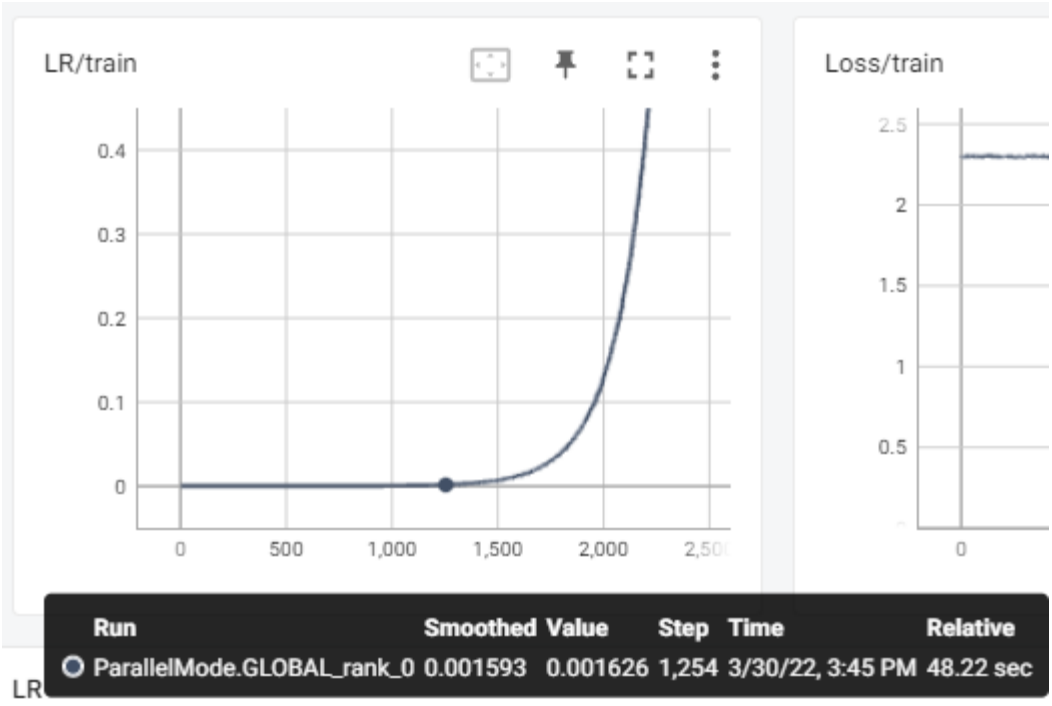
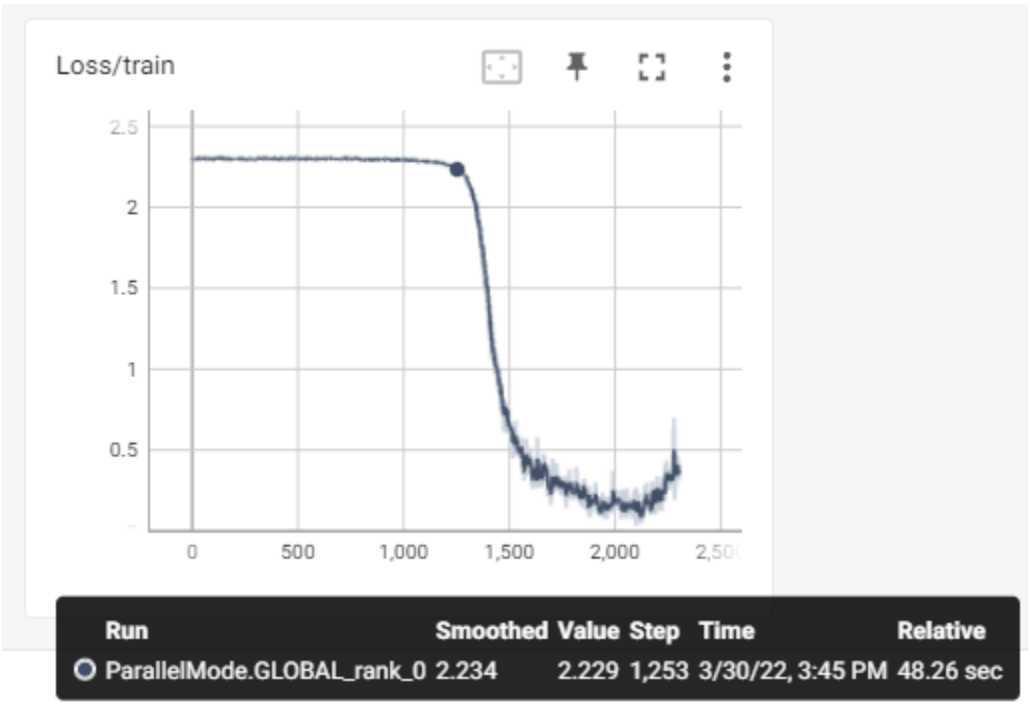
a) Find the best LR to start with.

