CMPT 733 Assignment 2

301474102 Sihui Wang

1. **Final Result**

   1.1 Performance

   Update: I did an experiment of *500 epochs* of training and obtained better results.

   One of the best results:

   Mean Position Error: 3.71542 meters; Mean Orientation Error: 3.18261 degrees

   Please refer to **Section 4** and the following link for details:

   https://colab.research.google.com/drive/1rcWXNM1K8IUf34z8siwTi_WWzj5gm0vN?usp=sharing

   Following is the best result that I have obtained *within 200 epochs* of training.

   https://colab.research.google.com/drive/1XCdGusWJz8gw-56jyv5UuauiDVIaymsj?usp=sharing

```
333
GT      | xyz: [-38.987651 -16.333623   1.408049]       wpqr: [ 0.607569  0.546396 -0.392806  0.421918]
PRED    | xyz: [-24.26133724 -15.09480869   1.68154251] wpqr: [ 0.61359425  0.5630312  -0.36732773  0.40517135]
ACC     | pos: 14.780858455496846 m     ori: 4.024710020238699 degrees
334
GT      | xyz: [ -8.519983 -19.774672   1.73377 ]       wpqr: [ 0.676746  0.617637 -0.286078  0.280533]
PRED    | xyz: [-11.8610045  -16.136162     1.72139646] wpqr: [ 0.66644945  0.61686078 -0.27923667  0.30406023]
ACC     | pos: 4.9397705202241555 m     ori: 3.0376469035942892 degrees
335
GT      | xyz: [ 28.283813 -28.406858   1.991715]       wpqr: [ 0.695298  0.647519  0.199379 -0.239848]
PRED    | xyz: [ 36.74899534 -25.24911133   1.65139403] wpqr: [ 0.71600113  0.6358877   0.20204665 -0.24508798]
ACC     | pos: 9.041376800550802 m      ori: 2.599239289290991 degrees
336
GT      | xyz: [ 39.311314 -30.590434   1.631193]       wpqr: [ 0.715271  0.664481  0.131422 -0.171989]
PRED    | xyz: [ 34.31816027 -24.98971944   1.69037206] wpqr: [ 0.72298289  0.65042603  0.18578296 -0.21919122]
ACC     | pos: 7.503538495934713 m      ori: 8.241053854357087 degrees
337
GT      | xyz: [ 23.985989 -26.478699   2.105896]       wpqr: [ 0.697299  0.693116  0.122218 -0.135746]
PRED    | xyz: [ 25.8597726  -22.16090455   1.6718614 ] wpqr: [ 0.7096424   0.64949679  0.14976758 -0.16325195]
ACC     | pos: 4.726817106974724 m      ori: 6.732073732625308 degrees
338
GT      | xyz: [  2.661493 -21.929875   1.822724]       wpqr: [ 0.69484   0.670043 -0.184019  0.18541 ]
PRED    | xyz: [  2.9688156  -18.70962874   1.78081592] wpqr: [ 0.71916706  0.65929331 -0.12960439  0.14561448]
ACC     | pos: 3.235149064327363 m      ori: 8.307045515246886 degrees
339
GT      | xyz: [ 14.066624 -24.0122     1.956258]       wpqr: [ 0.727741  0.674116  0.083287 -0.094996]
PRED    | xyz: [ 19.97769822 -21.5328742    1.74654568] wpqr: [ 0.74086752  0.66287019  0.07771286 -0.08926981]
ACC     | pos: 6.41341048664287 m       ori: 2.1770063565886915 degrees
340
GT      | xyz: [-16.41497 -18.00238   1.73304]  wpqr: [ 0.665271  0.590037 -0.314774  0.331945]
PRED    | xyz: [-18.51551631 -15.30124896   1.69353049] wpqr: [ 0.65461026  0.5756123  -0.32509817  0.36189705]
ACC     | pos: 3.421982567422539 m      ori: 4.1709434661268245 degrees
341
GT      | xyz: [  8.198709 -22.460965   1.748254]       wpqr: [ 0.764411  0.608538 -0.152823  0.148331]
PRED    | xyz: [  3.74087946 -19.5108154    1.80462807] wpqr: [ 0.76519504  0.60939442 -0.15684506  0.15975619]
ACC     | pos: 5.345914791482779 m      ori: 1.3331774170383317 degrees
342
GT      | xyz: [ 18.804973 -26.229792   1.945266]       wpqr: [ 0.767357  0.638709  0.0545   -0.015585]
PRED    | xyz: [ 20.93533148 -22.65420042   1.79089782] wpqr: [ 0.77664628  0.63689839  0.04441703 -0.07053441]
ACC     | pos: 4.164986422730838 m      ori: 6.406151202137063 degrees
-----------------------------
Median position error: 4.809528720389559 m       Median orientation error: 5.218451454703542 degrees
Mean position error: 6.637850713058656
Mean orientation error: 5.497746225017729
```

