Point Transformer based Auto-Encoder for Robot Grasping

BRINGER Leo
Department of Mechanical Engineering
University of Michigan
Ann Arbor, MI
lbringer@umich.edu

***Abstract*—Auto-encoders are artificial neural networks that learn how to efficiently compress and encode data into a smaller but meaningful latent-space and then learns how to decode it back such that the reconstructed input is as similar as possible to the original one. In Point cloud processing such algorithms can be very useful to interpret collected data in many different types of robotics application such as digital twins, autonomous vehicles, augmented and virtual reality or many other industrial purposes. However, point clouds are unordered, irregular and unstructured set of points and are thus particularly challenging to process with conventional neural network operation. In this paper we will study how transformer blocks that are set operators are particularly well suited for point clouds and thus make auto-encoders very efficient on point clouds, conserving their permutation invariance and study the application of such algorithms on an application for robot grasping.**

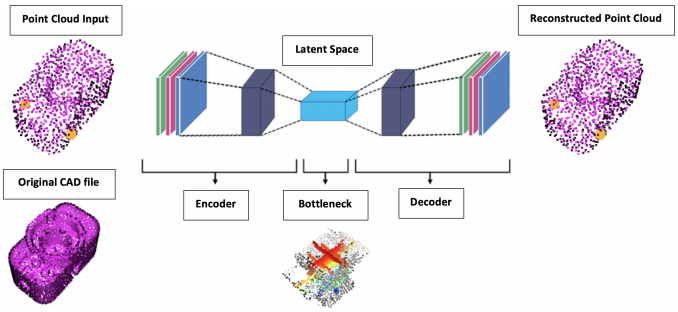


Fig. 1. The Point Transformer based Auto-Encoder encodes the information of the surfaces from a point cloud representing a CAD file in a more meaningful latent space useful to determine grasping point candidates and then reconstruct it back to its initial structure.

I. Introduction

3D point cloud processing and understanding are considerably more challenging than 2D images mainly due to the fact that point cloud samples are irregularly structured while 2D image samples (pixels) rely on a structured grid in the image plane with a regular spacing. Indeed, most conventional operators such as Convolutional Neural Network (CNN) requires data to be ordered, regular and on a structured grid. Hence, to use CNNs early approaches preprocessed the data into structured grid formats such as voxels. However, this method suffers from high memory consumption because it makes the data unnecessarily voluminous and sparse, while limiting its resolution thus also resulting in a loss of information. Moreover, since point sets are unordered they require the neural network to be permutation invariant and robust against rotation and translation to operate in the metric space. Also, since the points usually describe objects, the network needs to capture the spatial relations between them.

A pioneer solution that came up in 2017 by Qi et al. called PointNet[2], only used multi-layer perceptrons (MLPs), max-pooling and rigid transformations to ensure invariance under permutations and rotation by using only elementary binary symmetric operations and aggregation operations treating each point separately. This approach obtained a lot of the support from the community and has gathered an increasing interest in addressing the challenge. Although by treating each point separately by design PointNet does not capture local structures, limiting its ability to recognize fine-grained patterns.

In this paper, our point cloud auto-encoder tackles these issues through the use of a specific neural network structure inspired by the success of transformers [4] in natural language processing and image analysis. The transformer family of models is particularly appropriate for point cloud processing because the self-attention operator, which is at the core of transformer blocks, is in essence a set operator: it is invariant to permutation and cardinality of the input elements. The application of self-attention to 3D point clouds is therefore quite natural, since point clouds are essentially sets embedded in 3D space.

The self-attention module generates refined attention feature for its input based on global context. Whereas PointNet[2] fails to consider local structures, the self-attention layer takes as input the sum of input embedding and positional encoding and the latter allows the operator to adapt to local structure in the data. Finally, all operations of Transformer are parallelizable and order-independent. Therefore, in theory, it should be able to replace the convolution operation while having a better versatility.