By executing the unchanged notebook, we would get the following graphs.

The learning rate to time:



The loss to time:



The loss follows the exponential increase of learning rate, that is the loss remains unchanged at first, then starts to go down, and explodes at the end (a few more epochs and the loss would sky rocket).

Thus, following the theory, we would choose the learning rate when the loss just starts to decrease. That is around the 1250 steps, and the learning rate for that would be 0.0016.

SGD

1) experiment 1

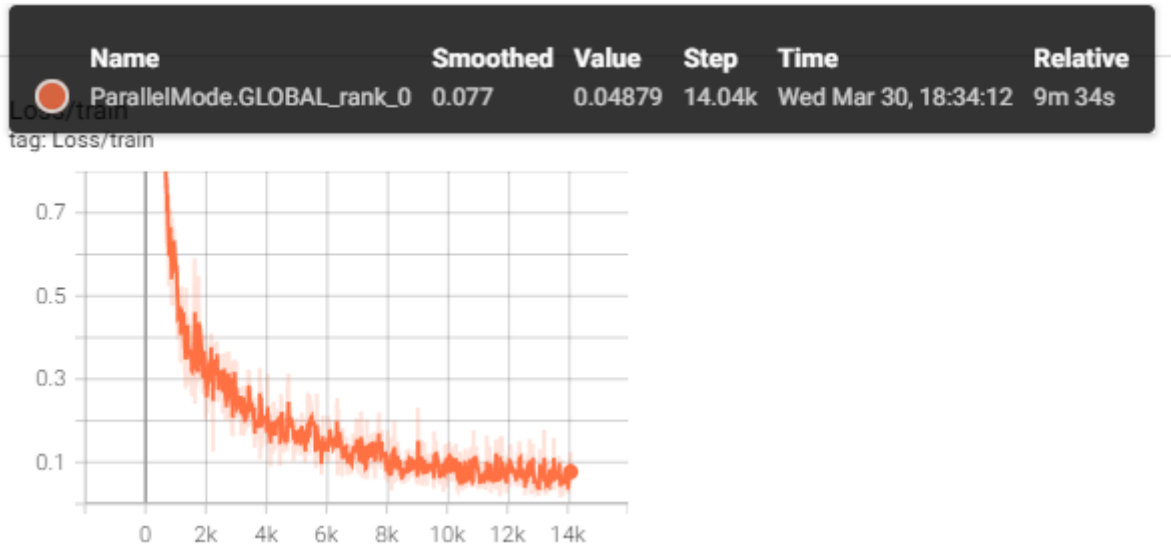
Optimizer: SGD

Learning rate : 0.0016

Scheduling method: No scheduling

Epoch: 30

The result as below:



It almost follows the result from the exponential increasing learning rate scheduling's decreasing part exactly because we used the exact learning rate at the decreasing point.

However, since this time, the learning rate is fixed, it would not result in the explode of the loss and would slowly converge as the epochs goes on.

This is mostly the chosen method I use personally when I was training neural networks , it provides a simple structure and satisfactory result.