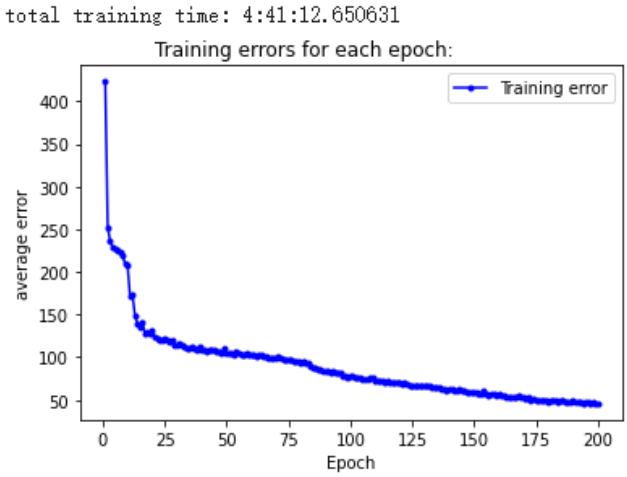
On the testing dataset, the **mean error for position estimation** is: 6.63785 meters;

The **mean error for orientation estimation** is: 5.49775 degrees.

1.2 Training Time and Training Loss

This model is trained for **200 epochs**, the above results are from **epoch 199**.

The **total training time** is: 4 hours 41 minutes 12.650631 seconds.



1.3 Model Configurations

1.3.1 Hyperparameters:

Loss Function: Sum of each row vector's L^2 norm (See Section 1.3.2(2) for details)
Weights for the loss function: $w = (0.3, 0.3, 1),\ \beta = 300$
Optimizer: Adam

Learning Rate: γ =0.0001
eps: $\epsilon = 1$
Weight decay: $\lambda = 0.0625$
Betas: $(\beta_1, \beta_2) = (0.9, 0.999)$

Batch size: 64
Training Epochs: 200

1.3.2 Modifications for the Implementation:

(1) *Replace LocalRespNorm by CrossMapLRN2d*:

CrossMap**LRN**2d is a variant of **L**ocal**R**esp**N**orm (LRN). In this implementation, I replaced LocalRespNorm by CrossMapLRN2d and found out that this replacement didn't negatively impact the performance of the model.

(2) *Change the Loss Function*:

Although it is considered the standard implementation to use:

torch.norm(v - v_gt, p = 2)

to calculate the L^2 norm of $|v - v_{gt}|$, in this implementation, I considered an alternative implementation for the loss function.

During the training process, the vectors v and v_{gt} are loaded in a batch manner, which means that v and v_{gt} are in fact matrices in calculations during training. So, torch.norm actually returns the norm of the matrix $v - v_{gt}$. However, what we want is the sum of the norms of the difference between the sample vectors and the ground truth vectors, which might be different from the norm of the matrix if batch size is greater than 1. (See Appendix for an example)

So, I proposed an alternative implementation of the loss function:

loss = nn.MSELoss(reduction = 'none')

torch.sum(torch.sqrt(torch.sum(loss(v, v_gt), dim = 1)))

Basically, torch.sqrt(torch.sum(loss(v, v_gt), dim = 1)) returns:

torch.norm(v[i, :] - v_gt[i, :], p = 2)

for each row i, and torch.sum() returns the sum of norms of the row vectors.