In this specific study we also present a potential application of point cloud auto-encoders for robot grasping. A Contact Point Pair Estimator (CPPE) for 6DOF grasp detection is used upstream of our algorithm. The former takes as input a CAD file of a target object to grasp, performs Object Pose Estimation to obtain its position and orientation, infers a point cloud of the represented object and then, through down

sampling and clustering performs a grasp synthesis to reduce the number of grasp candidates. Fig 8 resumes this method. Those grasp candidates which are potential surfaces for the gripper to grasp the object is the input of our algorithm. Our unsupervised neural network is suppose to learn how to efficiently compress and encode this data into a smaller latent-space that allows to select only a handful of points (forming surfaces), according to some quality factors, that the gripper could use to perform robot grasping, see Fig 1. Finally, the auto-encoder learns how to reconstruct the data back from the reduced encoded representation to a representation that is as close to the original input as possible. An auto-encoder, by design, reduces data dimensions by learning how to ignore the noise in the data. It could be define as feature extraction algorithm, its main purpose is to learn useful information from the more expressive recreated bottleneck that is commonly used to perform many different types of downstream tasks.

II. Implementation

A. Self-Attention Layer

Originally attention functions have been introduced as an improvement from Residual Neural Networks (RNN) that struggle to deal with long-range dependencies in sequence to sequence transduction tasks. Indeed, in RNNs, the input sequence is bottlenecked through a fixed-size vector called the context vector which results in a significant loss of information for long-range input sequences and a lack of consideration to local features. So this new idea of attention function introduced in the famous paper "Attention is all you need"[4], uses a new context vector at each step of the output sequence to "look at" different parts of the input sequence and consider local features.

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. Usually both the keys and the values are computed from the input. We compute the alignment between the keys and the query through a compatibility function (usually a weighted dot product) and then we normalize those results to obtain attention weights. Finally, we obtain the output by computing a linear combination of the values and the attention weights. This concept is resumed on Fig 2.

In a self-attention layer all of the keys, values and queries come from the same place which is the output of the previous layer. This technique allows to connect each set of the output to all the sets from the previous layer while keeping the permutation invariance.An example of a self-attention module is described on Fig 3.

Altough, self-attention operators can be classified into two types: scalar attention and vector attention. The type that we have just presented is closer to the scalar attention operators because the set of input are feature vectors that we apply a standard scalar dot-product on.

In vector attention, the computation of attention weights is a little bit different. In particular, attention weights are vectors that can modulate individual feature channels. In this case, the

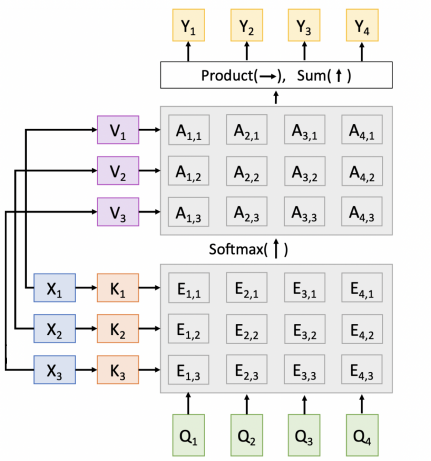


Fig. 2. the concept of attention functions.

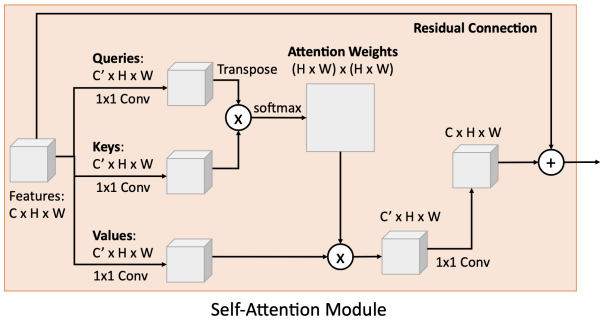


Fig. 3. Self-Attention layers of the Transformer Block.

compatibility function is different. Instead of computing the dot-product of the keys and query we apply linear functions on them separately and the apply a symmetric relation function (in our case we subtract the result). Afterwards, we apply a mapping function such as an MLP (multi-layer perceptron) on the results (in our case it will be two linear layers and one ReLU non-linearity). However, in both case (scalar or vector) self-attention functions remain set operators.

B. Positional encoding

As we said earlier, the main issue with PointNet is that it fails to consider local structures, yet, our self-attention layer bypass this issue through the use of a positional encoding trick. In the original Attention paper[4] they added positional encodings to the input embeddings of the same dimension as the embeddings, so that the two can be summed and proposed sine and cosine functions of different frequencies to easily learn to attend by relative positions. They actually used the sinusoidal version because it may allow the model to extrapolate to sequence lengths longer than the ones encountered during training.

