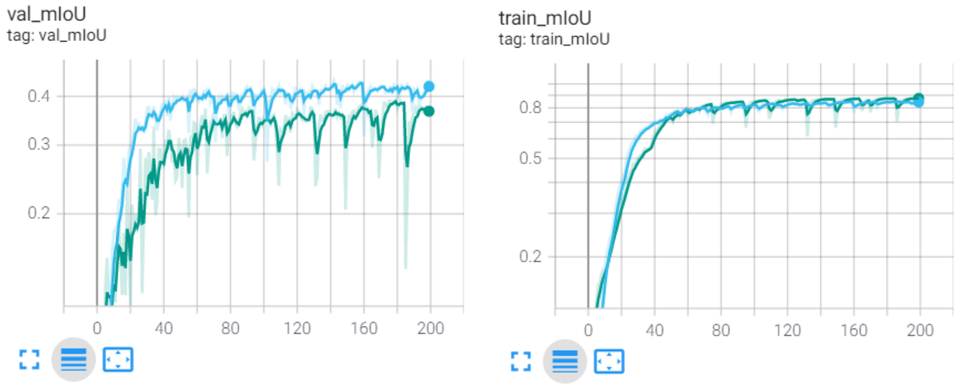


<blue=phase5; red=phase4>

The validation mIoU had slight improvement but overfitting still occurred very early and heavily. I experimented with other dropout probability of 0.2, 0.3, 0.4, 0.7 but none of them yielded better result.

6. **Phase 6**

Followed from model of phase 5 and trying to improve the training mIoU, the model complexity was increased by adding 1 more block of [Conv2D->Batnorm2D->ReLU] for each encoder block and after each TransposeConv2D layer.

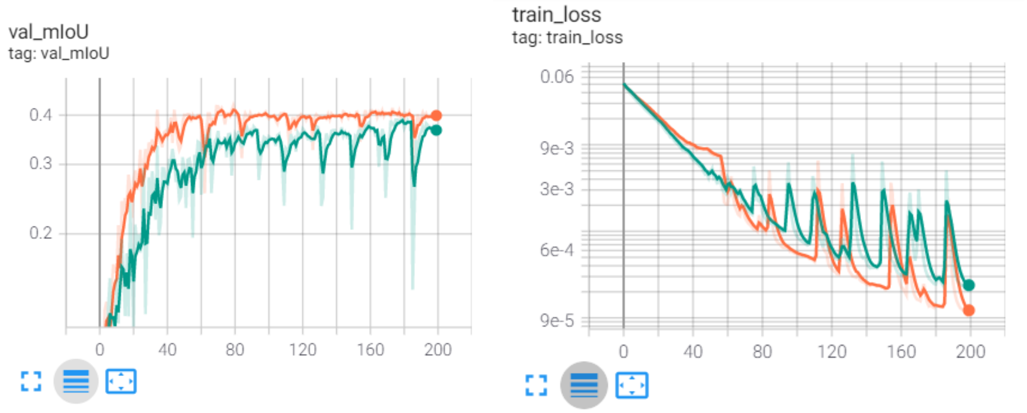


<blue=Phase5; green=Phase6>

Higher complexity did not work as expected. The training mIoU kept constant at ~85% and fluctuate more heavily before before that. After some experiment, I realized that higher weight decay or dropout probability only made the training mIoU fluctuate more but brought no noticeable result for validation loss or mIoU.

7. Phase 6.1

Continue with model from phase 6 but without dropout and weight decay



<orange=phase6.1; green=phase6>

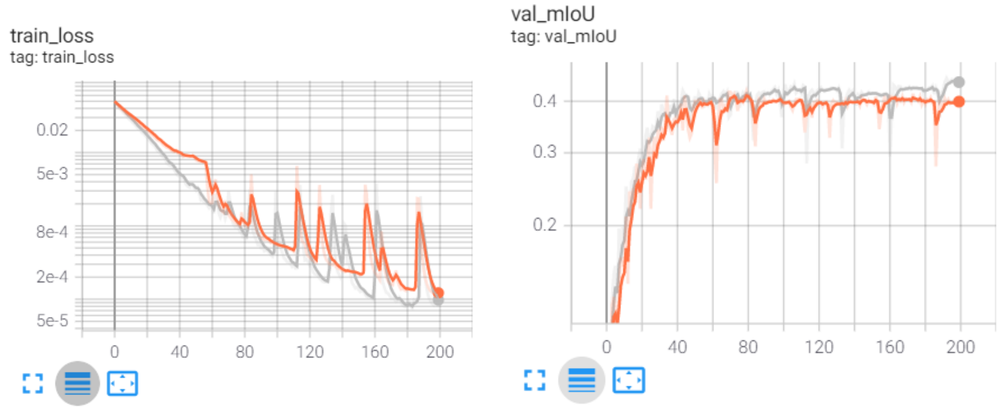
The training mIoU stay the same while validation mIoU and train loss has improvement in compare with last version. It seems like dropout and weight decay only made the problem more serious.

