3DP HW4

Juri Farruku

ID: 2157856

1 Sample generator

For the first part of the assignment we were asked to implement 3 different functions in order to sample the anchor point, a positive example, and a negative example.

In the function `sample_anchor_point` the anchor point is simply sampled from the list of points of the point cloud randomly, and it is returned along with its associated index.

For the positive example, through `sample_positive_point` we get the correspondent anchor point in the noisy point cloud and from there we get its neighborhood and return it, meanwhile in the `sample_negative_point` we randomly search in the noisy point cloud for a point which has from the anchor distance greater or equal than `min_dist`.

2 Netword building

In order to compose the **TinyPointNet** module, 5 multilayer perceptron were built, two for the first part of the net and three used in the second.

In the forward function the first 2 MLP layers, called `self.mlp_A_1 and self.mlp_A_2`, are applied and then the input is used to compute the 64 x 64 matrix used by PointNet to align points in feature space. Then the previously computed matrix is multiplied by the output of the first MLP, obtaining the input for the second MLP (whose layer are defined as aforementioned just changing the letter **A** with **B** in order to differentiate them, and then performing a pooling stage to obtain the desired feature.

3 SHOT canonical matrix and Loss functions

The weights used for the computation of the weighted covariance matrix are already given, so the computation can be easily done thanks to the `torch.einsum`

of pytorch, which takes as input the weights and the centered features and easily computes the needed matrix. This results in a quite fast implementation.
Then, using `torch.linalg.eigh` the eigenvalues and eigenvectors of the covariance matrix are computed; the eigenvectors, used for the shot frame, are then sorted in descending order and stored in a suitable variable.
The soft and standard triplet loss are then easily implemented thanks to torch: for the soft one we compute the distances of the anchor from both negative and positive examples thanks to `F.pairwise_distance` and then, instead of directly using the exponential function, it was decided to use the `F.softplus` for numerical stability. The standard loss has been almost implemented in the same way, the only difference in the last step, which was computed through the `F.relu` function.

4 Results

All results obtained through different runs, with the standard parameters, can be seen in table 1. From the table we can notice that using SHOT instead of the classical TNet for rotation alignment degrades the performances of the network of about 30%. This is probably because SHOT is hard computed and is not learned as TNet, and so it is not able to grasp the correct geometry of the point cloud.

Another interesting thing is that using Soft Triplet Loss instead of the classical one gives us better results. Also, from Figure 1 and Figure 2 it can be seen that both initial training and validation losses in the soft loss are significantly higher than the standard one. This could lead us to think that the soft one is less aggressive than the standard one, converging in a smoother way. This could also lead not to overfit the training data and obtaining thus a higher accuracy. Neither of the losses has a significantly faster convergence than the other.

Two hyperparameters have been modified: the **learning rate** and the **radius** of the neighborhood of the samples. Results obtained changing the learning rate from 0.0005 to 0.0001 can be seen in 2. All the different configurations benefit from this change, showing a higher accuracy despite losing a little bit in training speed. Indeed with this change the velocity moved from 1.51 it/s to 1.25 it/s, but the increase in accuracy is quite good, especially in the SHOT configurations, increasing by more than 3%.
Regarding the radius, configurations have been tested both increasing and decreasing it, using as values 0.01 and 0.03, as shown in 3 and 4. From these we can easily see that a smaller radius drags the Net towards worse results, especially in the SHOT configurations, meanwhile a bigger radius, and thus a bigger neighborhood, improves sensitively the results both for the SHOT configurations and the TNet with hard and soft loss, obtaining astonishing accuracy.

Initial Alignment	Hard Triplet Loss	Soft Triplet Loss
TNet	93.9%	96.35%
SHOT	61.4%	

Table 1: Different accuracy values of TinyPointNet using standard hyperparameters

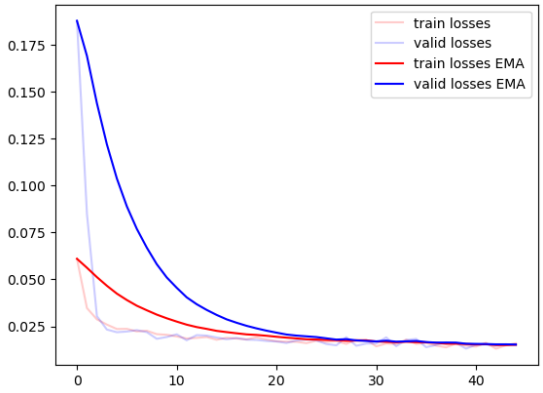

Figure 1: Loss curve for training and validation using Standard triplet loss

Initial Alignment	Hard Triplet Loss	Soft Triplet Loss
TNet	95.9%	97.15%
SHOT	64.7%	

Table 2: Different accuracy values of TinyPointNet using a learning rate of 0.0001

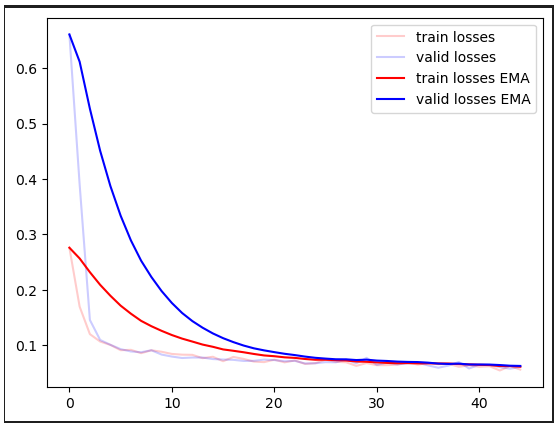

Figure 2: Loss curve for training and validation using Soft triplet loss

Initial Alignment	Hard Triplet Loss	Soft Triplet Loss
TNet	93.2%	95.34%
SHOT	52.450%	

Table 3: Different accuracy values of TinyPointNet using a radius of 0.01

Initial Alignment	Hard Triplet Loss	Soft Triplet Loss
TNet	97.850%	98.650%
SHOT	65.45%	

Table 4: Different accuracy values of TinyPointNet using a radius of 0.03