SE3-Unet - Visione Artificiale

Mattia Ferraretto [00072A]

Academic Year 2023/2024

Contents

1 Problem statement

The project consists in developing a novel neural network architecture with the aim to address a point cloud segmentation problem. The idea is to replace the classic convolution block in U-net architecture [1] with a SE3-Transformer block [2], in such a way to make it resilient to 3D roto-translation. Then, the network is trained on FaceScape [3], a large-scale detailed 3D face dataset, with the objective of predicting whether a point belongs to a specific face landmark.

2 Introduction

Before diving into SE3-Transformer let me introduce two fundamental concepts: Equivariance and Group Representation.

Equivariance Give a set of transformation $T_g : \mathcal{V} \to \mathcal{V}$ for $g \in G$ where G is an abstract group, a function $\phi : \mathcal{V} \to \mathcal{Y}$ is called equivariant if for every g there exist a transformations $S_g : \mathcal{Y} \to \mathcal{Y}$ such that

$$S_g\left[\phi(v)\right] = \phi(T_g\left[v\right]) \qquad \text{for all } \; g \in G, \, v \in \mathcal{V} \tag{1}$$

Where g can be seen as parameters describing the transformation. In the equivariance literature, deep networks are built from interleaved linear maps ϕ and equivariant nonlinearities. In the case of 3D roto-translations it has already been shown that a suitable structure for ϕ is a *tensor filed network*.

Group Representations In general, the transformation (T_g, S_g) are called *group representations*. Formally, a group representation $\rho : G \to GL(N)$ is a map from a group G to the set of $N \times N$ invertible matrices $GL(N)$. In the case of 3D rotations $G = SO(3)$ we have two interesting properties: 1) its representations are orthogonal matrices, 2) all representations can be decomposed as

$$\rho(g) = Q^T \left[\bigoplus_{\ell} \mathbf{D}_\ell(g)\right] Q \tag{2}$$

where Q is an orthogonal, $N \times N$, change-of-basis matrix; each $\mathbf{D}_\ell$ for $\ell = 0, 1, 2, \ldots$ is a $(2\ell + 1) \times (2\ell + 1)$ matrix known as a Wigner-D matrix; and the $\bigoplus$ is the *direct sum* or concatenation of matrices along the diagonal. Wigner-D matrices are *irreducible representations* of SO(3). Vectors transforming according to $\mathbf{D}_\ell$, are called type-ℓ vectors. Type-0 are invariant under rotations and type-1 vectors rotate according to 3D rotation matrices.[1]

3 SE3-Transformer

In the SE3-Transformer, the concept of **neighbourhoods** is introduced. Given a point cloud $\{(\mathbf{x_i}, \mathbf{f_i})\}$, the idea is to define a set of neighborhoods $\mathcal{N}_i \subseteq \{1, \ldots, N\}$ centered on each point i. These neighbourhoods are computed either via the nearest-neighbours methods or may already be defined. The introduction of this neighbourhoods is justified by the reduced cost of computing the attention mechanism since it's quadratic with respect to the input.

[1]For more details look at the original paper. [2]

Architecture The SE(3)-Transformer itself consist of three components: 1) edge-wise attention weights α_{ij}, constructed to be SE(3)-invariant on each edge ij; 2) edge-wise SE(3)-equivariant value messages, propagating information between nodes (as in Tensor Field Network); 3) a linear/attentive self-interaction layer. Attention is performed on a per-neighbourhood basis as follow:

$$\mathbf{f}^{\ell}_{\text{out},i} = \underbrace{\mathbf{W}^{\ell\ell}_{V}\mathbf{f}^{\ell}_{\text{in},i}}_{③\text{ self-interaction}} + \sum_{k\geq 0}\sum_{j\in\mathcal{N}_i\setminus i} \underbrace{\alpha_{ij}}_{①\text{ attention}} \underbrace{\mathbf{W}^{\ell k}_{V}\left(\mathbf{x}_j - \mathbf{x}_i\right)\mathbf{f}^{k}_{\text{in},j}}_{②\text{ value message}} \tag{3}$$