8. Phase 7 and Phase 7.1

Increasing model complexity did not bring improvement for training mIoU (kept at ~85%). From my inspection, more complexity (more layers / deeper model) only made the training process more slow and prevent model from learning, this is a sign of gradient vanishing problem. Therefore, the skip connections were applied to model of phase 6.1 to see if there could be any improvement.

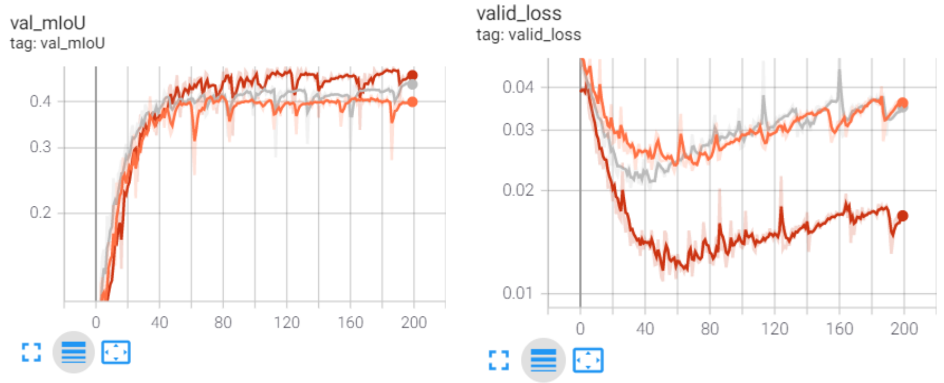
Phase 7 = model 6.1 + skip connection + weigh_decay = 1e-8; Phase 7.1 = model 6.1 + skip connection

Two phases were run simultaneously for experimental purpose:



<gray=Phase7; orange=Phase6.1>

In phase 7, while the training mIoU and validation loss kept nearly the same as phase 6.1, we could see slight improvement in training loss and validation mIoU. Still, the difference was expected to be more than that.



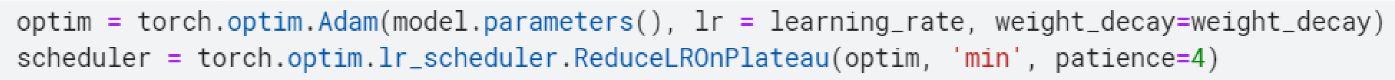
<gray=Phase7; orange=Phase6.1; red=Phase7.1>

We could see a great improvement in validation loss as well as in validation mIoU. The skip connections did helps the model to generalize better, and the result also shown that the weight decay could not help anything here. However, not as expected, the training loss and mIoU could not improve more than 85%.

9. Phase 8 & phase 9 & phase 9.1

In an experimental trial of phase 8, the Resnet structure as encoder part was applied: add a skip connection between input and output of each encoder convolutional block. This was expected to increase data derivation of mode during the training flow. However, the training kept constant after reaching ~86%, there was no noticeable result.

