Stefano Binotto 2052421

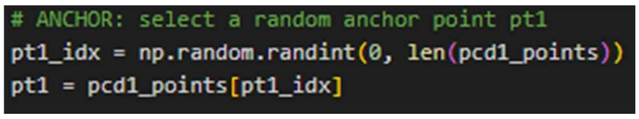
DEEP 3D DESCRIPTORS

In order to generate a 3D local descriptor, we adopted a reduced version of PointNet, the TinyPointNet, without the last classification layers and reducing the size of the global feature from 1024 to 256.

Sample generation

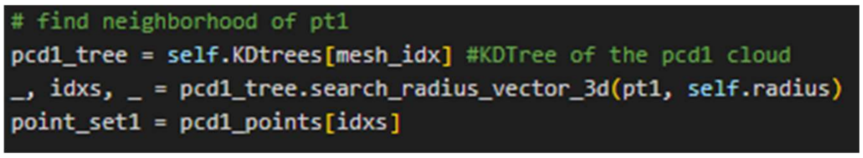
The first step was to implement the sample generation, to provide a set of positive and negative pairs of pointsets.

So at first, I picked a random **anchor** point ***pt1*** and its neighborhood from the starting point cloud ***self.pcd1***.



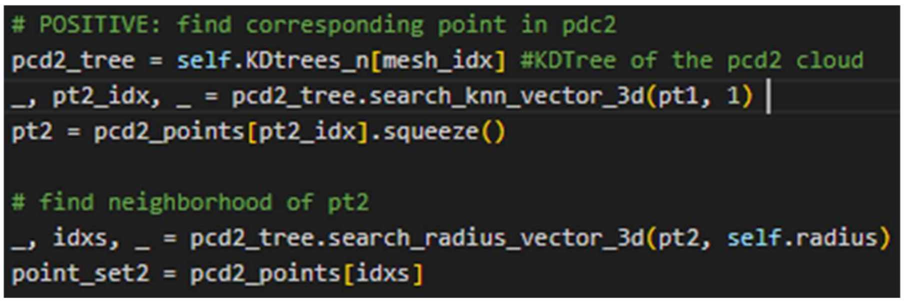
```
# ANCHOR: select a random anchor point pt1
pt1_idx = np.random.randint(0, len(pcd1_points))
pt1 = pcd1_points[pt1_idx]
```

I found the neighborhood ***point_set1*** using the original pointcloud KD-Tree to find the closest points to the anchor within a given radius.



```
# find neighborhood of pt1
pcd1_tree = self.KDtrees[mesh_idx] #KDTree of the pcd1 cloud
_, idxs, _ = pcd1_tree.search_radius_vector_3d(pt1, self.radius)
point_set1 = pcd1_points[idxs]
```

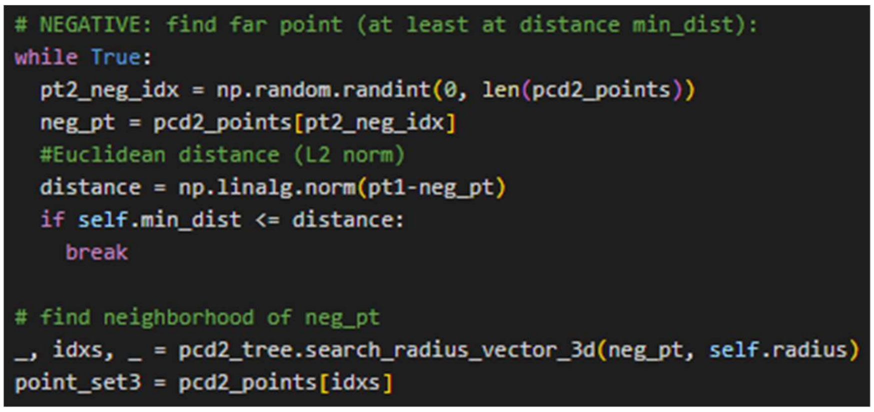
Then I found the **positive example** by exploiting the KD-Tree of the noisy pointcloud ***self.pcd2*** to compute the closest point ***pt2*** to the anchor. Then, using the same KD-Tree, I sampled the neighborhood ***point_set2*** within a given radius.



```
# POSITIVE: find corresponding point in pdc2
pcd2_tree = self.KDtrees_n[mesh_idx] #KDTree of the pcd2 cloud
_, pt2_idx, _ = pcd2_tree.search_knn_vector_3d(pt1, 1)
pt2 = pcd2_points[pt2_idx].squeeze()

# find neighborhood of pt2
_, idxs, _ = pcd2_tree.search_radius_vector_3d(pt2, self.radius)
point_set2 = pcd2_points[idxs]
```