In this equation, if we remove the attention weights then we have a tensor field convolution, and if we instead remove the dependence of $\mathbf{W}_V$ on $(\mathbf{x}_j - \mathbf{x}_i)$, we have a conventional attention mechanism. Provided that the attention weights α_{ij} are invariant, Eq. (3) is equivariant to SE(3)-transformations. Invariant attention weights can be computed with a dot-product attention structure shown in Eq. (4). This mechanism consist of a normalised inner product between a query vector $\mathbf{q}_i$ at node i and a set of key vectors $\{\mathbf{k}_{ij}\}_{j\in\mathcal{N}_i}$ along each edge ij in the neighbourhood $\mathcal{N}_i$ where

$$\alpha_{ij} = \frac{\exp(\mathbf{q}_i^{\top}\mathbf{k}_{ij})}{\sum_{j'\in\mathcal{N}_i\setminus i}\exp(\mathbf{q}_i^{\top}\mathbf{k}_{ij'})}, \quad \mathbf{q}_i = \bigoplus_{\ell\geq 0}\sum_{k\geq 0}\mathbf{W}^{\ell k}_{Q}\mathbf{f}^{k}_{\text{in},i}, \quad \mathbf{k}_{ij} = \bigoplus_{\ell\geq 0}\sum_{k\geq 0}\mathbf{W}^{\ell k}_{K}(\mathbf{x}_j - \mathbf{x}_i)\mathbf{f}^{k}_{\text{in},j} \tag{4}$$

$\bigoplus$ is the direct sum, i.e. vector concatenation in this instance. The linear embedding matrices $\mathbf{W}^{\ell k}_{Q}$ and $\mathbf{W}^{\ell k}_{K}(\mathbf{x}_j - \mathbf{x}_i)$ are TFN type. The attention weights α_{ij} are invariant for the following reason: when input features $\{\mathbf{f}_{in,j}\}$ are SO(3)-equivariant, then the query $\mathbf{q}_i$ and key vectors $\{\mathbf{k}_{ij}\}$ are also SE(3)-equivariant, since the linear embeddings matrices are of TFN type.[2]

4 Dataset

In this section, I'm going to introduce the dataset used for this work, the pre-processing phase, and finally, the resolution reduction technique applied to the data.

Facescape The dataset used for this work is called Fascescape. It consists of a collection of 18,760 high resolution 3D faces with 20 different expressions. Each face has been acquired by using a multi-view 3D reconstruction system composed by 68 DSLR cameras, 30 of which capture 8K images focusing on front side, and the other cameras capture 4K level images for the side parts. Subjects scanned are people between 16 and 70 years old and mostly from Asia. Additional information for each subject are recorded such as gender and job (by voluntary). Along with the dataset a set of landmark points are provided, each landmark identify a specific portion of a face. I focused my work only on a subset of faces (669 for the training set and 167 for the test set) and a single expression: Neutral. Then, the dataset has been pre-processed.

Pre-processing In this study the original dataset has been pre-processed by two different approaches: 1) by applying random 3D roto-translations and then re-aligning data using ICP; 2) by simply applying random 3D roto-tranaslation. The main objective of this kind of pre-process is to simulate a scenario in the wild where data are not acquired in a controlled environment. The second point is about to evaluate the capacity of the SE(3)-Transformer to be resilient to rotation and translation in the field of 3D data compared to other approaches.

[2]For additional detail read the original paper [2]

Point cloud resolution reduction Since FaceScape data are point clouds having a resolution 8192 points, due to the limited compute available they have been reduced to a manageable resolution (test were performed with 2048 points). To reduce the resolution the idea is to apply Farthest Point Sampling (FPS) algorithm which aims to select a representative subset of point from the original point cloud. Along with the point cloud reduction process, even each heatmap associated has been recomputed. In particular, with respect to each landmark i:

$$h_i = \exp\left(-\frac{d(x_i,\, x_j)^2}{2\sigma^2}\right) \tag{5}$$

where d is the Euclidean distance squared, x_i is the 3D coordinates of i-th landmark, x_j correspond to a point in the point cloud, and in the end, σ is a parameter shaping the Gaussian's bell. Repeating this transformation for each point, the resulting h_i is a vector of shape $1 \times N$, where N is the number of points, corresponding to the heatmap for the landmark. In my setting, σ is set to 0.08.

