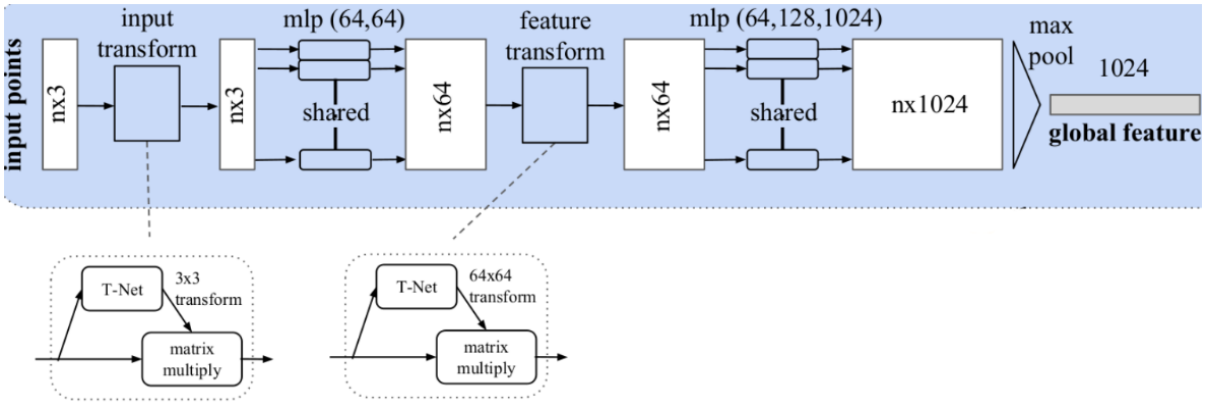
Assignment 4 - Deep 3D Descriptors

Introduction

This assignment involves implementing a simplified version of the PointNet (PN from now on) neural network architecture, called TinyPointNet (TPN from now on), to extract 3D local features from a point cloud.

Removing the classification head of PN (i.e. the last MLP layers), the proposed architecture can generate a global feature starting from the 3D input point.
To make it more manageable, the size of the output global feature is set to 256.



Implementation

The assignment requires three tasks:

- implementing the Triplet loss function: the network needs to learn to describe close points with "similar" vectors, and distant points with "dissimilar" vectors;
- sample generation: in order to compute the Triplet loss, the network needs to process both positive and negative samples with respect to an anchor point;
- defining TPN architecture, exploiting the PyTorch framework.

Triplet loss function

The Triplet loss function is defined as:

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

PyTorch provides the implementation, making it a one-liner:

```
tinypointnetloss = nn.TripletMarginLoss(margin=1.0, p=2)
```

the `margin` parameter corresponding to the α in the formula, and `p=2` meaning to use the L2-norm.

Sample generation

Starting from a randomly selected **anchor point**, a KD-Tree is used to define the neighborhood of points in the original point cloud within a given radius.

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

# find neighborhood of pt1
[k, idx, _] = pcd1_kdtree.search_radius_vector_3d(np.transpose(pt1), self.radius)
point_set1 = pcd1_points[idx[1:]]
```

A second KD-Tree is used to find the nearest neighbor of the anchor point in the second point cloud (the one affected by noise). This is considered the **positive sample**. The set of its neighbors within the radius defines the positive point set.

```
# POSITIVE: find corresponding point in pdc2
[_, pt2_idx, _] = pcd2_kdtree.search_knn_vector_3d(pt1, 1)
pt2 = pcd2_points[pt2_idx]