The final loss was around:0.04879

2) experiment 2

Optimizer: SGD

Learning rate : 0.0016

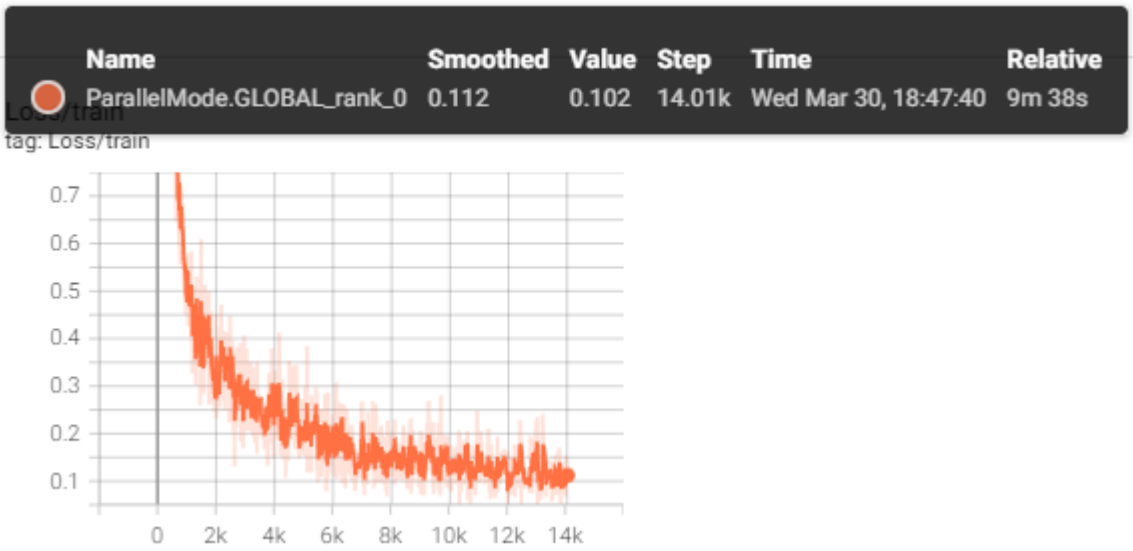
Scheduling method: multiplicativeLR

Epoch: 30

Lambda function : lmbda = lambda batch : 0.95

Scheduler = lr_scheduler = torch.optim.lr_scheduler.MultiplicativeLR(optimizer,lmbda)

The result as below :



This is a simple approach where the learning rate multiply by a certain portion of itself every epoch. However, this approach does not seem to yield better result than the default no scheduler approach. Firstly, no scheduler (experiment 1) has a more stable variance between losses and the final loss is at 0.048 which is much lower than this case in a 30 epoch approach.

The final loss was around: 0.102

3) experiment 3

Optimizer: SGD

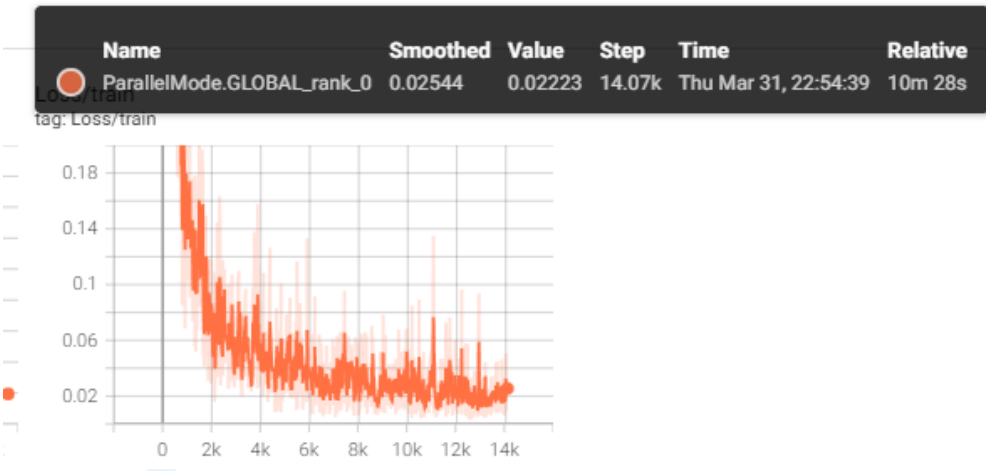
Learning rate : 0.0016

Scheduling method: CyclicLR ()

Epoch: 30

Scheduler: lr_scheduler = torch.optim.lr_scheduler.CyclicLR(optimizer,base_lr=0.01,max_lr=0.1)

The result as below :



This policy sets learning rate between the base and max boundaries using a constant frequency. Which yields a slightly better result when compared to the no scheduler approach.

