Edoardo Bastianello

ID: 2053077

REPORT LAB 4

1) Function *__get_item__(self, idx)* of the class *PointCloudData(Dataset)*:

First, I needed to select the anchor point *pt1*. To do this, I got a random point from the input point cloud. Next, I created a KD-tree with the point of its point cloud. Then, I extracted all the indices of the neighbors of the point *pt1* by calling the function *search_radius_vector_3d(pt1, self.radius)* on the KD-tree that I just created. In particular, this function returns all the indices of the points having a distance to *pt1* lower than the radius. Finally, I recovered the points corresponding to the indices that were returned. In this way, I extracted all points included in a spherical region with the specified radius around *pt1*.

Next, I had to find the positive correspondence *pt2* in the noise point cloud *pcd2*. To do this, I created a KD-tree with the points of the point cloud *pcd2* and I searched for the nearest neighbor of *pt1* by calling the function *search_knn_vector_3d(pt1, 1)* on the KD-tree I just created. Next, I got the neighbors of *pt2* as done previously for *pt1* by calling the function *search_radius_vector_3d(pt2, self.radius)* on the KD-tree with the points of the point cloud *pcd2*.

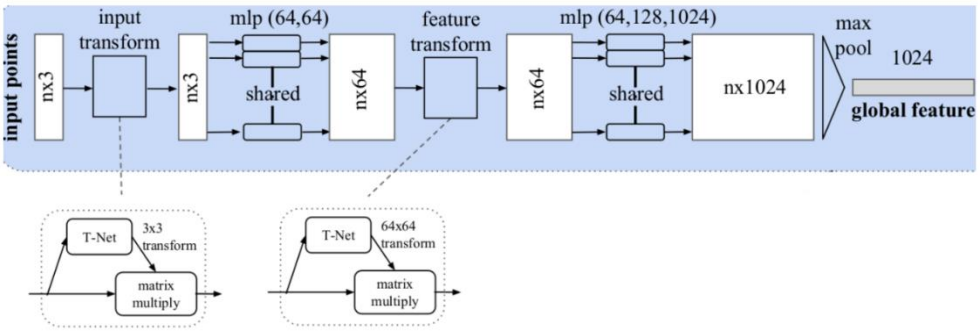
Finally, I had to get a negative correspondence *neg_pt*. To do this, I kept extracting a random point from the point cloud *pcd2* until I got a candidate having a distance from *pt1* greater or equal than the threshold *self.min_dist.* Once I found this candidate, I set it as *neg_pt*. At this point, I obtained the neighbors of *neg_pt* as done previously by using a KD-tree with the points of *pcd2* and the function *search_radius_vector_3d(neg_pt, self.radius).*

Finally, I normalized the points of the neighbors of the three extracted points (anchor, positive and negative) by subtracting the coordinates of the relative extracted point, as shown below.

```
# normalize points
point_set1 = np.subtract(point_set1, pt1)
point_set2 = np.subtract(point_set2, pt2)
point_set3 = np.subtract(point_set3, neg_pt)
```

2) *__init__(self)* in *TynyPointNet(nn.Module)*:

Here I defined the layers of the *TinyPointNet* architecture represented in the figure (however, the dimension of the output feature was set to 256 instead of the original value 1024):



I defined the MLP(64,64) represented in the figure above as follows:

```
self.mlp1_1 = MLP(3,64)
self.mlp1_2 = MLP(64, 64)
```

Next, I defined the feature transform module as follows, by using the already implemented TNet:

```
self.feature_transform = TNet(64)
```

Then I defined the MLP(64, 128, 256) as:

```
self.mlp2_1 = MLP(64, 64)
self.mlp2_2 = MLP(64, 128)
self.mlp2_3 = MLP(128, 256)
```

3) ***forward(self, input)*** **in** ***TynyPointNet(nn.Module)****:*

First, I sent the transformed input, *input_transform_output*, to the MLP layers:

```
x = self.mlp1_1(input_transform_output)
x = self.mlp1_2(x)
```

Then, I sent the result to the feature transform module *self.feature_transform(x)*. The result, *feature_transform_output*, was then sent to the other MLP layers:

```
x2 = self.mlp2_1(feature_transform_output)
x2 = self.mlp2_2(x2)
x2 = self.mlp2_3(x2)
```

Finally, the global feature was obtained by applying a max pooling layer to the output of the MLP.

4) ***tinypointnetloss*** **in** ***train(…):***

TinyPointNet should be trained by using a Triplet loss L as loss function:

$$\mathcal{L}(A, P, N) = \max(\| \mathrm{f}(A) - \mathrm{f}(P) \|_2 - \| \mathrm{f}(A) - \mathrm{f}(N) \|_2 + \alpha, 0)$$

This loss is already implemented in Pytorch by the function *nn.TripletMarginLoss(…).* So, I used this function to define *tinypointnetloss*. For the margin, I set it to its default which is 1.

Results:

Various tests were made to find the best parameters for the training. Here are reported the most remarkable ones. I also did other additional tests, but they didn't lead to any improvements.