I have done preliminary tests and found out the above two implementations return different results if the batch size is greater than 1. So, I decided to test both implementations' performance. In this part, I use my alternative implementation of the loss function.

(3) *Remove Normalization for the Images*:

In this assignment, we are required to subtract a mean image and use a mean of 0.5 to normalize each channel of the images. I think maybe we no longer need to use a mean of 0.5 to normalize the images after we subtract the mean images. So, in this part, I just keep the "mean image subtraction" part and remove the "image normalization" part. (Normalization with the standard deviation of 0.5 is also removed for this part.)

2. Implementation Details

I did tests with 5 variants of the neural network. Each variant has slightly different choices of implementations. Here in this section I make a brief introduction to all implementation choices presented in my tests.

2.1 Local Response Norm (LRN) in Pre-Layer

Choice A: nn.LocalResponseNorm(2)

Choice B: nn.CrossMapLRN2d(5, 0.0001, 0.75, 1)

It is possible that both of the above choices are variants of LRN. So I tested both. In my tests, I didn't find that choice B might be problematic or negatively impact the overall performance.

2.2 Loss Function

Choice A: torch.norm(, p=2)

This seems to be the most straight-forward implementation:

```
def forward(self, p1_xyz, p1_wpqr, p2_xyz, p2_wpqr, p3_xyz, p3_wpqr, poseGT):
    # TODO: Implement loss
    # First 3 entries of poseGT are ground truth xyz, last 4 values are ground truth wpqr
    gt_xyz = poseGT[:,0:3]
    gt_wpqr = F.normalize(poseGT[:,3:7])
    loss1_xyz = torch.norm(p1_xyz - gt_xyz, p = 2)
    loss1_wpqr = torch.norm(p1_wpqr - gt_wpqr, p = 2)
    loss2_xyz = torch.norm(p2_xyz - gt_xyz, p = 2)
    loss2_wpqr = torch.norm(p2_wpqr - gt_wpqr, p = 2)
    loss3_xyz = torch.norm(p3_xyz - gt_xyz, p = 2)
    loss3_wpqr = torch.norm(p3_wpqr - gt_wpqr, p = 2)
    loss = self.w1_xyz * (loss1_xyz + self.w1_wpqr * loss1_wpqr) + self.w2_xyz * (loss2_xyz + self.w2_wpqr * loss2_wpqr) + self.w3_xyz * (loss3_xyz + self.w3_wpqr * loss3_wpqr)

    return loss
```

Choice B: MSE

Choice B is suggested as an alternative which might boost performance. So, I did tests with this implementation.

```
def forward(self, p1_xyz, p1_wpqr, p2_xyz, p2_wpqr, p3_xyz, p3_wpqr, poseGT):
    # TODO: Implement loss
    # First 3 entries of poseGT are ground truth xyz, last 4 values are ground truth wpqr
    loss = nn.MSELoss()
    gt_xyz = poseGT[:,0:3]
    gt_wpqr = F.normalize(poseGT[:,3:7])
    loss1_xyz = loss(p1_xyz, gt_xyz)
    loss1_wpqr = loss(p1_wpqr, gt_wpqr)
    loss2_xyz = loss(p2_xyz, gt_xyz)
    loss2_wpqr = loss(p2_wpqr, gt_wpqr)
    loss3_xyz = loss(p3_xyz, gt_xyz)
    loss3_wpqr = loss(p3_wpqr, gt_wpqr)
    loss = self.w1_xyz * (loss1_xyz + self.w1_wpqr * loss1_wpqr) + self.w2_xyz * (loss2_xyz + self.w2_wpqr * loss2_wpqr) + self.w3_xyz * (loss3_xyz + self.w3_wpqr * loss3_wpqr)

    return loss
```