The final loss was around: 0.02223

4) experiment 4

Optimizer: SGD

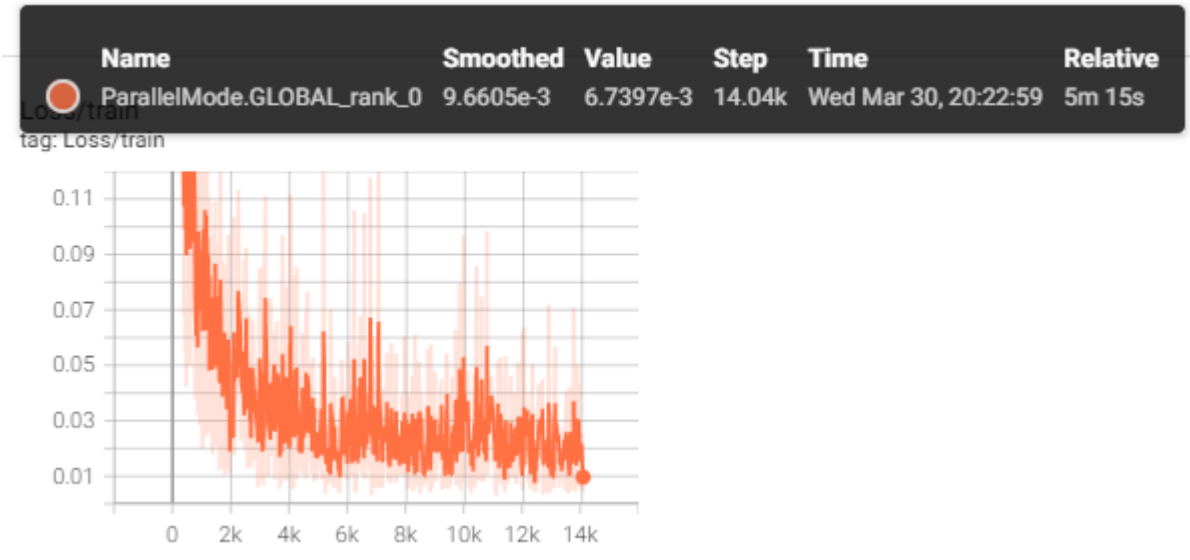
Learning rate : 0.0016

Scheduling method: OneCycleLR(max_lr = 0.5)

Scheduler : lr_scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer,max_lr=0.5,steps_per_epoch = len(train_dataloader),epochs = 30)

Epoch: 30

The result as below :



In short, one cycle lr anneals the learning from the initial lr to a maximum learning rate and back to a much lower learning rate, even lower than the initial given lr.

The result was surprisingly good, the lowest among all tested schedulers. Although the loss seems to be unsteady compared to the previous two approach, which much higher and lower bounds. Which means it is possible for us to get a final loss with is much higher than the previous two too.

The final loss was around:0.0067

Experiment 4.1

Optimizer: SGD

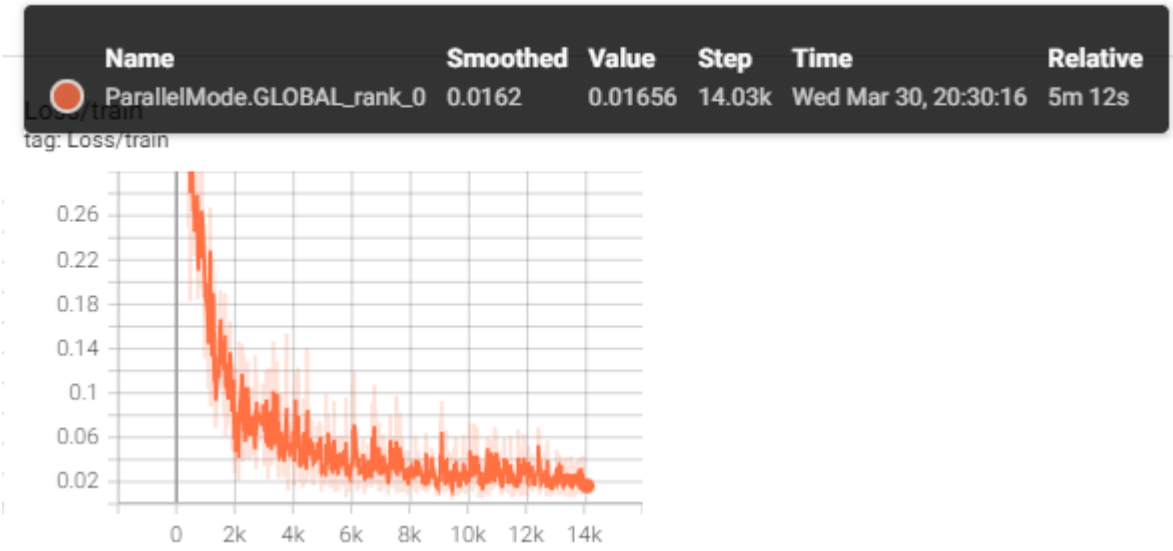
Learning rate : 0.0016

Scheduling method: OneCycleLR(max_lr = 0.1)

Scheduler : lr_scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer,max_lr=0.1,steps_per_epoch = len(train_dataloader),epochs = 30)

Epoch: 30

The result as below :



Same scheduler, but this time we use a max learning rate of 0.1 instead of 0.5. The final result is still very good compared to experiment 1 and almost similar to experiment 2. Also, it is much more steady when compared to the 0.5 learning rate counter part.

