Deep Learning for Autonomous Driving Spring 2021

Project 2: Multi-task learning for semantics and depth

Group members: Xiao'ao Song(18-747-949) Qi Ma(20-960-225)

Problem 1: Joint architecture

1.1 Hyper-parameters tuning:

a) Try different optimizer and learning rate:

Problem description: The baseline performance will be poor with sub-optimal default values for some of the hyperparameters. How does logarithmically changing the learning rate affect the learning? You can try three different values for each optimizer as an indicative bracket.

We tried different combinations of optimizer and learning rate as follows. The best settings are highlighted.

table1: Adam optimizer

optimizer \ lr rate	**0.0001**	0.001	0.01	0.1
Total train loss	**0.2716**	0.2463	0.6163	0.7411
Total val loss	**0.4024**	0.4547	0.8613	1.867
Total val metric	**43.07**	38.612	2.072	-33.733
Name	**G8_0508-1213_res_b4_adam_0.0001_c5767/**	G8_0425-1336_res_b4_adam_0.001_cecff	G8_0426-0942_res_b4_adam_0.01_8d405	G8_0427-0622_res_b4_adam_0.1_73d90

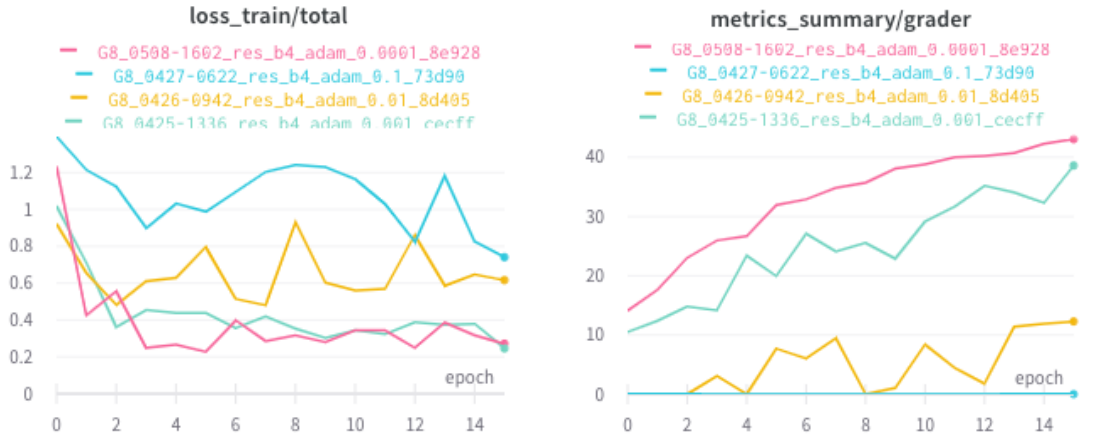

Figure1: Adam optimizer with different lr (0.0001: pink),(0.001:light green), (0.01:yellow),(0.1:light blue)

We first tried using Adam optimizer with different learning rates: 0.0001, 0.001, 0.01, 0.1. As we increase the learning rate, the total loss starts to increase and the total score starts to decrease. The best result is achieved by using learning rate **0.0001**. With a smaller learning rate, the optimizer might converge slowly or might not converge within the given running epochs. A larger learning rate might converge fast,

however, it may also oscillate a lot in the optimizing space. From the above figure, we can see that using learning rate 0.1, the loss curve oscillate a lot even at epoch #13 which is even close to the end of our training.

Additionally the table shows two interesting facts, one of which is when using lr 0.0001 we have lower train loss but however with less validation grader metric, it implies that too low learning rate will result in overfitting. Secondly, the negative grader metrics-summary shows very poor performance of task, according to metric function, we can see that: metric summary = iou - 50 +50 - si-logrmse = iou - si-logrmse, where metrics grader = max(iou - 50, 0) +max(50 + silogrmse, 0).