Choice C: Sum of each row vector's L^2 norm

I have discussed the reason why I tried this implementation in 1.3.2(2). Here is the code for this implementation:

```
def forward(self, p1_xyz, p1_wpqr, p2_xyz, p2_wpqr, p3_xyz, p3_wpqr, poseGT):
    # TODO: Implement loss
    # First 3 entries of poseGT are ground truth xyz, last 4 values are ground truth wpqr
    loss = nn.MSELoss(reduction='none')
    gt_xyz = poseGT[:,0:3]
    gt_wpqr = F.normalize(poseGT[:,3:7])
    loss1_xyz = torch.sum(torch.sqrt(torch.sum(loss(p1_xyz, gt_xyz),dim = 1)))
    loss1_wpqr = torch.sum(torch.sqrt(torch.sum(loss(p1_wpqr, gt_wpqr),dim = 1)))
    loss2_xyz = torch.sum(torch.sqrt(torch.sum(loss(p2_xyz, gt_xyz),dim = 1)))
    loss2_wpqr = torch.sum(torch.sqrt(torch.sum(loss(p2_wpqr, gt_wpqr),dim = 1)))
    loss3_xyz = torch.sum(torch.sqrt(torch.sum(loss(p3_xyz, gt_xyz),dim = 1)))
    loss3_wpqr = torch.sum(torch.sqrt(torch.sum(loss(p3_wpqr, gt_wpqr),dim = 1)))
    loss = self.w1_xyz * (loss1_xyz + self.w1_wpqr * loss1_wpqr) + self.w2_xyz * (loss2_xyz + self.w2_wpqr * loss2_wpqr) + self.w3_xyz * (loss3_xyz + self.w3_wpqr * loss3_wpqr)

    return loss
```

2.3 Normalization of Images

I have discussed in 1.3.2(3) that I have doubts if we still need image normalization after the "mean image subtraction" part. So, I did experiments with both options.

Choice A: Do both "mean image subtraction" and "image normalization"

Choice B: Only do "mean image subtraction" and remove "image normalization"

2.4 Normalization of Positions

In the experiments, what I found out is that the predicted position values tend to be within the range of -1 to 1, if we only train the neural network for 20 epochs. So, I think rescaling or normalizing the ground-truth position values to the range of -1 to 1 might help speed up the training process.

Choice A: No normalization

Choice B: Scaling of the position

https://colab.research.google.com/drive/1LA7sY0f3JjhGVty6IVu8DKRb6-cet8G_?usp=sharing

During training, the ground-truth position values are divided by 50 to train the neural network. During testing, the predicted position values are multiplied by 50 to compare with the ground truth.

Choice C: Normalization of the position

https://colab.research.google.com/drive/1PHdQss_bWv4tbT1i2bsC4i2T3W0xgMR5?usp=sharing

During initialization, the data loader reads the pose values of the training data and computes the means and standard deviations for the position values. For training data, each position value is subtracted by the mean and then divided by 3 times the standard deviation (because we assume 99.7% of the samples are within 3 standard deviations of the mean value).

$$GT' = \frac{GT - mean}{3 \times SD}$$

During testing, each predicted position value is multiplied by 3 times the standard deviation and then added by the mean.

$$pred' = (pred \times 3 \times SD) + mean$$

Here we assume that training data and testing data are from the same statistical distribution, and we only collect the means and standard deviations for the training dataset. This is to avoid collecting information from the testing data beforehand to unfairly improve the model's performance.

3. Results of different implementations

3.1 Model Configurations

I have done 5 tests with different implementation choices.