The final loss was around:0.0166

5) experiment 5

Optimizer: SGD

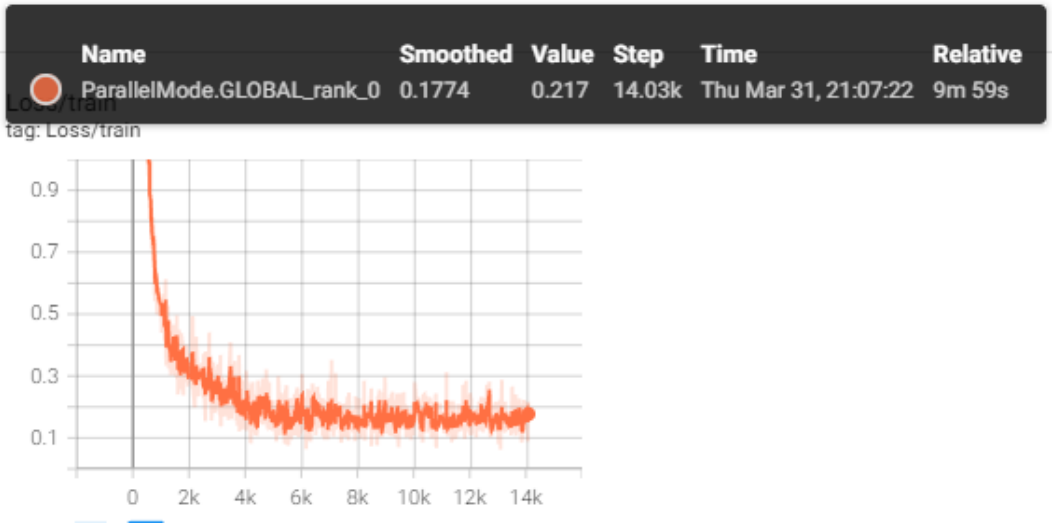
Learning rate : 0.0016

Scheduling method: MultiStepLR

Scheduler: lr_scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer,milestones=[10,20],gamma=0.1)

Epoch: 30

The result as below :



Multisteps scheduler adjust the learning according to various milestones given. The loss generates the smoothest graph yet to be seen among all experiments, which means the results it generates is robust and more likely to be converged to the true value. The end loss is satisfactory too.

The final loss was around: 0.217

Adam

6) experiment 6

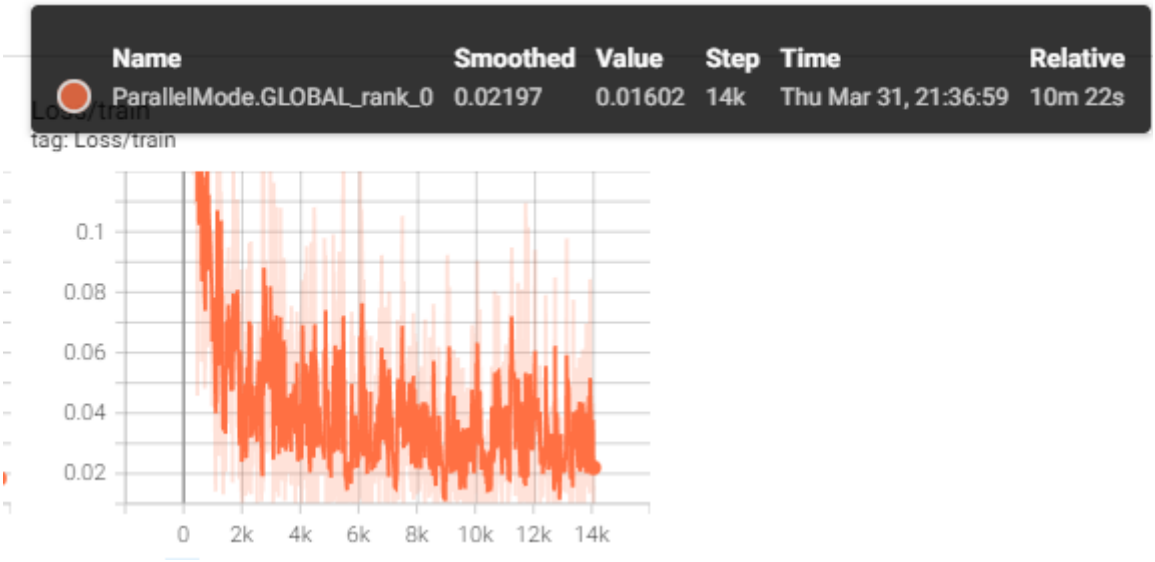
Optimizer: Adam

Learning rate : 0.0016

Scheduling method: No scheduling

Epoch: 30

The result as below:



When compared to the same no scheduling method by SGD, Adam yields a much lower end loss at the cost of unsteadiness in the graph.