Table2: SGD optimizer

optimizer \ lr rate	0.01	**0.05**	0.1	1
Total loss	0.4298	**0.3944**	0.4079	fail
Total mertic	40.555	**44.388**	43.432	fail
Name	G8_0507-2021_res_b4_sgd_0.01_5cc93	**G8_0508-0623_res_b4_sgd_0.05_a4150/**	G8_0427-1308_res_b4_sgd_0.1_adcf1	G8_0428-1105_res_b4_sgd_1_2464c

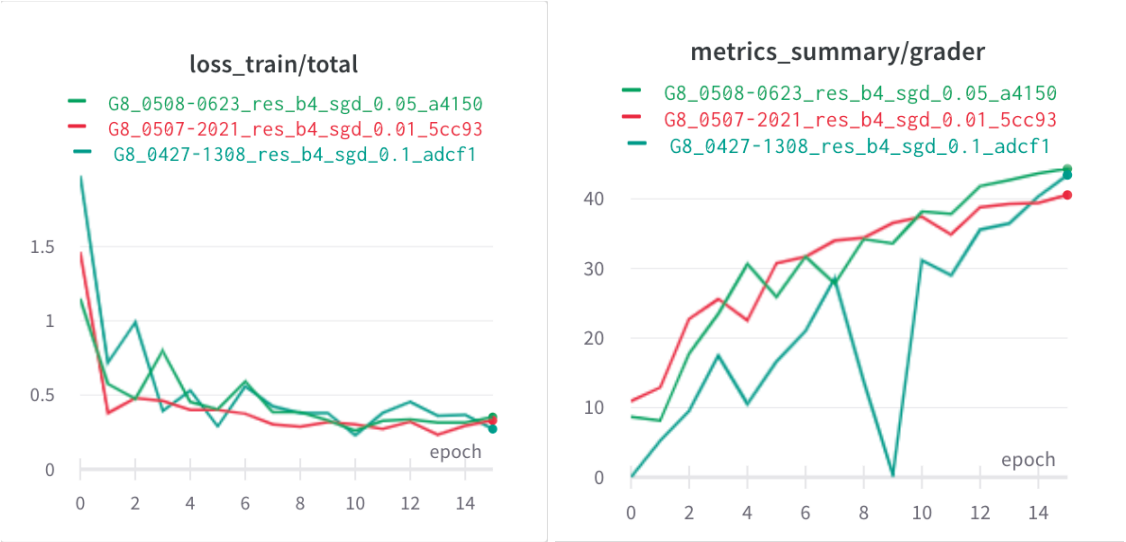

Figure2: SGD optimizer with different lr (0.01:red) ,(0.05green),(0.1: dark green)

We then tried using SGD optimizer with different learning rates: 0.01, 0.05, 0.1, 1. As we increase the learning rate, the total loss starts to decrease first and then increase, and the total score starts to decrease first and then increase. The best result is achieved by using a learning rate **0.05**. From the left figure, the smallest learning rate 0.01 yields the smoothest curve (colored in red), and larger learning rate yields a more oscillated curve. This is consistent with our observation above when using Adam optimizer.

Moreover, the failure in table means when running with learning rate 1 will result in NaN loss so the AWS always crashes.

Table3: Our parameter configuration

Optimizer	Learning rate	Grader Score	Run name
SGD	0.05	44.388	G8_0508-0623_res_b4_sgd_0.05_a4150

b) Try different batch size:

Problem description: Is a larger batch size preferable over a smaller one for our tasks?

Our experiments and result are attached as follow:

Table4L Different batch size

training samples	batch size	epochs	Total loss	Total mertic
20000	2	8	0.429	30.19
20000	4	16	0.3564	38.548
20000	**8**	**32**	**0.2732**	**42.599**
20000	16	64	0.3239	33.246

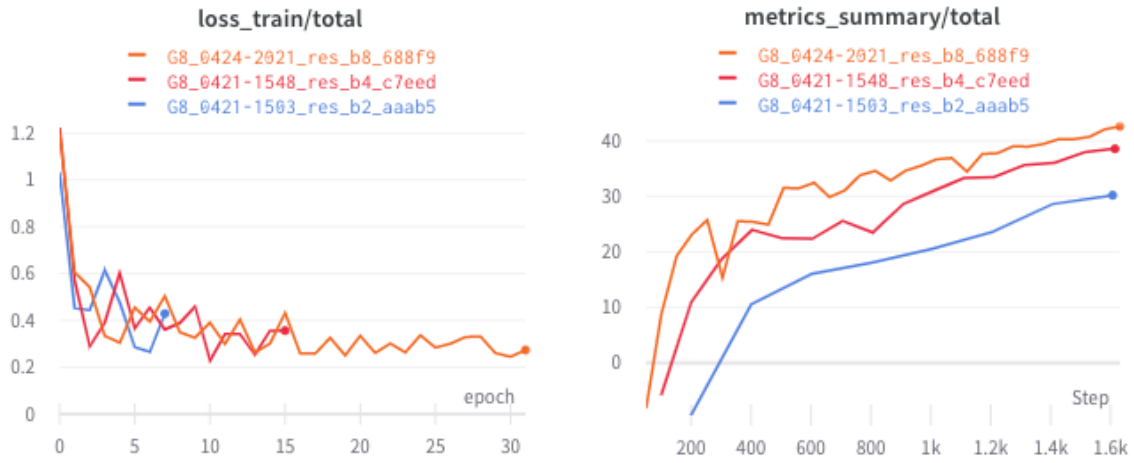

Figure3: Different batch comparison (left along epoch, right along step)
(batch size 2: blue),(batch size 4: red),(batch size 8:orange)