Model 1: Standard Model

LRN: Choice A

Loss Function: Choice A

Normalization of Images: Choice A

Normalization of Positions: Choice A

Model 2: Standard Model with MSE loss

LRN: Choice A

Loss Function: Choice B

Normalization of Images: Choice A

Normalization of Positions: Choice A

Model 3: Modified Model

LRN: Choice B

Loss Function: Choice C

Normalization of Images: Choice B

Normalization of Positions: Choice A

Model 4: Modified Model with Rescaling of Position Values

LRN: Choice B

Loss Function: Choice C

Normalization of Images: Choice B

Normalization of Positions: Choice B

Model 5: Modified Model with Normalization of Position Values

LRN: Choice B

Loss Function: Choice C

Normalization of Images: Choice B

Normalization of Positions: Choice C

All other hyperparameters are the same for these 5 models:

Weights for the loss function: $w = (0.3, 0.3, 1),\ \beta = 300$

Optimizer: Adam

Learning Rate: γ =0.0001

eps: $\epsilon = 1$

Weight decay: $\lambda = 0.0625$

Betas: $(\beta_1,\ \beta_2) = (0.9,\ 0.999)$

Batch size: 64

Training Epochs: 200

Following are the performance of the 5 models.

3.2 Model Performance

3.2.1 Model 1: Standard Model

https://colab.research.google.com/drive/1pGvKfvcda0PWGRl3hbzq-LN-TXLqmNhy?usp=sharing

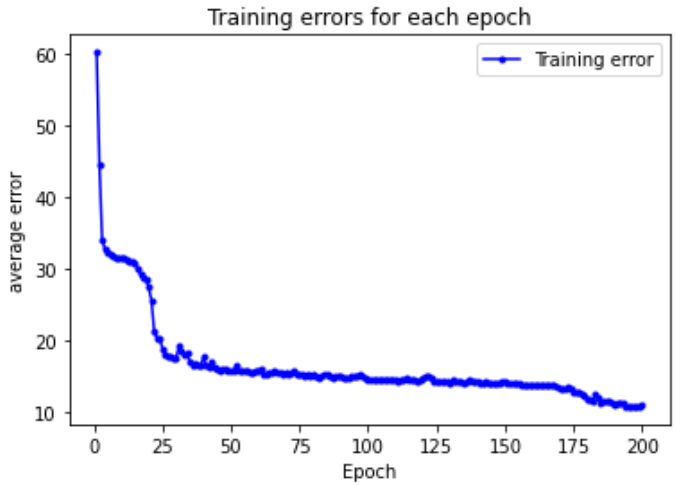
Training Epochs: 200

Training Time: 4 hours 7 minutes 46.916607 seconds

Mean error for Position Estimation: 17.45567 meters (at epoch 200)

Mean error for Orientation Estimation: 6.67963 degrees (at epoch 200)

Training Loss:



3.2.2 Model 2: Standard Model with MSE Loss

https://colab.research.google.com/drive/13Vz1IYH0RIlTMdHgSAhcC1pEdpoFcpxe?usp=sharing

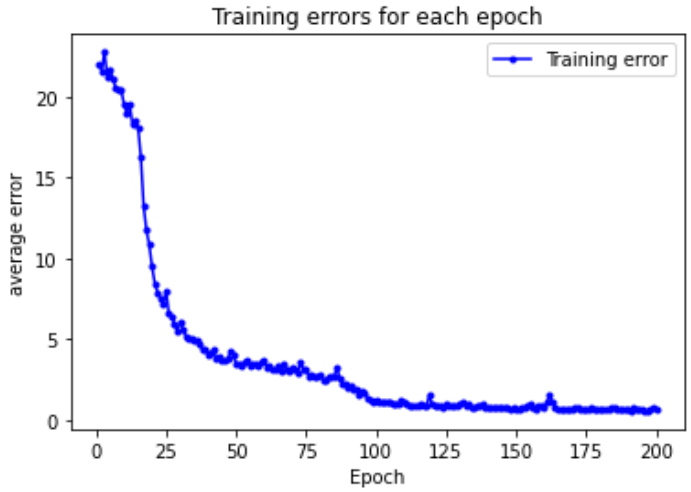
Training Epochs: 200

Training Time: 4 hours 10 minutes 46.760012 seconds

Mean error for Position Estimation: 7.29351 meters (at epoch 200)

Mean error for Orientation Estimation: 10.42262 degrees (at epoch 200)

Training Loss:



3.2.3 Model 3: Modified Model

https://colab.research.google.com/drive/1XCdGusWJz8gw-56jyv5UuauiDVIaymsj?usp=sharing

This is the best model which has been reported in Section 1.