5 3D - Pooling

In the classic CNN framework the pooling operation consists in replacing the value at certain location with a summary statistic of nearby values. In the literature the are different types of pooling functions, such as Max Pooling, Average Pooling, and they work well in grid-like data (e.g. 2D images) where a neighbourhood is clearly defined. However, 3D data has not a fixed structure and a pooling operation must take into account such fact for working correctly in this scenarios. For simulate traditional pooling my idea is to select a subset of points using FPS according to a specific pooling ratio, aggregate features based on their neighbourhoods (neighbourhood can be defined by k-nn methods or already pre-computed), and finally discard points not selected by FPS. The resulting point cloud will be one with a resolution reduced by the pooling ratio and features aggregated in the maintained points. Of course, also in that case, different kind of pooling can be applied (Max, Average ecc..). In the code base provide, Max and Average are supported, as well as, multi-channel pooling.

6 3D - Up sampling

Inverse pooling operation in CNN is called transpose convolution where a pooled input is reported to the original resolution with the help of a learnable kernel. To replicate this operation in a 3D setting my idea is to estimate features by applying a technique called Inverse Distance Weighting (IDW). IDW assumes that the influence of a known feature decreases as the distance to the unsampled location increases. Therefore, the weights assigned to each sampled feature are inversely proportional to their distance from the unsampled location. Formally speaking:

1. The first step consist in computing the weights:

$$w_j = \frac{1}{d(x_i,\, x_j)^p} \tag{6}$$

   where d is the Euclidean distance between x_i and x_j, x_i is the location in which we are interested in estimating the features, x_j is a location of a data point belonging to the neighbourhood of x_i, and p is the power parameter controlling how rapidly the influence of a point decreases as the distance increases.

2. Once the weights are computed, the features for x_i are estimated as the weighted average of its neighbourhood:
$$f_i = \frac{\sum_{j \in \mathcal{N}_i} w_j f_j}{\sum_{j \in \mathcal{N}_i} w_j} \tag{7}$$
where w_j is the weight associated to f_j and denominator is a normalization term.

To make this approach suitable to the problem setting, on top of point clouds a knn-graph has been built based on the distances of points, then during the forward pass of the network the structure of the graph has been kept at each level (for more datails see the next section).

7 Model architecture

As mentioned in Section (1) the architecture defined is similar to a U-net where instead to have classic convolutional layers SE(3)-Transformer layers are used in place.

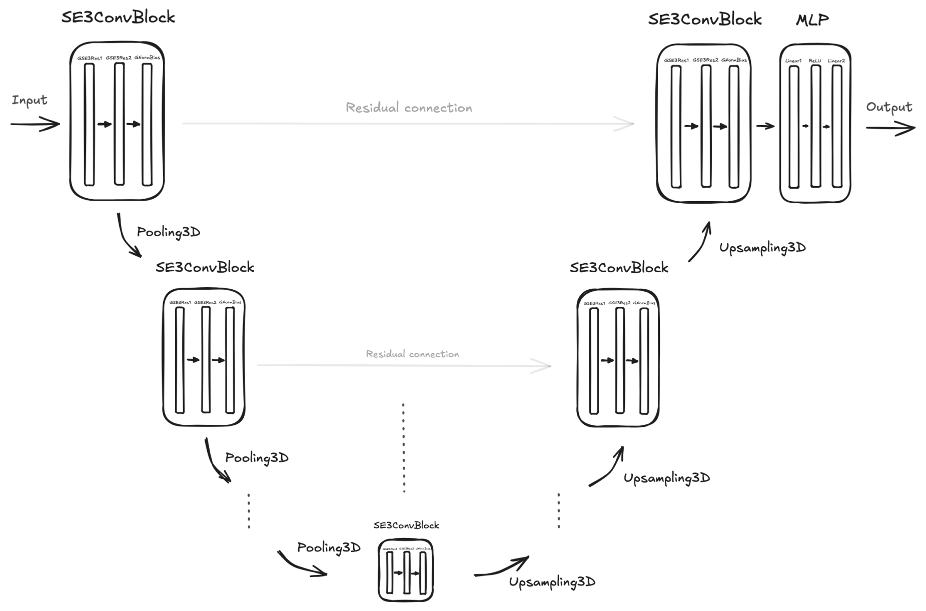


Figure 1: SE(3)-Unet architecture followed by a classification head.