The final loss was around: 0.01602

7) experiment 7

Optimizer: Adam

Learning rate : 0.0016

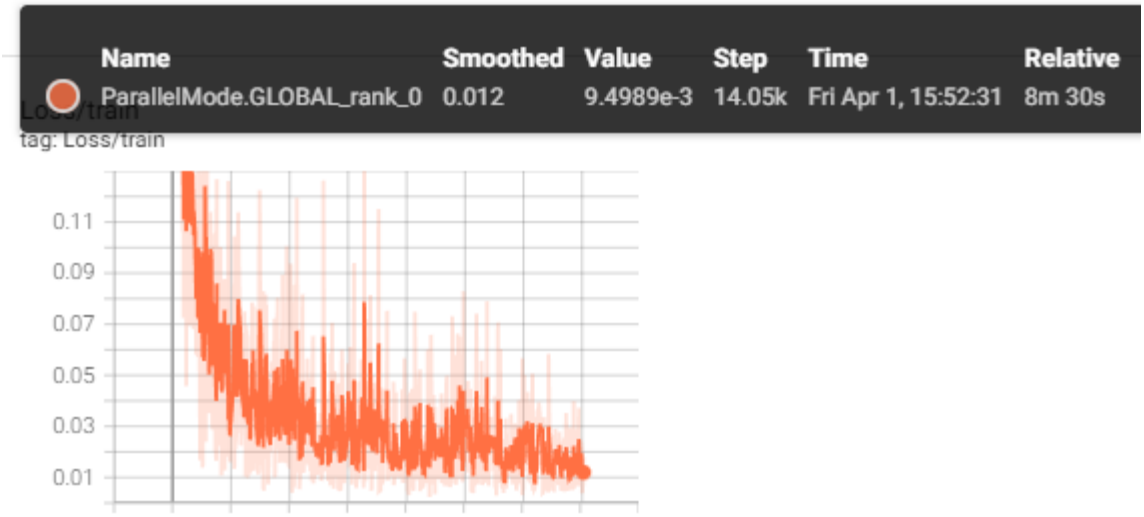
Scheduling method: multiplicativeLR

Epoch: 30

Lambda function : lmbda = lambda batch : 0.95

Scheduler = lr_scheduler = torch.optim.lr_scheduler.MultiplicativeLR(optimizer,lmbda)

The result as below :



The final loss was around: 0.0095

8) experiment 8

Optimizer: Adam

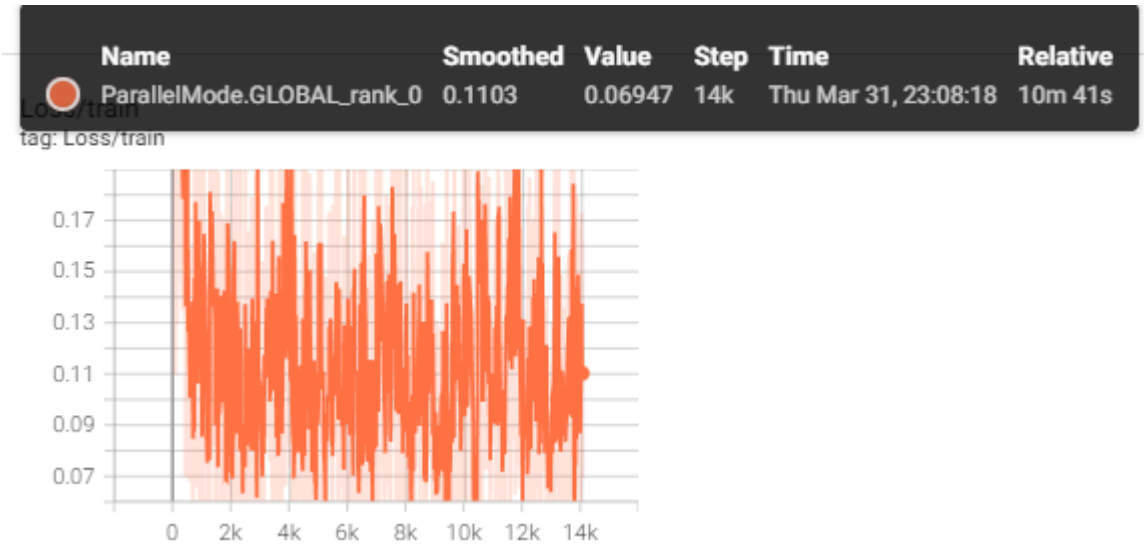
Learning rate : 0.0016

Scheduling method: CyclicLR ()

Epoch: 30

Scheduler: lr_scheduler = torch.optim.lr_scheduler.CyclicLR(optimizer,base_lr=0.01,max_lr=0.1,cycle_momentum=False)

The result as below :



Unsteady loss compared to SGD.