In 3D point cloud processing, the 3D point coordinates themselves are a natural candidate for position encoding. In our case, we implemented the trainable, parameterized position encoding introduced by Hengshuang Zhao et al.[6].

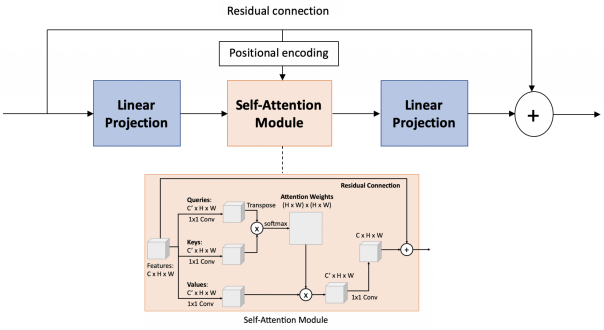


Fig. 4. The Transformer Block

This position encoding function is an MLP with two linear layers and one ReLU non-linearity. The position encoding is important for both the attention generation branch and the feature transformation branch. Thus in our vector self-attention layer the trainable position encoding are added in both the MLP that takes as input the keys and query, and also in aggregated values.

C. Transformer Block

After trying to stack self-attention layers to linear layers we realized that we obtained vanishing gradients and thus formed our Transformer block through residual layers inspired by the point transformer block[6]. The transformer block uses the self-attention layer as its core, as well as linear projections, normalization layers and a residual connection. The input is a set of feature vectors with $32*2^i$ channels (with $i = 1, 2, 3, 4$) with associated 3D coordinates. The structure of our Transformer Block is described in Fig 4The Transformer Block facilitates information exchange between these localized feature vectors, producing new feature vectors for all data points as its output considering both local and global features.

D. Auto-Encoder Architecture

The architecture of our point cloud auto-encoder is essentially constructed based on the Transformer Block. The encoder reduces the input dimensions downsampling blocks each separated by a Transformer block that aggregates operator the features at different scales throughout the network. In a similar way the decoder, upsamples the latent spaces features through interpolations and each upsampling block is separated by a Transformer block. The whole network architectures are visualized in Fig 5.

a) The Encoder: The feature encoder has five stages that operate on progressively downsampled point sets. The downsampling rates for the stages are $1, 4, 4, 4, 4$, thus the cardinality of the point set produced by each stage is N, N/4, N/16, N/64, N/256, where N is the number of input points, which is 1024 in our case. The number of stages and the down-sampling rates can be varied depending on the size of the dataset input and the type of applications. If there is a specific need to analyze fine-grained structures the number of stages is usually higher with smaller down-sampling rates but for real-time applications light-weight encoders allow faster processing. The input is primarily processed by linear projection layers connected by ReLU non-linearities to extract primary features. Then the consecutive stages of Transformer Blocks are connected by downsampling blocks that encodes the consecutive features. To reduce the cardinality of the point set we perform farthest point sampling and then pool the feature vectors with a kNN graph (k = 16) these choices results from a fine-tuning parameter study previously made in [1]. Hence, in each downsampling blocks the input feature goes through a linear transformation, followed by batch normalization and ReLU, followed by max pooling from its k original neighbors.

b) The Decoder: The decoder has a fairly similar structure where the downsampling blocks are replaced by interpolation layers to decode the features from the latent space. Indeed, they map the consecutive downsampled features onto a higher-resolution space. Hence, in each interpolation block the input point features are processed by a linear layer, followed by batch normalization and ReLU, and then the features are mapped onto the higher-resolution point set via trilinear interpolation. Moreover, we use a residual connection between the encoder and the decoder for each stage of the same feature dimensions. At each stage of the decoder, the interpolated features are summarized with the features from the corresponding encoder stage through a residual connection. Finally, at the last decoder stage we apply an MLP to map the feature vectors corresponding to each point in the input point set to the final logits.