U-net U-net architecture consists of a contracting path (left side) and an expansive path (right side). The contracting path follows the typical architecture of a convolutional network. It consists of the repeated application of two 3 convolution (unpadded convolutions), each followed by a rectified linear unit (ReLU) and 2×2 max pooling operation with stride 2 for downsampling. At each downsampling step authors double the numbers of feature channels. Every step in the expansive path consists of an upsampling of the feature map followed by a 2×2 convolution that halves the number of features channels, a concatenation with the correspondingly cropped features map from the contracting path, and two 3×3 convolution, each followed by a ReLU. At the final layer a 1×1 convolution is applied.[1]

SE(3)-Unet Similarly to the original U-net, SE(3)-Unet has a contracting path (left side) and an expansive path (right side). In this architecture, the contracting path consists of a repeated application of a SE3ConvBlock, where it composed by two GSE3Res (Graph attention block with SE(3)-equivariance and skip connection) followed by a GNormBias (Norm-based SE(3)-equivariant nonlinearity with only learned biases), and a Pooling3D operation with a specified polling ratio. As a strong difference with respect to the the original network is that the number of channel is fixed to 5. Every step in the expansive path consists in a Upsampling3D operation followed by the application of a SE3ConvBlock. At each up sampling step, after the SE3ConvBlock layer transformation, residual connections coming from the contracting path are summed. Finally, features are mapped to the target classes using an MLP composed as a linear layer, followed by a ReLU activation, followed by another linear layer.

8 Experiments and results

In this section I'm going to describe the two experiments performed and results obtained, but first let me introduce the environment. Experiments have been made in Amazon Sagemaker Studio Lab free tier. It offers a CPU 4-core with 12 GB of RAM and a single Nvidia T4 with 16 GB of VRAM. A Python virtual environment has been defined using Conda where libraries required to run the code have been installed. In details, the most important libraries and their respective versions are: python v3.11.9, dgl v2.4.0+cu118, torch v2.1.0, torch-geometric v2.5.3, open3d v0.18.0.

Since the challenging optimization problem, due to the extreme nature of classes imbalance, I tried two different optimization approaches by using two different loss functions: **Binary Cross Entropy** and **Focal Loss**.

BCE Binary Cross Entropy (BCE) is a classic loss used when the problem consist in discriminate two classes (usually 1 and 0). It is defined as:

$$\ell(x_n, y_n) = y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n))$$

where n it the n-th element of the dataset and σ is the logistic function. In Pytorch a more sophisticated version is available, called `BEWithLogitsLoss`, where there is the possibility to define weights to positive classes, in such away to gain the gap between them.

FL Focal loss is designed to address extreme classes imbalance. It is similar to the BCE loss whit some differences, in particular, 2 parameters have been introduced: α and γ. α gives a focus on class imbalance while γ, also called focus parameter, focusing the model to examples hard to classify. Formally speaking:

$$FL(p_t) = -\alpha(1 - p_t)^\gamma \log(p_t)$$

where $p \in [0, 1]$ usually defined through a sigmoid function [4].

SE(3)-Unet configuration details In both experiments SE(3)-Unet has been defined with a single layer, 5 hidden channels, a self-interaction in hidden layers of type 1×1 (Graph Linear SE(3)-equivariant layer, equivalent to a 1x1 convolution), a self-attentive interaction in the final layer, a Max pooling operation with a ratio of 0.1, and finally, an output map of 68 features (I used a single layer due to the limited compute available in the Amazon free tier, while I chose 5 hidden channels according to [2] in the point cloud segmentation experiment).

Dataset configuration details Also the dataset (train set and test set) in both experiments has been pre-processed in the same way. In particular, each point cloud has been reduced to 2048 points as described in Section 4. Afterwards, 3D spatial transformations have been applied (introducing random translations and rotations) and then realigned using ICP, on top of each point clouds a k-nn graph has been built setting k equals to 40 (according to [2]), in the end, heatmaps have been transformed in binary labels according to a threshold set to 0.5 (when $h_{ij} > 0.5$ then the i-th point belongs to the j-th landmark).