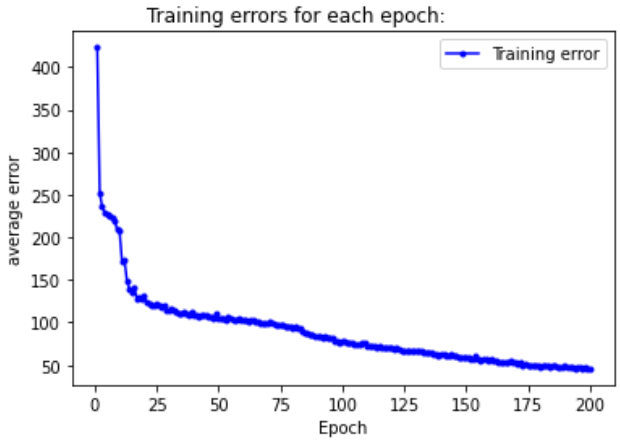
Training Epochs: 200

Training Time: 4 hours 41 minutes 12.650631 seconds

Mean error for Position Estimation: 6.63785 meters (at epoch 199)

Mean error for Orientation Estimation: 5.49775 degrees (at epoch 199)

Training Loss:



3.2.4 Model 4: Modified Model with Rescaling of the Position Values

https://colab.research.google.com/drive/1LA7sY0f3JjhGVty6IVu8DKRb6-cet8G_?usp=sharing

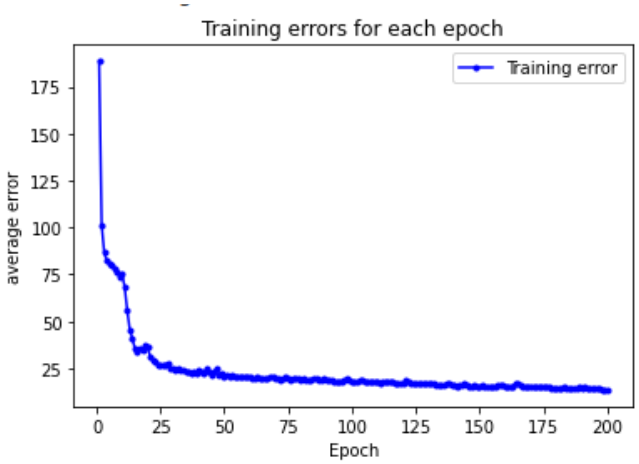
Training Epochs: 200

Training Time: 4 hours 24 minutes 44.265994 seconds

Mean error for Position Estimation: 9.41927 meters (at epoch 198)

Mean error for Orientation Estimation: 4.10936 degrees (at epoch 198)

Training Loss:



3.2.5 Model 5: Modified Model with Normalization of the Position Values

https://colab.research.google.com/drive/1PHdQss_bWv4tbT1i2bsC4i2T3W0xgMR5?usp=sharing