III. Experiments and Results

A. Procedure

To assess the performance of our auto-encoder, we have cut the work in two parts. First we have tested the accuracy of our encoder separately to the rest of the network on a 3D shape classification task using the common dataset ModelNet40[5]. Then, to evaluate the effectiveness of the whole structure we have evaluated our auto-encoder on a 3D part-segmentation task using another very common dataset called ShapeNetPart[5]. Another experiment that could be done in a future work is to evaluate the ability of the auto-encoder to reconstruct the data into its initial shape. This should not be a daunting task using as a foundation the neural network structure used to perform part segmentation since it reconstruct the data back into its initial shape while adding a label to each point, but instead of using a Cross-Entropy loss as we did for both tasks, using a Chamfer loss should be more suited for a reconstruction task.

B. Implementation details

Our Point-Cloud Auto-Encoder is implemented in PyTorch. We use the SGD optimizer with momentum set to 0.9 and learning rate set to 0.05. For both 3D shape classification on ModelNet40 and 3D object part segmentation on ShapeNet-Part, we only trained the network for 50 epochs for some

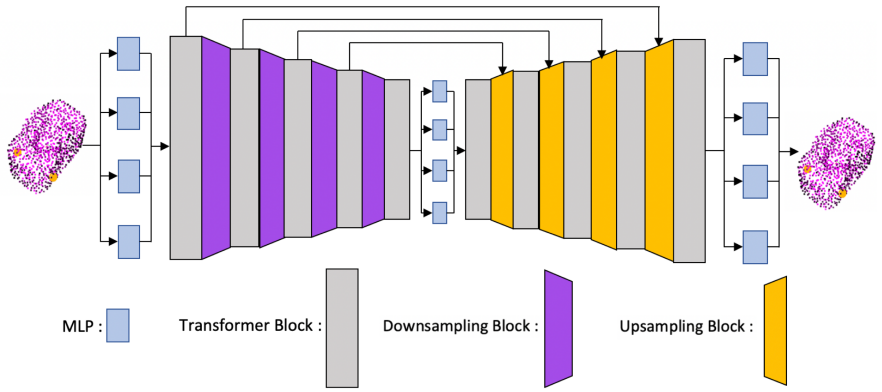


Fig. 5. The Neural Network architecture of our Auto-Encoder

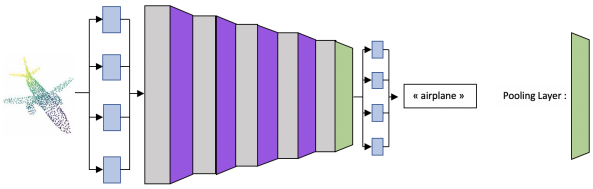


Fig. 6. The modified architecture of our encoder to test it on 3D Shape Classification

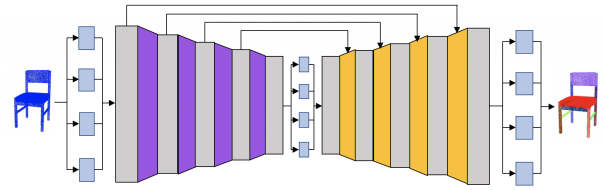


Fig. 7. The architecture of our auto-encoder to test it on 3D Part Segmentation

capacity and GPU issues. Also, for 3D object part segmentation, we only kept a handful of objects (Airplane, Bag, Cap and Car) instead of training on the whole dataset.

C. Testing the Encoder on 3D Shape Classification task

To train our encoder on 3D Shape Classification task we had to make some small modifications on the network architecture in order to interpret and reduce the latent space into a set of vectors representing the object category labels. To this extent we added a Global Average Pooling Layer before the final projection layer to which we also adapted the dimensions in order to get a vector of the size of the number of categories. Fig 6 represents the structure used for 3D Shape Classification.

Then, we evaluated it on ModelNet40, a dataset containing 12,311 CAD models with 40 object categories. They are split into 9,843 models for training and 2,468 for testing. As in PointNet++ [3] we preprocessed the data through uniform sampling of each CAD model together with the normal vectors from the object meshes. For evaluation metrics, we use the mean accuracy within each category and the overall accuracy over all classes.

The overall accuracy of our Auto-encoder on ModelNet40 is 81.5% and the mean class accuracy is 73.3% which is lower than the state of the art which is respectively 93.7 and 89.7% but we consider that for such a reduced number of training epochs it is quite satisfying.