We tried different batch sizes: 2, 4, 8. As we increase the batch size, the total loss starts to decrease and the total score starts to increase. The best result is achieved by using batch size: **8**.

By default, SGD is the optimizer we are using for the experiments. In SGD, the gradient of the loss function is computed over the entire batch. If we use a very small batch size, the gradient will really be all over the place and pretty random, because learning will be image by image. With a large batch size, we get more "accurate" gradients because we are optimizing the loss simultaneously over a larger set of images.

Although we get more frequent updates when using a smaller batch size, those updates aren't necessarily better. However, the main disadvantage of using large batch size is that the parameters are only updated after each batch and this might be extremely time-consuming. Therefore, it is always a trade off between small and large batch size.

After comparison, the best result we achieved from our experiments attached as follow:

Table5: Our configuration

Batch size	Epochs	Grader Score	Run name
8	32	42.599	G8_0424-2021_res_b8_688f9

c) **Try different task weights:**

Problem description: Can you find proper loss weights for the employed tasks to improve their joint performance?

Our experiments and result are attached as follow (by default, the SGD optimizer is used):

Table6: Different loss weight:

weight depth	weight semseg	grader depth	grader semseg	grader total
0.6	0.4	25.62	69.683	44.063
0.5	0.5	25.865	70.203	44.338
0.4	0.6	26.351	70.804	44.453
0.35	**0.65**	**26.002**	**70.965**	**44.693**

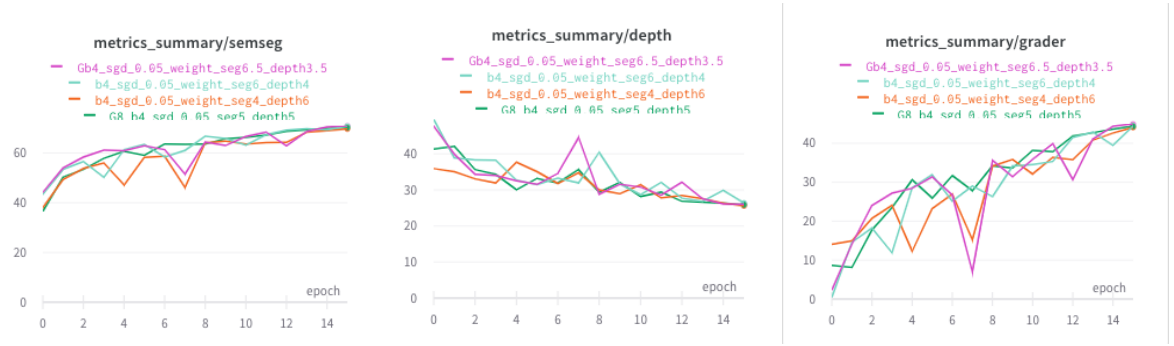

Figure4: Different task weight comparison (depth0.5, semseg0.5:green), (depth0.6,semseg0.4:orange), (depth0.4,semseg0.6:light blue),(depth0.35,semseg6.5:purple)

After comparison, the best result we achieved from our experiments attached as follow:

Table7: Our configuration:

weight depth	weight semseg	grader depth	grader semseg	grader total
0.35	**0.65**	**26.002**	**70.965**	**44.693**

We have tried four sets of weights. From the result we can see as we put more emphasis on the weight of **semseg** and less emphasis on the weight of **depth**, both the grader score of **semseg** and **depth** reflected an increasing trend. The increasing grader score of **semseg** indicates the model is being more capable for this sub-task whereas the increasing grader score of **depth** indicates the model is being more incapable for predicting depth information. This is consistent with our experimental expectations.

The only outlier occurred when putting 35% weights on the depth part and we believe this is due to experimental randomness. Therefore, we have chosen the weight set **{depth:0.4; semseg:0.6}** for the remaining experiments. This will also avoid one task overwhelms the other one.