Evaluation metrics To test the model performance I decided to use the classic metrics: Precision, Recall, F_1 score, and Precision-Recall curve. Each metric has been computed at different classification thresholds, ranging from 0 to 1 with a step of 0.1. Since the objective is to evaluate the model performance over a given input, output classes have been aggregated during the computation of the confusion matrix (applying a micro aggregation approach). The model's performance is finally obtained by averaging overall inputs (macro aggregation approach). Clearly, metrics have been computed over the test set, and over both dataset pre-process approaches proposed.

8.1 Experiment 1: optimization with BCEWithLogitsLoss

In the first experiment the model has been trained by using the classic Binary Cross Entropy with logits. In this section I'll explain hyperparameters defined and evidence discovered.

Let's begin by analyzing hyperparameters defined. In the table below you find their detailed values:

epochs	**lr**	**lr-sched.**	**batch size**	**opt.**	**grad. acc.**
10	0.00003	CosineAnWR	2	Adam	1

Table 1: Hyperparameters defined for the training (experiment 1).

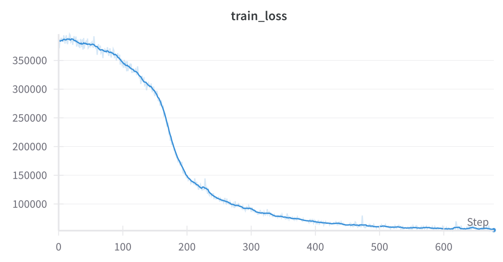
The network has been trained for 10 epochs, the learning rate defined is 3e-5 with a bach size of 2. As you can imagine, this values are due to the limited amount of hardware resources. The scheduler introduces is the CosineAneling with warm restart. Gradients accumulation steps are set to 1.

Since point clouds are highly imbalanced, using the simple BCE is not enough, and as result, the network is biased toward the majority class. To address this problem, BCEWithLogistLoss available on Pytorch allows to introduce a so called `pos_weight` with the objective to balance minority classes. There are many possible ways to compute this values, in particular for this experiment I computed `pos_weights` as follow:

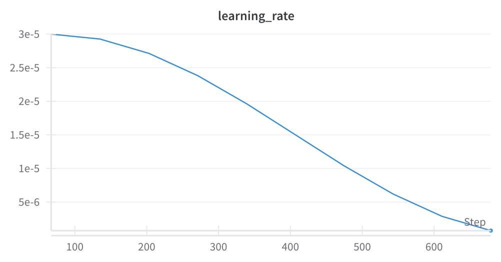
$$pos\_weight_j = \frac{N}{\sum_{i=0}^{N} 1 \cdot \mathbb{I}\{y_i = 1\}} \tag{8}$$

where N are the total number of labels in the training set, j correspond to the j-th class, $\mathbb{I}$ is the indicator function.

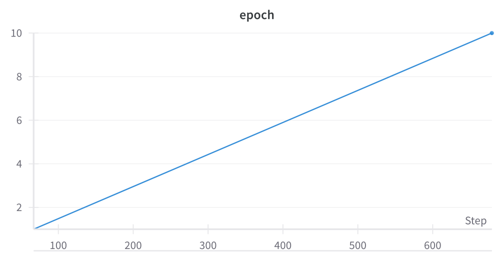
In the next Figure (2), you see the curves obtained during the training phase with hyperparameters described before. The number of total steps is over 600, the learning rate decreases smoothly according to the scheduler, but the more interesting curves are the ones related to training loss and validation loss. In particular we can observe that both curves have high value at the beginning, then they decreases quickly. Around 500 step they start to stabilize (but still decreasing) with a roughly error of 60,000. A so large error is due to sum aggregation method used, in place by average. This approach is simply motivated by the idea to keep larger gradients and redistributing a larger error during backward steps.



