Supplementary

In the following sections we provide additional details and analysis regarding our results. In Section A we provide experimental details and additional results for our method. Section B displays visual examples of retrieved shapes and progressive sampling. Section C offers a new application for the progressive concept: A progressive autoencoder.

A. Additional results

In this section we extend the results section of the paper to include: details of our experimental parameters, analysis of the different matching methods, evaluation of regularization parameters, elaboration of the space and time considerations, analysis of critical set sampling, and comparison of the class accuracy.

A.1. Experimental details

Classification Architecture The architecture of S-NET for the classification experiment is inspired by the vanilla version of PointNet by Qi *et al.* [31]. We use per point 1×1 convolution layers with output sizes of $[64, 64, 64, 128, 128]$. Then, a feature-wise max-pooling layer is used to obtain a global feature vector. This feature vector is passed through four fully connected layers of size $[256, 256, 256, k \times 3]$, where k is the sample size.. All convolution and fully connected layers are followed by ReLU non-linearity [28] and batch-normalization layer [15], except for the output layer. ProgressiveNet takes the architecture of S-NET with $k = 1024$.

Classification Optimization We used Adam optimizer [18] with a learning rate of 0.01, decay rate of 0.7 every 60000 steps and batch size of 32 point clouds. The regularization weights (for equations 5 and 6) were set to $\alpha = 30$, $\beta = 1$, $\gamma = 1$, $\delta = 0$ for S-NET and $\alpha = 30$, $\beta = 1$, $\gamma = 0.5$, $\delta = 1/30$ for ProgressiveNet. We trained the networks for 500 epochs with a GTX 1080 Ti GPU. When using PointNet as the task network, S-NET training takes between 2 to 7 hours, depending on the sample size. ProgressiveNet training takes 6 hours, when using PointNet vanilla as the task network.

Classification Augmentation We employed the augmentation proposed by Qi *et al.* [31]: random rotation of the point cloud along the up-axis, and jittering the position of each point by a Gaussian noise with zero mean and 0.02 standard deviation.

Reconstruction Architecture The architecture of S-NET for the reconstruction experiment follows the same basic structure as in the classification case, with 1×1 of sizes $[64, 128, 128, 256, 128]$ and fully connected layers of seizes $[256, 256, k \times 3]$. ProgressiveNet takes the architecture of S-NET with $k = 2048$.

Sampling ratio	FPS	ProgressiveNet $(1+\epsilon)$ matching	ProgressiveNet NN matching
1	87.3	87.3	87.3
2	85.6	**85.7**	85.4
4	81.2	**82.4**	82.3
8	68.1	76.4	**78.2**
16	49.4	68.3	**74.4**
32	29.7	50.7	**61.0**
64	16.3	27.8	**40.0**

Table 3. **Matching methods comparison.** We compare the classification accuracy of PointNet vanilla on the sampled points when using different matching methods: $(1+\epsilon)$ and Nearest Neighbour (NN). FPS is shown for reference. NN matching is better for all sampling ratios larger than 4.

Reconstruction Optimization To train both S-NET and ProgressiveNet, we used Adam optimizer with a learning rate of 0.0005, momentum 0.9 and mini-batches of 50 shapes. The regularization weights were set to $\alpha = 0.01$, $\beta = 1$, $\gamma = 0$, $\delta = 1/64$. For ProgressiveNet, the total loss is divided by the number of sample sizes $|C_s|$. We trained the networks for 500 epochs with a Titan X Pascal GPU. S-NET training takes between 4 to 8 hours, depending on the sample size. ProgressiveNet training time is about 12 hours. No augmentation was used for reconstruction.

A.2. Matching methods

We examined two matching methods. The first one is based on nearest neighbour (NN) matching, as detailed in Section 3.2. In the second one, we adapted the implementation of the $(1+\epsilon)$ approximation scheme for EMD [4], provided by Fan *et al.* [8]. This implementation gives a matching score for each point in one point cloud to each point in the other one. We feed the algorithm with G and P and take the points from P corresponding to the highest matching score for each point in G. In Table 3 we compare these matching methods. The accuracy is better with NN matching for large sampling ratios, and is almost the same for small ones.

NN matches every point in G with its closest point in P. This ensures that all generated points that S-NET learned are reflected in the matched set. This is not the case when using $(1+\epsilon)$, where a generated point might be matched with a far input point, if it serves the $(1+\epsilon)$ approximation scheme. In addition, NN matching is independent for each point, thus it guarantees to keep the order of the points produced by ProgressiveNet.