TEST 1

1) Parameters of the first test:
    - ***radius* = 2*10e-3**
    - *samples_per_epoch* = 500
    - ***learning rate* = 0.005**
    - *epochs* = 45
    - *batch_size* = 50

I trained 3 different models with the same parameters in order to reduce the influence of random processes. Here are reported a few examples of accuracy for the different models:

- Model_1_1 = 89,47%
- Model_1_2 = 72.727%
- Model_1_3 = 45.161%

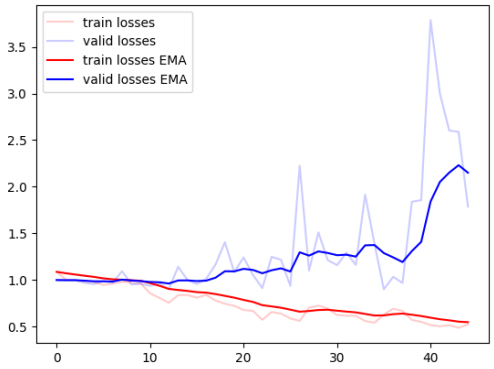


Figure 1- train graph relative to Model_1_1

The accuracy values for different models trained with the same parameters are very different, with a large range between the best and the worst results.

Also, from the graph (*Figure 1*) one can notice that the model is overfitting since the training loss is decreasing while the validation loss is increasing. This is due to the low amount of training data and to the high learning rate. This also explains the very different values of the accuracy.

TEST 2

2) Parameters of the second test:
    - ***radius* = 3*10e-3**
    - the rest of the parameters is the same as **test 1**.

These are examples of the resulting accuracy values obtain for different models all trained with these parameters (multiple training were made to reduce the influence of random processes):

- Model_2_1 = 83,871%
- Model_2_2 = 73,529%
- Model_2_3 = 51,351%

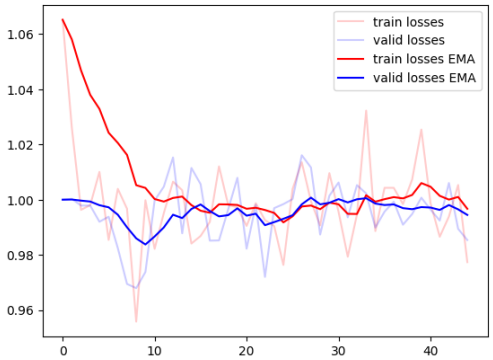


Figure 2 - train graph relative to Model_2_1

From the graph (*Figure 2*), one can notice how the validation loss is more constant with respect to **test 1**. Also, the training loss starts to become constant after a while as well. So, it seems that increasing the radius of the neighbourhood helps to diversify the descriptors of the positive matches with respect to the negative matches. However, it still seems that the model hasn't converged yet, since the values of the accuracy are still

very different. An explanation of this, could be that the model learns too quickly. For this reason, in the next test a lower learning rate was used.

TEST 3

3) Parameters of the third test:
    - ***learning rate* = 0.001**
    - the rest of the parameters is the same as **test 2**.

These are the resulting accuracy values obtain for different models all trained with these parameters (multiple training were made to reduce the influence of random processes):

- Model_3_1 = 64,865%
- Model_3_2 = 71.053%
- Model_3_3 = 68,571%
- Model_3_4 = 81.081%

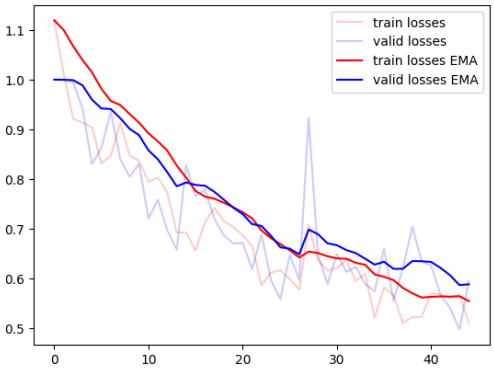


Figure 3 - train graph relative to Model_3_1

Among the previous tests, this one got the best and most consistent accuracy results. Also, from the graph (*Figure 3*), one can see that both the training and the validation loss are decreasing, which shows that the model isn't overfitting. For this reason, for the next test, I further halved the learning rate to see if this could lead to further improvements.

TEST 4 - BEST RESULTS AND FINAL MODEL

4) Parameters of the fourth test:
    - ***radius* = 3*10e-3**
    - *samples_per_epoch* = 500
    - ***learning rate* = 0.0005**
    - *epochs* = 45
    - *batch_size* = 50

As for the other tests, I trained different models with the same parameters in order to reduce the influence of random processes. Here are reported a few examples of accuracy for the different models:

- Model_4_1 = 88.889%
- Model_4_2 = 66,667%
- Model_4_3 = 60,000%
- **Model_4_4 = 92,500%**

With these parameters, the results are slight better than the ones obtained with **test 3**. For this reason, I chose these parameters for the final model that I uploaded. In particular, the model that I uploaded is **Model_4_4**.

Here are some accuracy results obtained by testing the **same final model (Model_4_4)** multiple times:

- 92,500%
- 97,222%
- 87,879%
- 94,737%
- 94,286%
- 93,750%

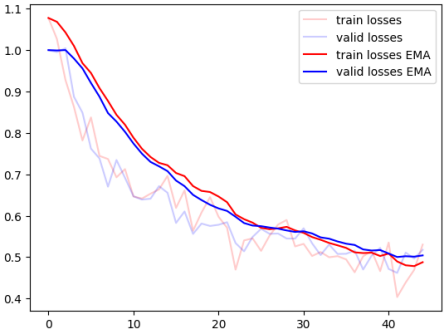


Figure 4 - train graph of Model_4_4 which is the one I uploaded.

Observation: The results obtained with this final model are very good. However, for different runs of the training, the resulting models can have different results. In any case, while the accuracy with this final model is very high, as reported above, none of the models trained with these parameters got an accuracy lower than 60%.

Note: As an additional test, I trained other models by using the parameters of **test 4**, while lowering the value of the radius to **2.75*10e-3**. However, the result that I obtained were worse, so I kept **Model_4_4** as the final model.