1.2 Hardcoded hyperparameters:

The results of our experiments are attached as follow

Table8: Pretrain or dilated convolution:

Pretrained weights	Dilated convolutions	Semseg	Depth	Grader Score	Run name
Off	Off	70.804	26.351	44.453	G8_0509-0525_res_b4_sgd_0.05_weight_64_2625d/
On	Off	72.152	25.399	46.753	G8_0510-0552_res_b4_sgd_0.05_weight_64_ec2c1/
On	On	79.263	21.745	57.518	G8_0510-2148_res_b4_sgd_0.05_weight_64_dilate_pretrain_81c4d/

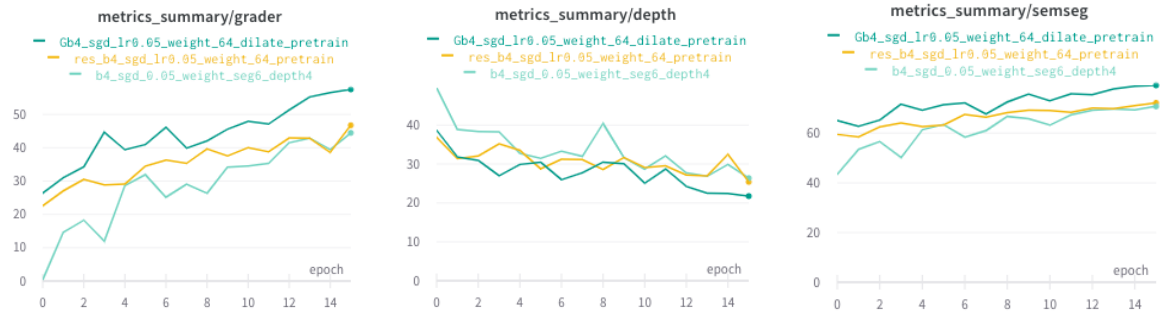

Figure5: Comparison of (no pretrain, no dilated conv:Light blue),
(with pretrain, no dilated conv:Yellow),
(with pretrain, no dilated conv:green)

a) Initialization with ImageNet weights:

Problem description: Can you verify whether the encoder network is initialized with weights of a model trained on the ImageNet classification task?

Initially, using the pretrained weights is toggled off. After toggling it on, the network will start from these pretrained weights instead of initialized randomly from some distribution. From the above result we can see that the performance has been improved a lot after using pre-trained weights.

b) Dilated convolutions

Problem description: Set dilation flags to (False, False, True) and train the model. Does the performance improve? If so, why?

By setting the dilation flags to (False, False, True), the model is trained with dilated convolutions and the performance has been improved a lot again by comparison to use the pretrained weights only.
Dilated convolutions have generally improved performance. And the more important point is that the

architecture is based on the fact that dilated convolutions support exponential expansion of the receptive field without loss of resolution or coverage. Therefore, it allows one to have a larger receptive field with the same computation and memory costs while also preserving resolution. Meanwhile, pooling and strided convolutions are similar concepts but both reduce the resolution.

The pretrained weights and dilated convolutions are both toggled on for the rest of our experiments.

1.3 ASPP and skip connections:

Table9: Improvement of ASPP and skip connections

Model	Semseg	Depth	Grader Score	Run name
Pretrained + Dilated convolutions	79.263	21.745	57.518	Gb4_sgd_lr0.05_weight_64_dilate_pretrain
ASPP and Skip Connection	80.936	20.655	60.281	res_b4_sgd_0.05_weight_64_dilate_pretrain_aspp_skip

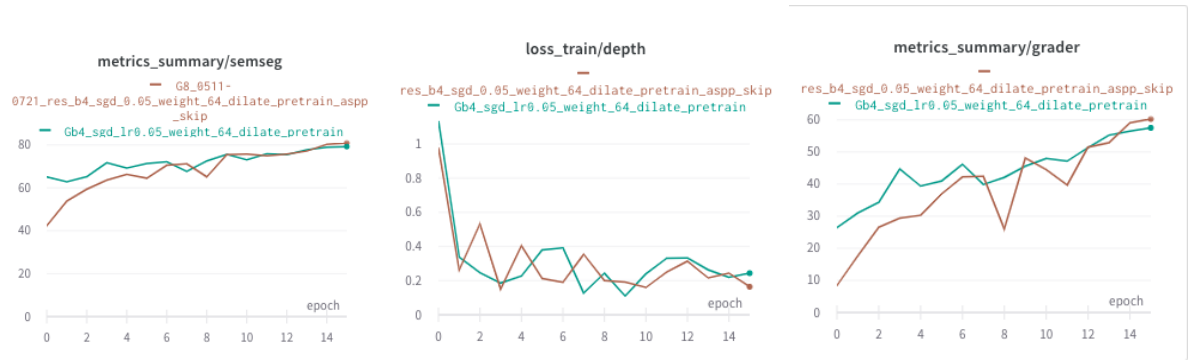

Figure6: Comparison of Semseg | Depth | Grader Score of (Pretrained + Dilated convolutions:green) ,(ASPP and Skip Connection: red)

Problem description: How does the model performance change with ASPP and skip connections functioning?