To find the **negative example** I just scanned the noisy cloud *self.pcd2* to look for a point far from the anchor. As soon as a candidate was farther than the ***self.min_dist*** threshold, that candidate was set as the negative example ***neg_pt***. Then, like for the positive example, I found the surrounding neighborhood ***point_set3*** around that point.



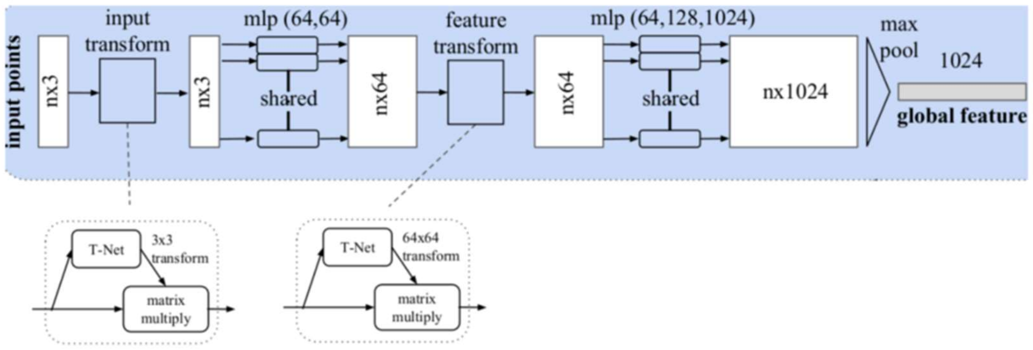
```
# NEGATIVE: find far point (at least at distance min_dist):
while True:
  pt2_neg_idx = np.random.randint(0, len(pcd2_points))
  neg_pt = pcd2_points[pt2_neg_idx]
  #Euclidean distance (L2 norm)
  distance = np.linalg.norm(pt1-neg_pt)
  if self.min_dist <= distance:
    break

# find neighborhood of neg_pt
_, idxs, _ = pcd2_tree.search_radius_vector_3d(neg_pt, self.radius)
point_set3 = pcd2_points[idxs]
```

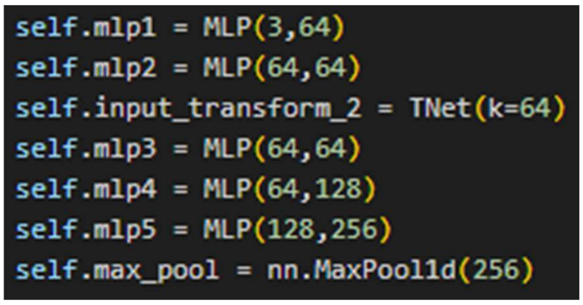
Finally, the three pointsets were all **normalized** subtracting the corresponding selected point.

TinyPointNet architecture

The second step was the implementation of the TinyPointNet.



In the ***__init__*** method of the network class I initialized all the layers as indicated in the figure above, reducing only the dimensionality of the output ***global feature*** to **256**.



```
self.mlp1 = MLP(3,64)
self.mlp2 = MLP(64,64)
self.input_transform_2 = TNet(k=64)
self.mlp3 = MLP(64,64)
self.mlp4 = MLP(64,128)
self.mlp5 = MLP(128,256)
self.max_pool = nn.MaxPool1d(256)
```

