**不懂的函数**

ScaleAndShiftInvariantLoss

oss\_seg\_penality = (

loss\_seg\_new - self.loss\_seg\_min

) \* self.p.loss\_seg\_penality\_factor

参数alpha和scale

1. Which model to be used?

[https://github.com/pprp/timm#models](https://github.com/pprp/timm" \l "models)

not essentials as long as it converges in the course of training

只要在训练过程中收敛converges，就不是要领

https://huggingface.co/models

<https://huggingface.co/models?sort=downloads&search=vit>

try vit series from google first

eg. vit\_base\_patch16\_384

ps. By using timm.create\_model, we could call and use a specific model without keyin their source codes. https://timm.fast.ai/models

Should we try an other model? In principle, yes.

2. supervised vs unsupervised

Because depth map is more “continuous” than “discrete”, the loss function must quantify the “incorrectness”. Segment map is more “discrete” than “continuous”, the loss function needs to classify rather than to quantify.

Thus, to quantify depth map loss, we will use “mse” while to classify segmentation loss, will use “cross entropy”.

Because of the need to label the training material and the need to adjust the weighting and to quantify the depth map loss, we will use “supervised training”

There is an intrinsic incompatibilities between “discrete” and “continuous”. Their gradient losses are very different and their relative weights are very difficult to quantity. I could not find any reference or guideline to define their relative weightings.

3. Convergence

I noted that there are at least two different stages of convergences, namely: coarse and fine convergence

Coarse Convergence

During the early stage(usually epoch less than 10), the convergence tends to be fast and unsteady.

To ensure a “correct” convergence, we need to ensure that

a) the weight applied on loss\_segmentation must be large enough so that the predicted segmentation must be of good quality and without noises.

b) the weight applied on loss\_in and loss\_out must be normalized. To normalize it, I use “ScaleAndShiftInvariantLoss”. This method is not very accurate and will create errors during fine convergence stage

As stated previously, there is no well-received way to quantify the weighting between loos\_segmentation, loss\_in and loss\_out.

I propose the following formula to handle the weightings:

loss = loss\_seg + loss\_in + loss\_out

loss = loss\_seg + loss\_depth # + loss\_smoothness + loss\_ssim

fx = self.p.loss\_coarse\_threshold\_factor / np.sqrt(max(epoch \*

self.p.loss\_ratio\_out\_attenuation\_factor, 1))

fx\_mininum = 0.5 # increase the value to reduce noise; decrease this value to get more depth details

fx = max(fx, fx\_mininum)

print("....................fx=", fx)

return (

loss\_seg

\* (1 + loss\_depth / loss \* fx),

loss\_seg, loss\_ratio\_out

)

Based on my experience, self.p.loss\_coarse\_threshold\_factor should not be too big. The smaller the self.p.loss\_coarse\_threshold\_factor, the better the ultimate weighting of loss\_seg. However, if the self.p.loss\_coarse\_threshold\_factor is too small, you will lose all the details that a predicted depth map should have.

All the loss parameters used in Coarse convergence must support precise and steady back-propagation. You might noted that there is no loss parameter responsible to the estimation of losses due to inaccurate (absolute) depth estimation. As yet, I could not find any loss parameter that could provide an covariance invariant loss estimation on absolute depth. To tackle this, I decided to make validation loss function differ from training loss function. The rationale is pretty straightforward. Validation loss function does not shoulder the burden to support precise back-propagation.

However, there is still another outstanding issue to be resolved. Without a proper covariance invariant treatment over the training samples and the validation samples, the loss estimation is still not accurate enough to support a steady and meaningful convergence. I propose another method to tack this difficulty by assuming that

if the predicted and ground truth result becomes very very close to each other, then there will have no genuine needs to do covariance invariant treatment on samples.

To do so, I propose another function

def get\_loss\_depth(self, epoch, output\_depths, Y\_depths, fx=0.1, epoch\_threshold=10):

if epoch < 1 and b\_epoch\_0\_pure\_segementation:

return nn.MSELoss()(output\_depths, Y\_depths) \* 0

if epoch < epoch\_threshold:

return self.loss\_depth(output\_depths, Y\_depths)

else:

return nn.MSELoss()(output\_depths, Y\_depths) \* fx # if fx is very small => loss\_depth is always zero

when epoch =0 , I set the loss\_depth to 0. I want the convergence of depth map to start with segmentation map as the first guess.

self.loss\_depth is defined in config.json. The default value is ScaleAndShiftInvariantLoss

(alpha = 0.7)

The value of alpha will determine the gradient loss wt. the higher the alpha, the higher the impact of gradient loss on the loss loss

You could play round with alpha to decide what is the best alpha you want.

