Student Name: Federico Gelain
Student ID: 2076737

UNIVERSITÀ DEGLI STUDI DI PADOVA
3D Data Processing
Lab 4: Deep 3D Descriptors

Task 1: Sample generation

The goal is to correctly sample the following set of points: the reference set of points, called **anchor**, the set of **positive** points, which represents a correct match with respect to the anchor and the set of **negative** points, which represents a wrong match with respect to the anchor.
Starting from the anchor set, the anchor point is sampled uniformly at random and its neighbourhood is computed using the function ***search_radius_vector_3d()***, which retrieves all points whose distance from the anchor is smaller than the class variable ***radius***.
As for the positive set, the closest point to the anchor set is obtained using ***search_knn_vector_3d()***, setting $k = 1$, and its neighbourhood in the same way described above.
Finally, for the negative set, to find the negative reference point, the fastest way to do it is to perform a while loop where a point is sampled and its distance from the anchor point is computed. If it is larger than the class variable ***min_dist***, then the point chosen is a valid negative point and the loop can be interrupted. Its neighbourhood is again found using search_radius_vector_3d(). Below some examples of the sets of points obtained trying different values for the radius:

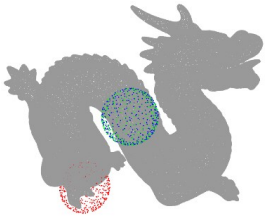


Figure 1: spheres created using default radius of 0.02

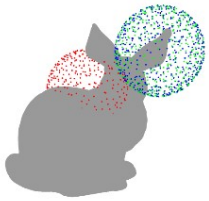
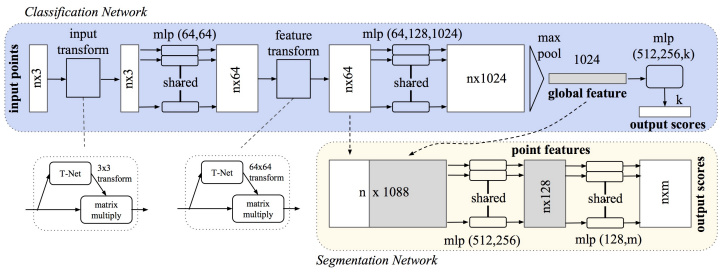


Figure 2: spheres created using radius of 0.04

Task 2: TinyPointNet implementation

The implementation of the TinyPointNet architecture follows the one of PointNet, shown below, with two main differences: the input transform is computed as the local reference rotation matrix using SHOT (Signatures of Histograms of OrienTations) and the global feature returned by the network will have a lower size (256 instead of 1024).



Starting from the first point, the covariance matrix is computed as:

$$M = \frac{1}{\sum_{i:d_i \leq R} R - d_i} \sum_{i:d_i \leq R} (R - d_i)(p_i - p)(p_i - p)^T, \qquad R = 1, \quad d_i = \|p_i - p\|^2$$

To make the computation easier and more efficient, the points whose distance is greater than R are assigned a weight equal to 0 and the matrix containing the difference between each point and centroid is already stored in the matrix ***centered_features***, allowing to compute the covariance matrix in a single line. From that, the matrix containing the eigenvectors is obtained using ***torch.linalg.eig()***, which represents the local input rotation. Since the function returns complex vectors (even if in this case the imaginary part is always equal to 0), an explicit cast to double is performed. The vectors are sorted in the matrix based on the value of the corresponding eigenvalues in decreasing order.
As for the implementation of the network architecture:

- in the **__init__** method, the feature transform layer (TNet with k=64) and the 5 MLP layers are created;
- in the **forward** method, the rotation matrix is first obtained using the SHOT procedure described above and multiplied with the input. Then, using two MLP layers, the result is mapped from dimension 3 to 64 and from 64 to 64. The TNet layer returns the feature transform, which is multiplied to the input it received and after that, three MLP layers map the result from 64 to 64, from 64 to 128 and finally from 128 to 256. The final global feature vector is obtained by applying max pooling.

Task 3: Loss function definition

To train the network, the **Triplet Loss function** is used:

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

where A, P, N are respectively the anchor, positive and negative point sets, α is the margin. Pytorch already provides it as **torch.nn.TripletMarginLoss()**, leaving its default parameters $\alpha = 1.0$ and $p = 2.0$ (p is the norm degree for pairwise distance).

Results

The results can vary quite a bit between each run of the code. Using the default parameters provided, the accuracy obtained varies between 30% and 55%. Increasing the radius to find the neighbourhood for anchor, positive and negative points seems to improve the performance quite a bit, more consistently achieving an accuracy above 50%, at the expense of a slightly higher training time. It has to be noted though that this comes with the risk of obtaining sets spheres that intersect (so negative points that are at the same distance from the positive ones for example). Finally, generating the test set sampling keypoints using open3D (returning about 40 samples, as opposed to the usual 2000) yields way better results, with an accuracy achieved by the model which is usually around 80% (keeping the default radius of 0.02 and with the lowest registered accuracy of 61.3%). Below are shown the graphs of the losses obtained for the best models for each one of the three described settings:

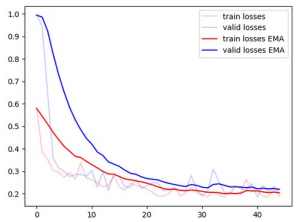


Figure 3: Loss using the default settings and test set of 2000 sampled points
accuracy: 52.2%

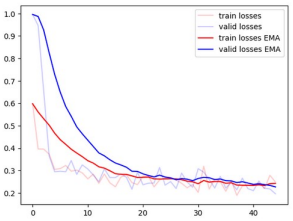


Figure 4: Loss using the default settings, increasing the radius to 0.04
accuracy: 61.1%

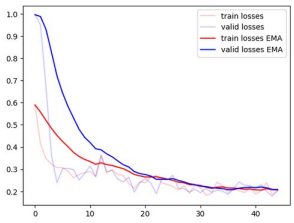


Figure 5: Loss using (38) keypoints sampled using open3D as test set
accuracy: 81.5%