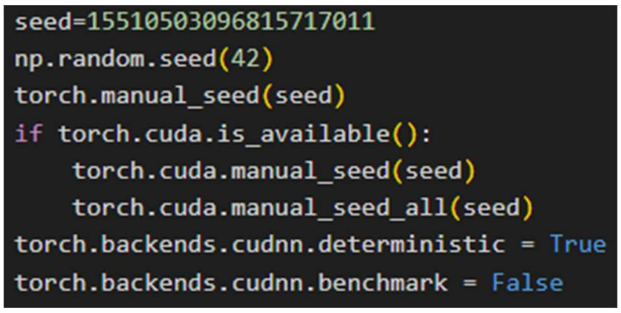
After the first input trasformation, to allign the original pointcloud to a canonical space with the rotation matrix obtained using the **TNet**, I appended a set of MLPs, in order to have a couple of layers with 64 neurons each, to learn a local feature representation of size 64 for each point.
Then I added another transformation module in order to perform allignment also in feature space.
After that I processed the aligned feature vectors again using another set of three MLPs, specifically 3 layers of 64, 128, and 256 neurons, respectively. Then a symmetric function (MaxPooling) is applied to aggregate the information from each point of the spherical neighborhood to obtain the 256-dimensional feature representation (descriptor).
So, the implementation of the **forward** method was straightforward.

Loss function

The network is trained using a Triple Loss function (**nn.TripletMarginLoss**), with the default value of the **margin**, equal to 1.

Experiment and result

In order to make the results as reproducible as possible, I added this code at the beginning in order to **control the sources of randomness**.



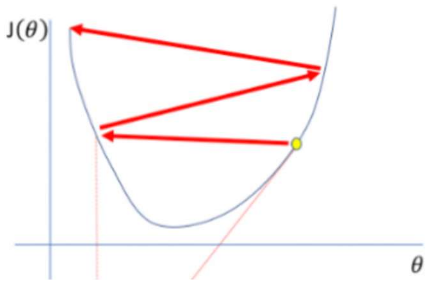
```
seed=15510503096815717011
np.random.seed(42)
torch.manual_seed(seed)
if torch.cuda.is_available():
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

This part of the assignment was the hardest one, since it's very time-consuming, especially using Collaboratory.
The main tunable hyperparameters I found more effective in terms of accuracy increase are the following:

- **Radius** of the neighborhood
- **Learning rate**
- Number of **samples per epoch**

I also tried to change other parameters, like the margin in the Triplet loss function, however in all these tests the accuracy decreases.

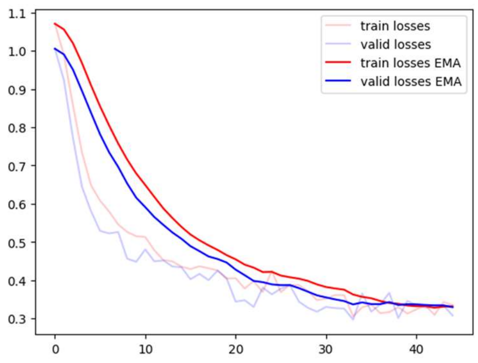
Initially I used the **default values** of the radius, lr and samples per epoch. However the model was clearly **overfitting**, since the training loss curve (and also validation loss) was unstable and wasn't decreasing.
So I thought that the training set was **not sufficiently large** since, by default, the samples per epoch were only 500. So I tried to increase this parameter until I found that the maximum number of samples per epoch that increases the accuracy was **2000**.



However I observed that the training loss curve was still a bit unstable. This made me think that the learning rate was maybe too high, so the updates were **too drastic** (like the example on the figure on the right) and the gradient descend couldn't converge properly. So I decreased the learning rate until I found the best value equal to **0.0001**.

Finally, I increased also the radius of the neighborhood and I observed that the accuracy scores were a little bit higher.

The final learning curves were like the following one:



To sum up, using the following **hyperparameters**:

1. Learning rate = 0.0001
2. Samples per epoch = 2000
3. Radius = 3*10.0e-3

over **<u>10 different trained models</u>**, I achieved the following **average accuracy** score: **76.564%**, with a **minimum of 66.667%**, a **maximum of 84.848%** and a **standard deviation** equal to **7.02**.
As we can notice from the informations above, every accuracy can be a **bit different** from the others. Probably, the main reasons are that the training set is not large enough and diversified, and also because the set of test samples is pretty small (only about 30).

Inside the submitted folder I inserted the jupyter notebook, this report and the model .yml of one of the 10 trained models I used to compute these results.