D. Testing the Auto-Encoder on 3D Part-Segmentation task

To train our whole network on 3D Shape Classification task we also had to make some small modifications on the architecture. Indeed, an auto-encoder is only expected to output a reconstruction of the input in its initial structure with a little degradation due to the loss of information in the data compression for the latent space. However, to perform part-segmentation we had to assign each point to a part label.

Then, we evaluated it on the ShapeNetPart dataset which is annotated for 3D object part segmentation. It consists of 16,880 models from 16 shape categories, with 14,006 3D models for training and 2,874 for testing. The number of parts for each category is between 2 and 6, with 50 different parts in total. In our case we only kept 4 parts: the airplane which as 4 parts, the bag which as 2 parts, the cap which as 2 parts and the car which as 4 parts. Again we preprocessed the data using PointNet++[3] sampling technique. For evaluation metrics, we report category mIoU and instance mIoU.

The Point-Cloud Auto-Encoder obtains a category mIoU of 78.9% and instance mIoU of 79.3% which is not very far from the state of the art of respectively 83.7% and 86.6%. In more details, the mIoU of the Airplane is 81.0%, Bag: 75.8%, Cap: 83.3% and the Car: 75.7%. Our best class average mIOU is 79.3% and best instance average mIOU is 79.5%. For only 50 epochs we consider those results pretty much satisfying.

E. Application for Robot Grasping

Our point cloud auto-encoder using transformer blocks is intended to be used for a robot grasping application. A Contact

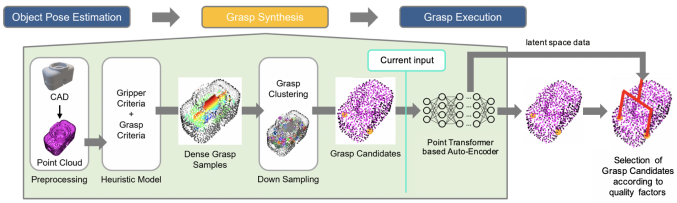


Fig. 8. Application of our Point Transformer based Auto-Encoder in a Robot Grasping Application

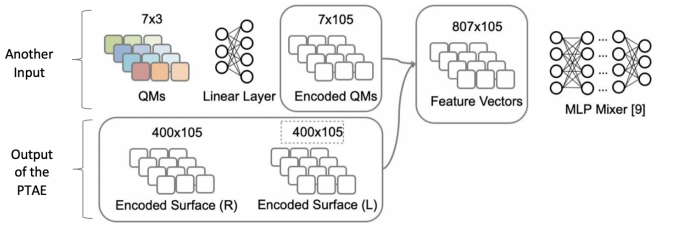


Fig. 9. Use of the encoded surfaces in the PTAE latent space for Robot Grasping

Point Pair Estimator (CPPE) for 6DOF grasp detection takes as input a CAD file of a target object to grasp, performs Object Pose Estimation to obtain its position and orientation, infers a point cloud of the represented object and then, through down sampling and clustering performs a grasp synthesis to reduce the number of grasp candidates. Those grasp candidates which are potential surfaces for the gripper to grasp the object is the input of our algorithm as we can see on Fig 8. The goal of our auto-encoder is to encode the points in a smaller but higher dimensional feature-space that will allow to select and only keep a handful of anchor points that the gripper will use to perform robot grasping. The main idea is that for each CAD file we will consider some physical parameters such as friction coefficients and gripper force that will be gathered to form Quality Matrices (QMs on Fig 9). Then, a Linear Layer will encode those QMs on a latent-space that has as many features as the PTAE latent-space to aggregate those QMs to each encoded surfaces and then apply a classifier (MLP Mixer). The classifier will assign to each encoded surface a quality grasping rate to elect only a handful of anchor points that will be targets for the robot to perform the grasping.

References

[1] Meng-Hao Guo et al. “PCT: Point Cloud Transformer”. In: *Computational Visual Media* 7 (2021), pp. 187–199.

[2] Charles R Qi et al. “PointNet: Deep learning on point sets for 3d classification and segmentation”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 652–660.

[3] Charles Ruizhongtai Qi et al. “PointNet++: Deep hierarchical feature learning on point sets in a metric space”. In: *Advances in neural information processing systems* 30 (2017).

[4] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[5] Zhirong Wu et al. “3D ShapeNets: A deep representation for volumetric shapes”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1912–1920.

[6] Hengshuang Zhao et al. “Point Transformer”. In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pp. 16259–16268.