# find neighborhood of pt2
[k, idx, _] = pcd2_kdtree.search_radius_vector_3d(np.transpose(pt2), self.radius)
point_set2 = pcd2_points[idx[1:]]
```

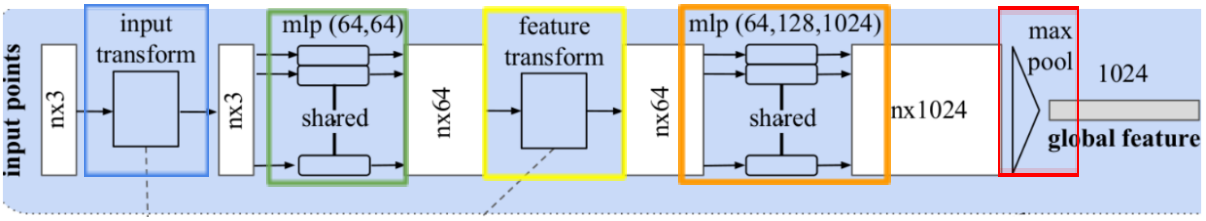
The **negative sample** is extracted from the noisy point cloud by making sure that it's distant enough from the anchor point. The corresponding point set is defined as explained above.

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

# find neighborhood of neg_pt
[k, idx, _] = pcd2_kdtree.search_radius_vector_3d(np.transpose(neg_pt), self.radius)
point_set3 = pcd2_points[idx[1:]]
```

The point sets are finally normalized by subtracting their originating point.

TinyPointNet Architecture



In the figure above, the main computational components of TPN are highlighted, and here the implementation is proposed:

- the input transformation is obtained by multiplying the input points by the (3x3) rotation matrix provided by the TNet subnetwork;
- the first MLP round is composed of two MLP layers of sizes (3 → 64) and (64 → 64) respectively. The weights are shared, meaning that a single instance of these layers processes all the input points. The outputs of this computation are the point features, not aligned;

```
point_features = self.mlp2(
    self.mlp1(input_transform_output)
    )
```

- the third computational block aligns the feature space by using again an instance of TNet, of size 64;

```
feature_transformation_matrix = self.feature_transform(
    point_features
    )
transformed_point_features = torch.bmm(
    torch.transpose(point_features,1,2),
    feature_transformation_matrix
    ).transpose(1,2)
```

- finally, the cloud features are generated by a round of three MLP layers of sizes (64 → 64), (64 → 128), and (128 → 256). From this, a MaxPool1d is applied to obtain the global features.

```
cloud_features = self.mlp5(
    self.mlp4(
        self.mlp3(transformed_point_features)
        )
    )
global_feature = self.max_pool(cloud_features)
```

Experiments and results

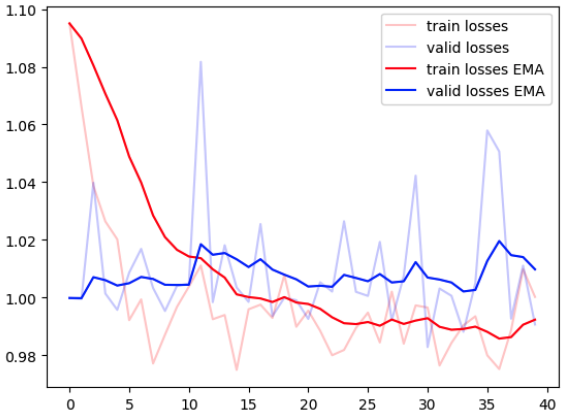
To obtain the final results, the following hyperparameters were tuned:

```
RADIUS = 3*10.0e-3
BATCH_SIZE = 45
SAMPLES_PER_EPOCH = 500
LEARNING_RATE = 0.01
EPOCHS = 40
PATIENCE = 6 # for early stopping
```

So, with respect to the initial parameters, the optimal radius is a little higher.
All other parameters have less impact on the result than the radius, but some tests were performed anyway.

Early-stopping was implemented in the training loop, to avoid wasting time when performances degrade. However, the system early-stopped just a few times over several runs.

The obtained training curve is the following:



and the trained model achieves an accuracy of 80.00% (on 100 samples) or 89.19% (on 37 samples).
One important thing to notice is that the resulting accuracy may vary in subsequent training instances for various reasons. One of these is the small size of the test points it is verified against. For this reason, the accuracies are reported both for a set of 37 samples and one of 100 samples (obtained by using the default parameters in the `compute_iss_keypoints` function, instead of overriding them).
Also to mention is that the accuracy measurement only takes into consideration the positive part of the descriptor generation task. That is, an anchor and a positive corresponding point, it checks that the generated descriptors are similar.
To more thoroughly test the system, further testing could be implemented by considering also the negative part of the task (different points are mapped into distant vectors).