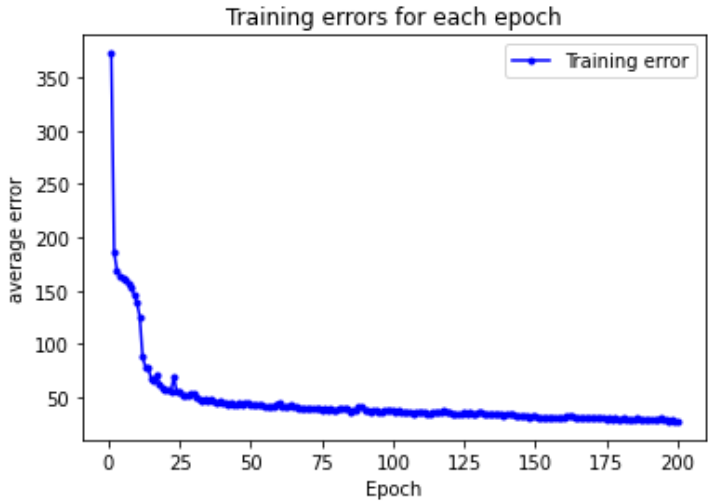
Training Epochs: 200

Training Time: 4 hours 41 minutes 26.985954 seconds

Mean error for Position Estimation: 9.33576 meters (at epoch 200)

Mean error for Orientation Estimation: 3.86015 degrees (at epoch 200)

Training Loss:



3.3 Comparison of Model 3~5

If we set a *performance threshold*:
Mean error for Position Estimation: < 10 meters;
Mean error for Orientation Estimation: <10 degrees;
then model 3,4,5 all pass this threshold.

One of the advantages of Model 4 and Model 5 is that their training losses drop rapidly and their performances improve faster in the early stage of training. In my experiments, ***Model 4*** *passed the performance threshold at* ***epoch 53****, and* ***Model 5*** *passed the performance threshold at* ***epoch 50****, whereas* ***Model 3*** *passed the performance threshold at* ***epoch 138***. This is understandable, since in the early stage of training the predicted values tend to be within the range of -1 to 1, and it takes a while for the network to adjust to the unnormalized values that are not in the range of -1 to 1, given the learning rate of 0.0001.

However, in the later stage of training, Model 4 and Model 5's performances stagnate, while Model 3's performance continues to improve. Perhaps this is because after rescaling the learning rate becomes too large for Model 4 and Model 5.

Despite Model 3's drawbacks, I still consider Model 3 as the best model in current experiments and configurations, because it provides more accurate prediction for the position, and its accuracy continues to improve with more epochs of training.

4. **Best-so-far Results**

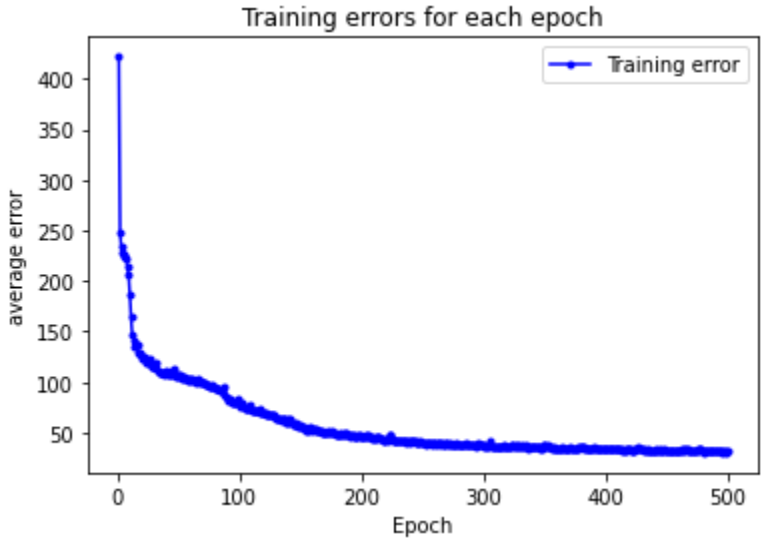
Since the loss for Model 3 is still decreasing by the end of 200 epochs of training, I decided to test Model 3 with more epochs of training.

In my experiment, I trained Model 3 for 500 epochs. Here are the results.