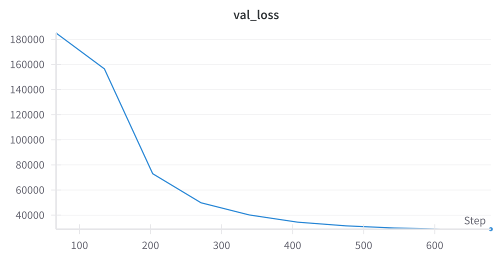
(a) Training loss



(b) Learning rate



(c) Epochs



(d) Validation loss

Figure 2: Training metrics (experiment 1)

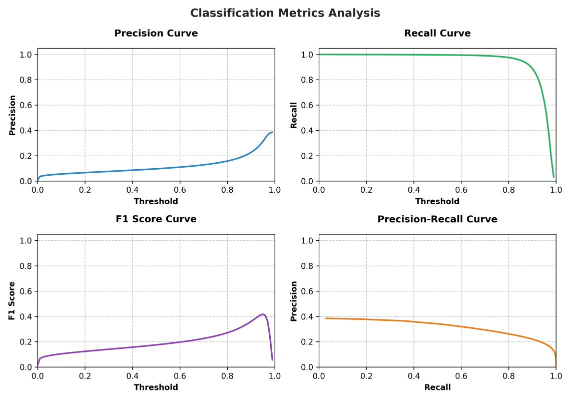
One may say: well training loss is decreasing, on the same way also the validation loss, so everything sound good. But before converging to wrong conclusion, let me takes a look to the testing results!

8.1.1 Test results

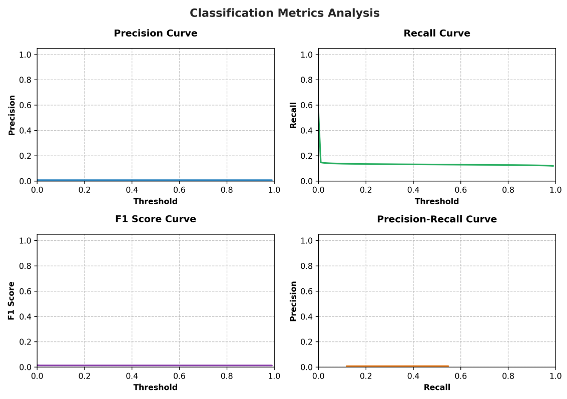
Here you find the results obtained in the first experiment where the model has been trained using BCE loss.

In Figure 3 on your right you find charts related to the model performance over a test set pre-processed by using ICP approach (3a). The precision curve rises as thresholds growth, and this is an expected behavior considered its definition ($tp/(tp + fp)$). However in a healthy system we'll expect a faster growth reaching values close to 1, but can be easily seen from the chart that it reaches its peak at 0.4 when threshold is close to 1. On the other hand, the recall curve reflects an expected one by its definition ($tp/(tp + fn)$), since there are no *false negative* at the beginning and then they increase as thresholds growth pushing the curve toward zero. Then, observing F_1 score curve it reflects both curves (precision and recall) well with a slower growth, a peak reached when threshold is close to 1, and then a very fast dump. Finally, the precision-recall curve synthesize in very impactful way the model performance definitively confirming its poor results against the task for which it was trained.

Analyzing the left side of Figure 3, sincerely, no words are needed. This graphs are obtained by running the test script over the test set pre-processed by applying the spatial transformation



(a) ICP



(b) Spatial transformation

Figure 3: Classification results using Binary Cross Entropy

approach. Since the model has been trained over a training set pre-processed by using ICP, a drop in performance was excepted, but graphs are clearly speaking, when the model is processing data transformed in a different way, it is not longer able to handle them in a sene manner.

8.2 Experiment 2: optimization with Focal loss

In the second experiment performed, the network has been optimized by using a custom implementation of the Focal Loss described in the previous paragraph. Here I'm going to discuss hyperparameters settings and results obtained from this experiment.

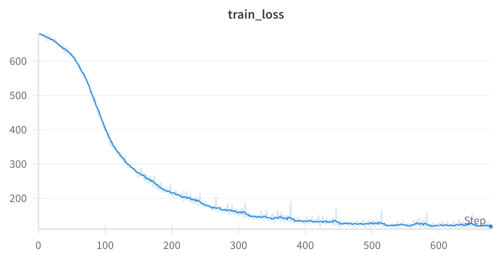
Let's start by talking about hyperparameters. In the following table you find details about the most important ones:

epochs	**lr**	**lr-sched.**	**batch size**	**opt.**	**grad. acc.**	α	γ
10	0.00003	CosineAnWR	2	Adam	1	0.98	3

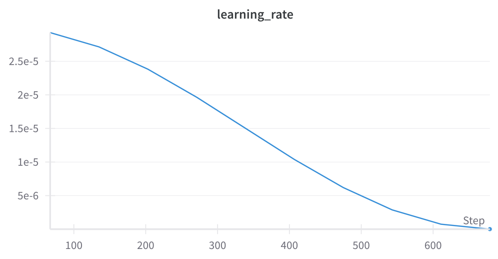
Table 2: Hyperparameters defined for the training (experiment 2).