Spatial pyramid pooling can capture rich contextual information by pooling features at different resolutions. In order to do so, our model DeepLabv3_plus applies several parallel atrous convolutions with different rates (called Atrous Spatial Pyramid Pooling, or ASPP).

There was some information that was captured in the initial layers and was required for reconstruction during the up-sampling layer. If we would not have used the skip architecture that information would have been lost (or turned too abstract for it to be used further). So the low-level information that we had in the primary layers can be fed explicitly to the later layers using the skip architecture.

By adopting ASPP and skip connections, the grader score improved a lot from 57.518 to 60.281.

Problem 2: Branched architecture

Problem description: How does it compare to the joint architecture both in terms of performance but also w.r.t. the model size and the required computations?

We successfully implement the code with branched architecture. From following table the branched architecture brings performance from **60.281** to **66.467.** In Total we make three changes to our code: including config.py, helpers.py, and model_branched.py in mtl/model.

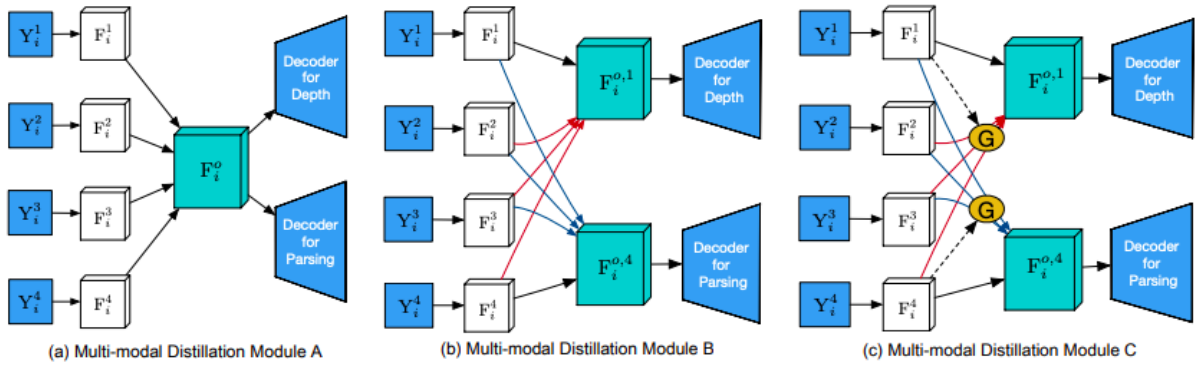

Figure7: Different multi-model Distillation (from PAD-Net)

Basically the model deeplabv3p adopts the idea of Module A where Model-Branched fuse the information like module B. Compared to deeplabv3p the branched model has more capacity to balance the prediction of two tasks instead of mixing together like deeplabv3p. Using independent ASPP and Decoder will improve the task and path architecture helps pass messages between different predictions.
However, adding more component in the network also means larger model size and more computations requirements, as the table below shows:

.

Table10: Model size and computation:

Model	Deeplabv3p	Branched_model
Computation time (16 Epoch)	6h 46m 28s	9h 26m 50s
Model .ckpt size(MB)	204.M	245.4

From this table we see that by adding one more decoder and ASPP component lead to 50M increase in model size and 3 more hour training time. In order to get the improvement it costs a lot of resources.

Table11: Improvement of Branched_architecture

Model	Semseg	Depth	Grader Score	Run name
Joint	80.936	20.655	60.281	res_b4_sgd_0.05_weight_64_dilate_pretrain_aspp_skip