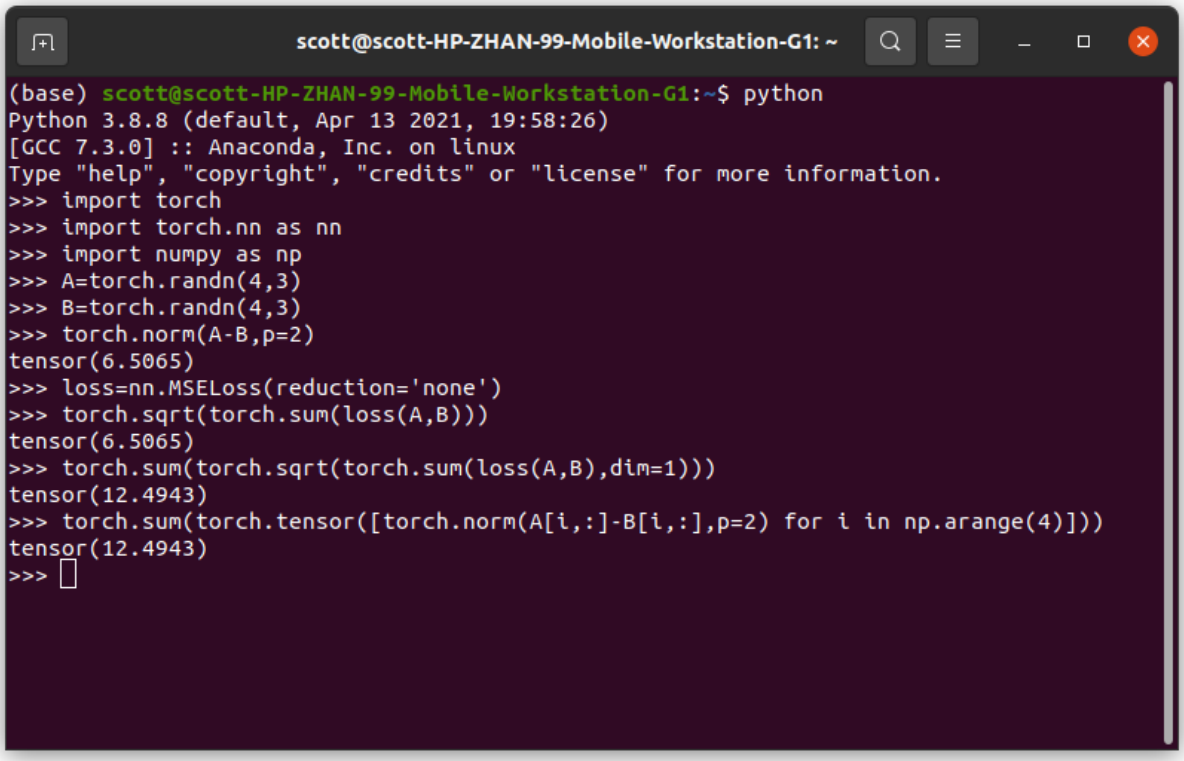
https://colab.research.google.com/drive/1rcWXNM1K8IUf34z8siwTi_WWzj5gm0vN?usp=sharing

Training Epochs: 500
Training Time: 11 hours 35 minutes 26.653830 seconds
Mean error for Position Estimation: 3.75705 meters (at epoch 500); 3.71542 meters (at epoch 400)
Mean error for Orientation Estimation: 3.53772 degrees (at epoch 500); 3.18261 degrees (at epoch 400)

Training Loss:



Appendix: Difference between Choice A and Choice C of the Loss Function



Both:

torch.norm(A-B, p=2)

and:

loss=nn.MSELoss(reduction='none')
torch.sqrt(torch.sum(loss(A,B)))

did the same job. They compute the following:

$$\sqrt{\Sigma_{i,j}(a_{ij}-b_{ij})^2}$$

Both:

torch.sum(torch.sqrt(torch.sum(loss(A,B),dim=1)))

and:

torch.sum(torch.tensor([torch.norm(A[i,:]-B[i,:],p=2) for i in np.arange(4)]))

did the same job. They compute the following:

$$\Sigma_i\sqrt{\Sigma_j(a_{ij}-b_{ij})^2}$$

This is why I think choice A and choice C are different and I test them both in the experiments.