The final loss was around: 0.06947

9) experiment 9

Optimizer: Adam

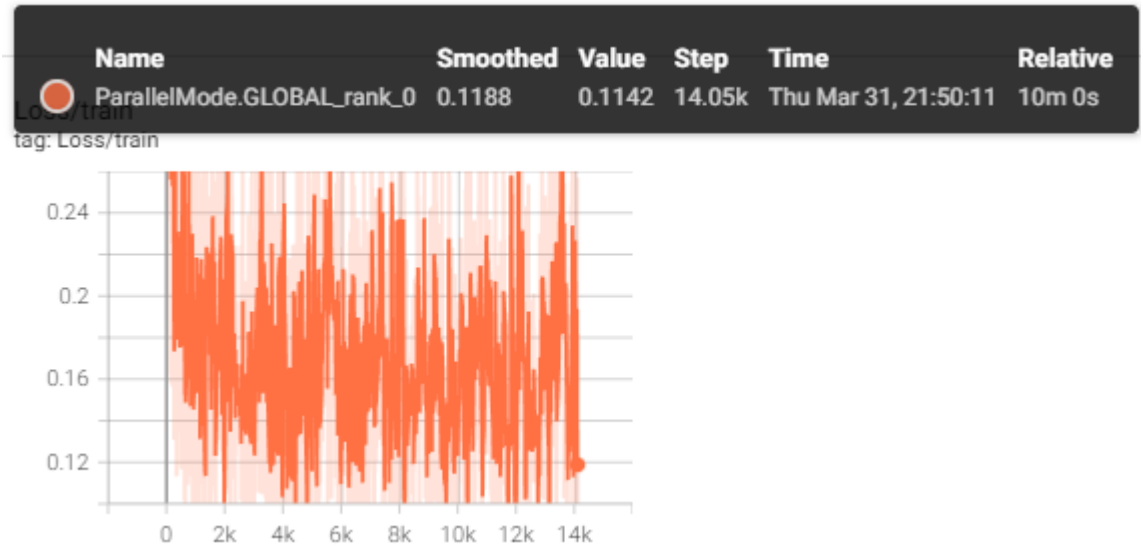
Learning rate : 0.0016

Scheduling method: OneCycleLR(max_lr = 0.5)

Scheduler : lr_scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer,max_lr=0.5,steps_per_epoch = len(train_dataloader),epochs = 30)

Epoch: 30

The result as below :



Unsteady loss compared to SGD

The final loss was around: 0.1142

Experiment 9.1

Optimizer: Adam

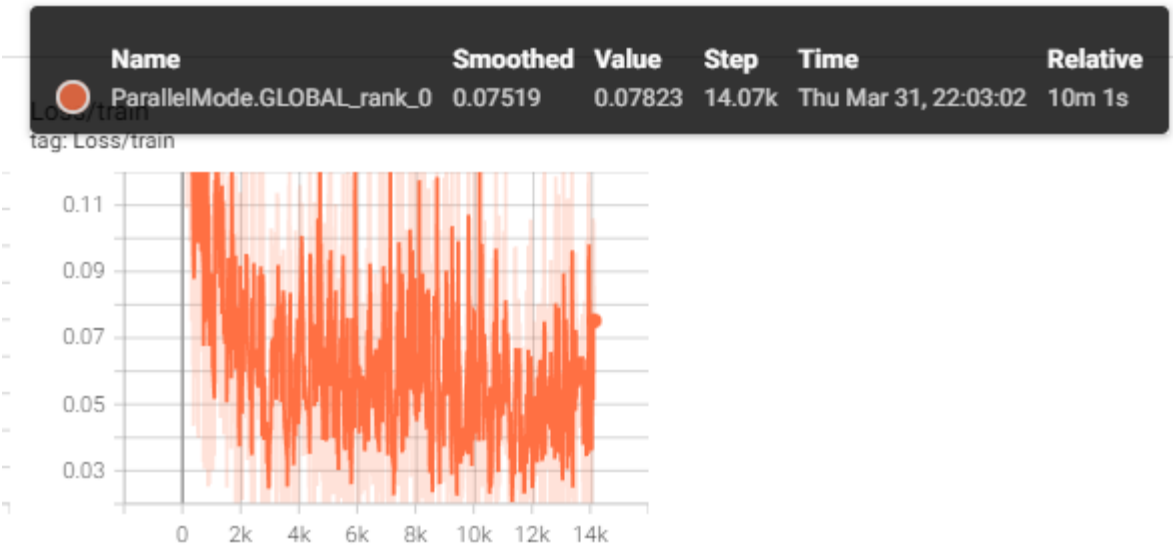
Learning rate : 0.0016

Scheduling method: OneCycleLR(max_lr = 0.1)