"ssi": ScaleAndShiftInvariantLoss(),

"ssi03": ScaleAndShiftInvariantLoss(0.3),

"ssi04": ScaleAndShiftInvariantLoss(0.4),

"ssi06": ScaleAndShiftInvariantLoss(0.6),

"ssi07": ScaleAndShiftInvariantLoss(0.7),

defined in j\_utils.py

Fine Convergence

Here come the concept of fine convergence. All the loss parameters used for fine convergence(LPFC) will have negligible impact on total loss when the impact of covariance invariant is still very significant(coarse convergence stage). To do so, we will not include any loss parameters used for fine convergence to estimate training loss.

based on my experience self.p.loss\_fine\_threshold\_factor should be less than 1. The bigger the value, the more the details on depthmap. However, if it is too big, the weight of loss\_seg will drop and the depth map will become very noisy.

loss\_smoothness: estimate the smoothness loss

loss\_ssim: estimate the similarity loss

loss\_mse: estimate the absolute depth value loss

As stated previously, all these loss parameter does not support back-propagation well and will not provide any precise loss estimation during the coarse convergence stage.

4. Monotonic Decrease of loss\_seg

I don’t have any good mean to ensure a monotonic decrease of loss\_seg. Any attempt to do so will end up create a damping which is too big that the convergence stopped after less than 10 epochs

Instead, I propose another formula:

if b\_valid\_use\_loss\_seg\_penality:

if self.loss\_seg\_min is not None and self.loss\_seg\_min < loss\_seg\_new:

loss\_seg\_penality = (

loss\_seg\_new - self.loss\_seg\_min

) \* self.p.loss\_seg\_penality\_factor

print("loss\_seg\_penality=", loss\_seg\_penality.item())

loss\_fines.append(loss\_seg\_penality)

This formula penalize any deviation of monotonic decrease of loss\_seg.

self.p.loss\_seg\_penality\_factor will create damping on convergence and is therefore cannot be too big.

5. Wtss

# self.names = [

# 0"depth\_datum",

# 1"loss\_seg\_penality\_factor", # too small => segmentation wt decrease; too big ==> cannot converge smoothly

# 2"loss\_fine\_threshold\_factor", # too small => no improvement in smoothness, depth accurarcy and similarities; too big => segmentatin wt decreases

# 3"loss\_coarse\_threshod\_factor", # too small => segmentatin wt increases, cannot converge smoothly; too big => segmentatin wt decreases, cannot converge smoothly

# 4"loss\_ratio\_out\_factor", # too big (>10) => cannot determine in vs out, too small(<2) => cannot converge

#

# 5"loss\_ratio\_out\_attenuation\_factor" # too big => segment wt will outweight others when epoch increases, too small => veritical striation appears when epoch increases

# 6"loss\_segmentation\_factor", # segmentation wt

# 7"loss\_mse\_factor", # mse wt

# 8"loss\_depth\_in\_factor", # depth in wt

# 9"loss\_depth\_out\_factor", # depth out wt

# 10"loss\_smooth\_factor", # edge wt ;too small => smoothness decrease

# 11"loss\_ssim\_factor", # similarities wt

# ]

wtss = Wtss()

# 0 1 2 3 4 5 6 7 8 9 10 11

wtss.add(0.7, 3.0, 0.4, 1.0, 0.1, 4.0, 0.5, 0.2, 0.5, 0.5, 0.5, 0.5) # good

Wtss is a weight helper class, so that you could define all these values easier

6. Number samples used for each epoch

My approach is different from <https://github.com/antocad/FocusOnDepth>. They used all the samples available and repeatedly using them for each epoch. I tend to think that repeatedly using the sample of samples for all the epochs will result in over-fitting. It is my pure speculation.

Anyway, I changed it to randomly extract a pre-defined num\_samples to carry out training for each epoch.

num\_samples = 30

=> randomly extract 30 samples from each dataset

Assuming that

"splits": {

"split\_train": 0.6,

"split\_val": 0.2,

"split\_test": 0.2

}, “num of datasets” : {3}

=> we will use 30 \* 3 \* 0.6 samples for training and 30 \* 3 \* 0.2 for validation.

Shall we use a bigger value for num\_samples? You should try.

I did not use any samples from the datasets for testing. Why? No special reason. But I incline to believe that using sample outside the datasets will be more meaningful on studying the inference accuracy.

7. More Datasets

Right now there are only three different set of datasets. We should add more but I am too lazy to do so. I can’t find any information of the format requirements of datasets.

The folder structure is:

name\_dataset

- segmentations

- images

- depths

nyuv2 (640x480), inria (960x540), posetrack(1280x720)

- segmentation: png with only two colors RGBA = (150, 5, 61, 255) or (0, 0, 0, 255)

- images: jpg three colors

- depths: jpg three same colors R=G=B

If you want to add more, please note that

1) the image size (w, h) is very flexible.

2) segmentation map is png rather than jpg because of png is lossless format. I don’t know why the non black color is RGBA = (150, 5, 61, 255). But to play safe, better keep it this way

3) depths map is jpg. Although it is a gray image, the file format is still 3 colors. (shape=(h,w,3))

8. Epoch vs Noise vs Details

When epoch number increases, the details added to depth map could either be correct or incorrect. Adding correct details will improve the quality of depth map while adding incorrect details will do the opposite. There is no good way to prevent the latter from happening.

Only viable means used right now are:

1) decrease the value of loss\_failed\_count\_threshold( in function Trainer.train) to say 3

2) increase the value of fx\_mininum(in function Trainer.j\_loss) to say 0.7