I set number of epochs to 10 due to the limited amount of compute time available. I started the training with a learning rate relatively low compared to the one used in the segmentation task in [2]. Then using a CosineAneling with warm restart, learning rate has been reduced gradually until has reached η_{min} set to $1e-8$ (in that setting the maximum number of iterations for the scheduler is equals to the number of epochs, as a consequence, no warm restart has been occurred). As you can notice to make a fair comparison between the two experiments, hyperparameters are practically kept identical. The only difference introduced are parameters necessary for the Focal Loss which are 0.98 for α, used to give importance to the minority class and since points clouds are highly imbalanced its value is very high close to 1, and γ, also known as the focus parameter allowing the model to concentrate to difficult samples, set to 3. Of course I tried different values for both and this pair might seem the optimal one for the data.

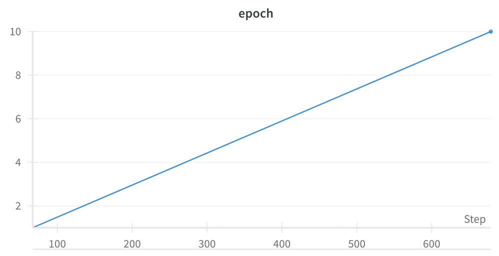
In the Figure 4 you see metrics kept under observation during the training. The total number of steps performed are over than 600 (4c), the learning rate perfectly follows the scheduler (4b), but the most interesting graphs are the ones about training loss (4a) and eval loss (4d). The training loss at the beginning is very large (roughly 679), but this big value is due to the aggregation method used, sum instead of mean. This choice is justified by the idea to redistribute a larger error during back-propagation step while maintaining larger gradients (using mean I noticed a very low loss with ”limited effectiveness”). Observing the training loss curve we can deduce that training is stable, it decreases at the beginning very quickly then in the ending phase it decreases slower stabilizing around 118. Talking about evaluation loss, it is performed at the end of each epoch, also in this case at the beginning the eval loss is large, then, during the optimization phase decreases well. In conclusion, observing the curves, we can state that training was successful. Now, let’s see the performance reached by the resulting model in the test phase!



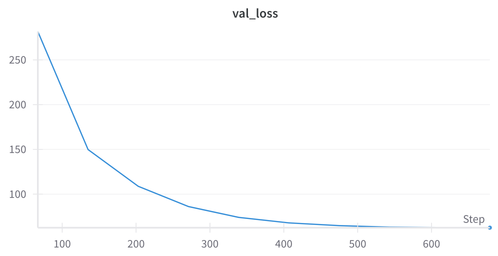
(a) Training loss



(b) Learning rate



(c) Epochs

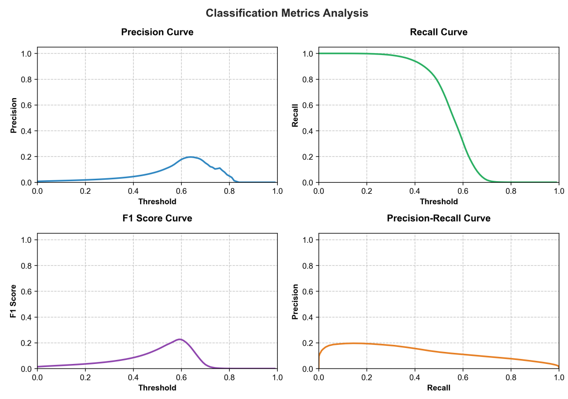


(d) Validation loss

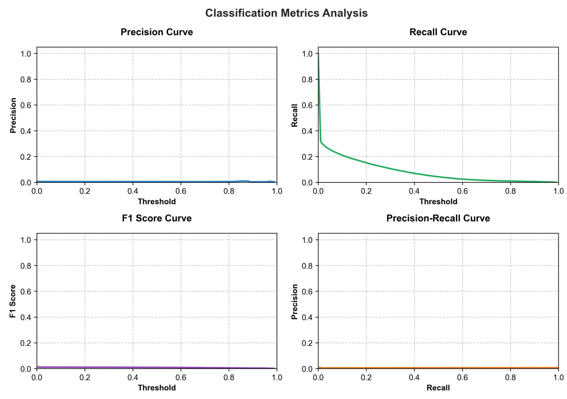
Figure 4: Training metrics (experiment 2)