Branched	84.646	18.179	66.467	G8_0424-2021_res_b8_688f9

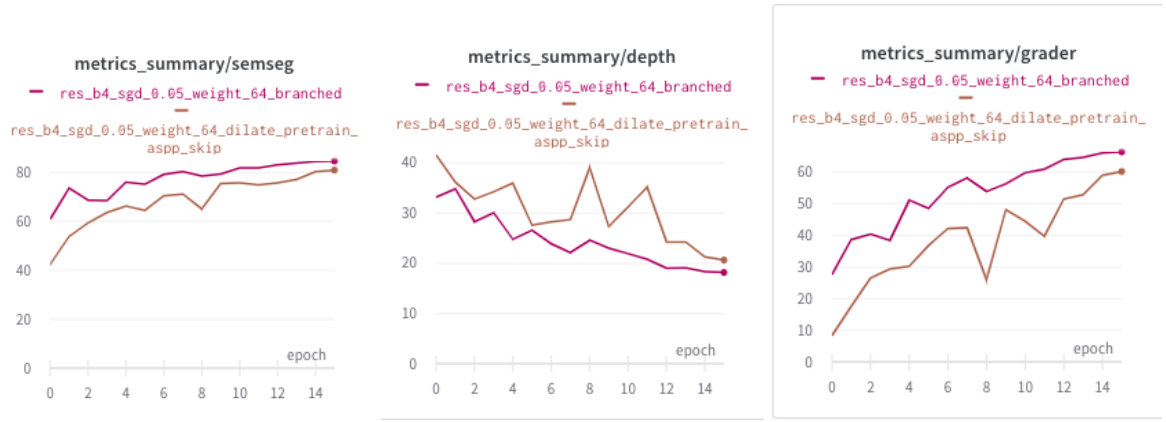

Figure8: Comparison of Semseg | Depth | Grader Score of (Joint:pink) ,(branched: purple)

As the graph shows the branched network shows more smoothness in training, that is because by using the task-specific distillation feature maps, the network can preserve more information for each individual task and is able to facilitate smooth convergence.

Problem 3: Task distillation

Problem description: How does the distillation procedure compare to the branched architecture in the previous problem?

Table12: Model size and computation:

Model	Deeplabv3p	Branched_model	Branched with task distillation
Computation time (16 Epoch)	6h 46m 28s	9h 26m 50s	11h 59m 54s + 3h1m15s = 15h1m9s
Model .ckpt size(MB)	204.M	245.4M	254.5M

The time cost of distillation has two parts since it crashed once. As you can see, the training of the distillation network is really expensive compared to Deeplabv3p. 15 hours training and also have more possibility for crashing. Due to the time limitation we do not improve the distillation network comprehensively and it results in almost the same performance with branched_network.

Table13:Comparison of Branched_architecture

Model	Semseg	Depth	Grader Score	Run name
Branched	84.646	18.179	66.467	G8_0424-2021_res_b8_688f9
Branched with task distillation	84.621	18.778	65.844	G8_0424-2021_res_b8_688f9

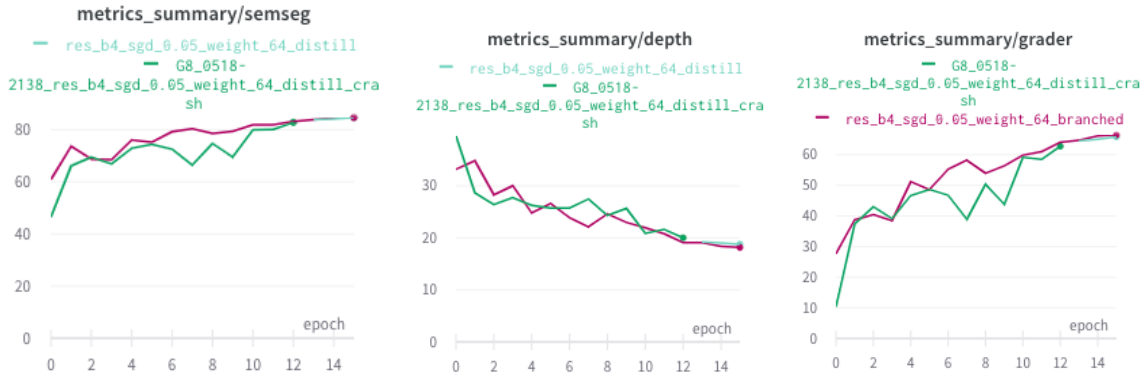

Figure9: Comparison of Semseg | Depth | Grader Score of (branched:purple) ,(branched with task distillation: green)

As shown in Figure7, the distillation network with self-attention have same mechanism in PAD-net module C.In principle with a self-attention module helps to distinguish good and bad passed information flow which is not always useful, the attention can act as a gate function to control the flow, in other words to make the network automatically learn to focus or to ignore information from other features.

During the joint learning, the intermediate tasks not only act as supervision for learning more robust deep representations but also provide rich multi-modal information for improving the final tasks.