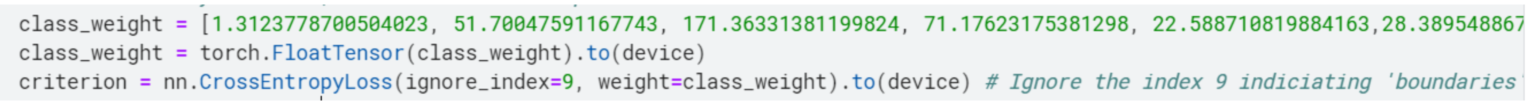
In phase 9, the learning scheduler was deploy for phase 8 model to decrease learning rate as validation loss started decreasing:



```
optim = torch.optim.Adam(model.parameters(), lr = learning_rate, weight_decay=weight_decay)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optim, 'min', patience=4)
```

After training, there was no improvement while the training mIoU stopped after reaching 40%

In phase 9.1, the learning rate scheduler was removed and weighted loss function was deployed since I noticed there was huge gaps among classes (e.g. area of background over 100 times larger than area of bicycle):



```
class_weight = [1.3123778700504023, 51.70047591167743, 171.36331381199824, 71.17623175381298, 22.588710819884163,28.389548867
class_weight = torch.FloatTensor(class_weight).to(device)
criterion = nn.CrossEntropyLoss(ignore_index=9, weight=class_weight).to(device) # Ignore the index 9 indiciating 'boundaries
```

The area of each classes were calculated manually [3], took the inverse value for loss weights.

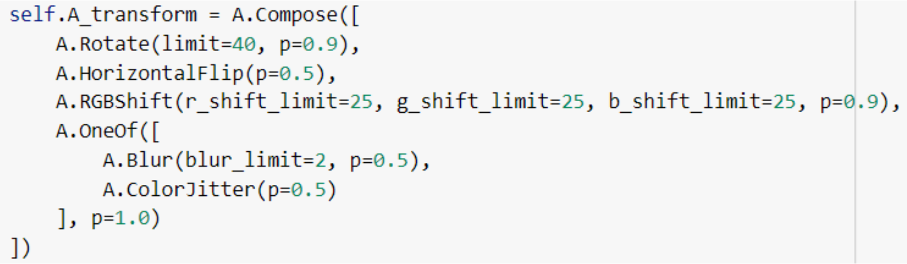
=> Result from training shown no improvement in compare with previous models, while the training time increased greatly, which made the train process incomplete.

10. Phase 7.3

Returned to the best model of phase 7.1, the model size is reduce with expect that the overfitting could be handled: reduce from 6-6 encoder-decoder to 4-4 encoder-decoder blocks structure => The model could not finish training because the large image size at middle costed much time for each epoch (> 136sec / epochs).

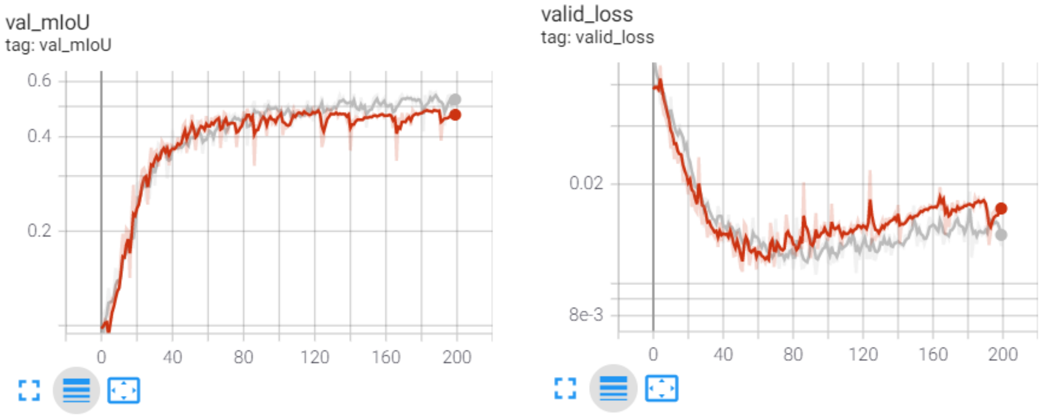
11. Phase 7.4

I founded out from the net a way to rotate the image and label at same time and reflect the border to prevent black pixel. This utilized albumentations library, which also allows horizontal flips and color shifting:



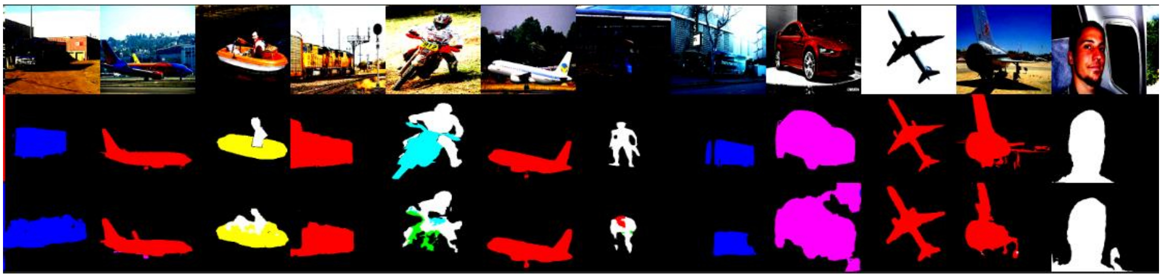
```
self.A_transform = A.Compose([
    A.Rotate(limit=40, p=0.9),
    A.HorizontalFlip(p=0.5),
    A.RGBShift(r_shift_limit=25, g_shift_limit=25, b_shift_limit=25, p=0.9),
    A.OneOf([
        A.Blur(blur_limit=2, p=0.5),
        A.ColorJitter(p=0.5)
    ], p=1.0)
])
```

Above augmentations were applied to the best model of phase 7.1 without dropout or weight decay. The result was slightly improved as the more random input images helped the model to generalize better:



<red=Phase7.1; gray=Phase7.4>

The validation results were close to the ground truths:



12. Phase 7.5 & 7.6

I tried to improve model from phase 7.4 with learning_rate scheduler to decrease after epoch 100 in phase 7.5 or increase the batch size from 40 steps by steps to 60 in phase 7.6 or change the optimizer to RMSprop with momentum=0.9 in phase 7.7 but there was no noticeable improvement.

III. Discussion and Conclusion

1. Overfitting problem

The final model could not handle overfitting problem. This is likely because of the limit of training dataset, which need to be large for a fully convolutional neural network to work well. The result is expected to be greatly improved if a pretrained convolutional neural network is employed as the encoder because they have already trained through thousands of images as from the Imagnet dataset.

2. Regularization methods

There were total 4 regularization techniques deployed during the training process, which included: batch normalization, weight decay (L2 regularization), dropout, and data transformations. Weight decay and dropout took much time to apply in each model because they need parameters and they are dependent to each other. So the weight decay values were adjusted manually for each Dropout probability in each trial. However, both of these methods did not bring any noticeable improvement, and were removed in the final model.

3. Turning the hyperparameters

The two main interested hyperparemeters are: batch size and learning rate. The original learning rate was 1e-3 for the baseline model, which I derived from my model in homework 3. The learning seemed to be stable in most model. Learning rate schedulers were also deployed but, in all trials, they only slow down the learning process and had no improvement on the validation loss or mIoU. About the batch size, the value of 40 for batch size was tested as the best for the last model 7.4.

4. GPU limit on Colab platform

Colab platform has a strict limit for free user. For a new account, there are about 2 training processes with 3 hours each could be finished before the GPU go down. I created 6 accounts and use them in a circle direction to create time for used account recovering. Moreover, running training on an account with data from another account costed more time, so I had to re-upload the dataset each time switching account. In addition, on my inspection, there are more available GPU at night (from 8pm in GMT+7).

5. Final Model and Conclusion

Phase 7.4 has been the best model so far, which achieved about 60% in validation dataset after training and ~64% on test dataset. This model is a combination of basic UNet architecture, image preprocessing with crop_size=256 and normalize by mean and standard deviation, image transformation during training with albumentations library, Batches normalization, and fine-tuned hyperparameters through trials and errors.

Model: [Conv2D->Batchnorm2D->ReLU]*2 -> [[Conv2D->Batchnorm2D->ReLU]*2->MaxPooling2D]*6 -> [ConvTranspose2D->ReLU->Conv2D->Batchnorm2D->ReLU]*6

Channels: 3->10->32->64->128->256->512->1024->512->256->128->64->32->10

IV. Reference

* https://kozodoi.me/python/deep%20learning/pytorch/tutorial/2021/03/08/image-mean-std.html

* https://pytorch.org/vision/stable/transforms.html

* https://github.com/milesial/Pytorch-UNet

* https://medium.com/analytics-vidhya/introduction-to-semantic-image-segmentation-856cda5e5de8

V. Appendix

[1] Code is implemented in file '*data_preprocess.py*'

[2] Code is implemented in file '*mean_std_calculator.py*'

[3] Code is implemented in file '*class_area_calculator.py*'