8.2.1 Test results

In this section I'm going to comment results obtained by the model optimized using Focal Loss. In the Figure below you can see on your right (5a) results obtained over ICP, on your left (5b) results over a simple spatial transformation of the data:



(a) ICP



(b) Spatial transformation

Figure 5: Classification results using Focal Loss

Observing the right charts, you can see the precision curve pretty flat, with a smaller spike in the range [0.4, 0.8]. Such a curve suggests that the model confuses many negative points with positive ones. Looking at the recall curve, on the contrary, we see a well-defined one with a value of 1 in the range $[0, 0.3]$ and then it starts to decrease until it reaches 0. This behavior can be considered expected since at the increasing of the thresholds the number of false negative increases pushing the recall score toward zero. Talking about F_1 score we can state that the model reaches its maximum classification performance over the 0.5 threshold, for the rest, it reflects well the poor performance of the precision. Finally, we have precision-recall curve, which basically confirms the poor performance reached by the model since its area under the curve is closer to zero than to one.

Now, what happen when the dataset is broken by applying 3D random rotation and translation, and the model is trained over a training set pre-processed with ICP? Well, graphs on your left talk by themselves! In general, can be observed a strong performance degradation, where firstly is noticeable by the precision, F_1 score, and the precision-recall curves that are completely flat, secondly by the recall curve that decreases very quickly.

9 Conclusion

In conclusion I can state that the model is not working well at all. Both optimization approach are resulting very ineffective, and I think the reason can be associated to two facts: 1) the nature of data, in particular I'm referring to the high imbalance between positive and negative samples in each classes; 2) the limited amount of time dedicated to the training phase, due to the poor hardware resources available. Honestly another possible reason can be related to the novel architecture proposed here, maybe it is not able to capture the complexity of the data. On the other side, talking about the performance when data are transformed according to a different pre-process approach applied during the training, the expectation of finding a model resilient to rotation and translation in 3D field was not confirmed, since that when test data are treated in the same way poor performance are observed in both optimization approach, while when data are broken no considerable results are depicted. Also in this case reasons can be many, but in my opinion this result is mainly due to the kind of architecture and its dimension (only a single "hidden layer").

10 Further improvements

There are many changes that can be made and that can lead to performance improvements. In this section I'm going to list many of them:

- Training the network for more than 10 epochs, maybe 20 or also 30 can help to converge in the right direction resulting in better model performance;
- Changing the classification threshold during the training to lower value, reducing the gap between positive and negative samples.
- Changing σ with a larger value than 0.08 when re-computing the heatmap, obtaining a smoother Gaussian curve, and as consequence a lower gap between positive and negative samples;
- Varying the pooling ratio used in the down sampling branch of the network;
- Using a smoother pooling function such as 3D Average Pooling instead of Max Pooling;
- Using a totally different approach for pooling, as well as a totally different upsampling approach;
- Increasing the model size by using more than one single layer;
- Testing other different loss functions, defining some particular one able to address the optimization problem.

References

[1] U-Net: Convolutional Networks for Biomedical Image Segmentation, Olaf Ronneberger, Philipp Fischer, and Thomas Brox; https://arxiv.org/pdf/1505.04597

[2] SE(3)-Transformers: 3D Roto-Translation Equivariant Attention Networks, Fabian B. Fuchs, Daniel E. Worrall, Volker Fischer, Max Welling; https://arxiv.org/pdf/2006.10503

[3] FaceScape: a Large-scale High Quality 3D Face Dataset and Detailed Riggable 3D Face Prediction, Haotian Yang, Hao Zhu, Yanru Wang, Mingkai Huang, Qiu Shen, Ruigang Yang, Xun Cao; https://arxiv.org/pdf/2003.13989

[4] Focal Loss for Dense Object Detection, Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, Piotr Dollar; https://arxiv.org/pdf/1708.02002