Scheduler : lr_scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer,max_lr=0.1,steps_per_epoch = len(train_dataloader),epochs = 30)

Epoch: 30

The result as below :



Unsteady loss compared to SGD

The final loss was around: 0.07823

10) experiment 10

Optimizer: Adam

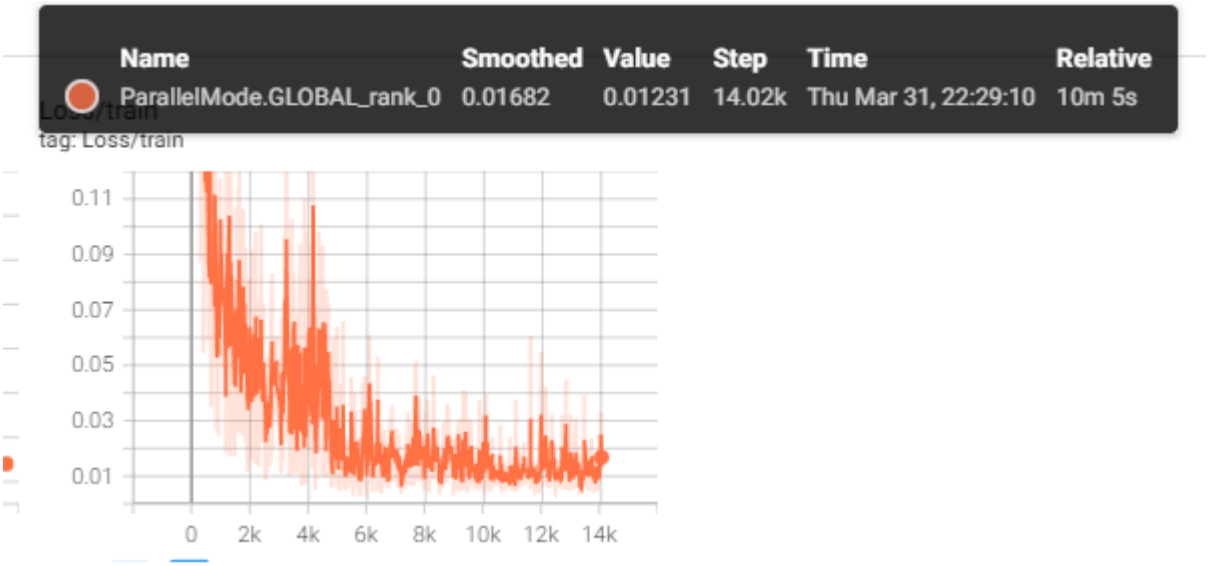
Learning rate : 0.0016

Scheduling method: MultiStepLR

Scheduler: lr_scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer,milestones=[10,20],gamma=0.1)

Epoch: 30

The result as below:



This is the only scheduler that yielded a somewhat smooth graph for ADAM. But still SGD has a much more steady loss than ADAM.

The final loss was around: 0.01231

Conclusion

Scheduler comparison among the two chosen optimizer

1. No scheduler

ADAM had a slightly lower final loss in this case, however, it is still obvious that SGD has a much more robust and steady result, which could compensate the slightly higher final loss.

2. Multisteps

Multisteps seem to generally yield a smoother loss curve compared to all other schedulers, which can be seen in both SGD and ADAM.

However, Adam seems to have a better compatibility with multisteps as it yields not only a better final loss, but its overall loss is below 0.1 which SGD is just above 0.1.

3. CyclicLR()
4. OneCycleLR(lr = 0.5)
5. OneCycleLR(lr = 0.1)

For all cycle related scheduler, Adam generates terrible results, not only is the loss curve highly unsteady, but the result is also overshadowed by SGD. SGD on the other hand, performs exceptionally well with cyclic schedulers, OneCycleLR(lr=0.1) in particular.

6. MultiplicativeLR

Its hard to tell which optimizer works better with this scheduler, although the final loss of Adam is much lower compared to SGD, however, SGD seems to yield losses around the same range nearing the end of the epoch. Also, Adam has more unsteady loss curve in this case too.

To summarize, in general, ADAM has high ups and downs compared to SGD, which means SGD always tend to have a smoother loss curve compared to ADAM.

Adam works well with multisteps, SGD works well with cyclic type of schedulers. No scheduler and multiplicative LR did not have any significant difference.