A.3. Regularization

In this section we explore the influence of the sampling regularization loss and it's components. Please see Section 3.1 for the relevant background and equations.

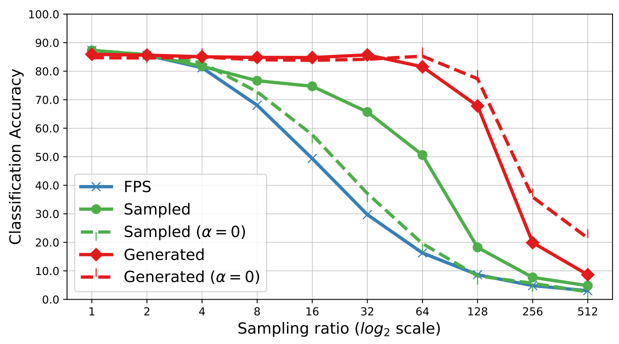

Figure 11. **Turning off the sampling regularization loss ($\alpha = 0$).** PointNet vanilla was trained on complete point clouds (1024 points) and evaluated on either S-NET's generated or sampled points, either with or without the sampling regularization loss. Without the regularization, the accuracy using the generated points is slightly increased. However, the accuracy when using the sampled points decreases substantially.

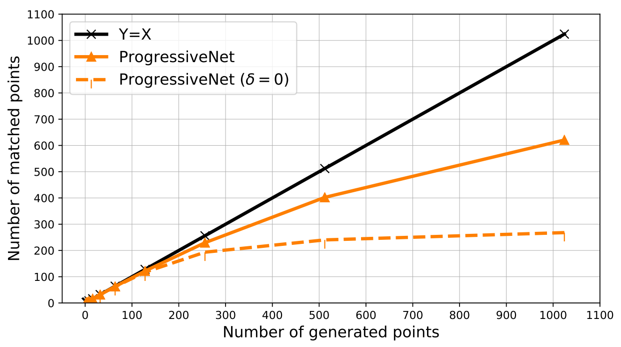

Figure 12. **Turning off the linear factor for L_b component ($\delta = 0$).** Average number of unique points in the initial sampled set (after NN matching, before FPS completion) is shown as a function of the number of generated points. Ideally, we would like to get the line $Y = X$, which means that every generated point has a unique matched input point. Setting $\delta = 0$ greatly reduce the number of unique matches.

Turning off the regularization ($\alpha = 0$) Figure 11 shows the accuracy of PointNet vanilla using either S-NET's generated or sampled points, with and without regularization. Without the regularization, the accuracy using the generated points is slightly increased, which is expected, since the regularization sacrifices some of the classification loss to minimize the sampling loss. However, after the matching process the classification accuracy on the unregularized sampled points is much lower than when using the regularization. Without the sampling regularization, the generated points are not close to the original points and are not spread out over the whole shape. As a result, the matching process causes the sampled points to be very different from the generated points. Therefore, the sampled points do not preserve the contribution of the generated points to the task.

Turning off the linear factor for L_b component ($\delta = 0$) In Figure 12 we see the average number of unique points in the initial sampled set (after NN matching, before FPS completion) as a function of the number of generated points. For low values, the numbers are almost the same, but they diverge for higher values. Ideally, we would like to get the identity line $Y = X$, meaning that each generated point is uniquely matched with an input point without any collisions. Using a linear factor for the L_b regularization term, such that the weight is larger for large sampling sizes, improves the spread of the generated points over the shape and reduces collisions.

Turning off the L_b component ($\gamma, \delta = 0$) In Figure 13 we see the accuracy of PointNet vanilla using ProgressiveNet sampled points, either with or without the L_b regularization term. Without this term, the generated points tend to concentrate on small portions of the input point cloud, resulting in many collisions in the matching process. This

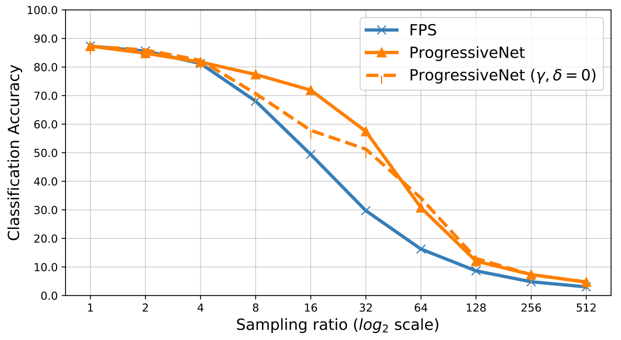

Figure 13. **Turning off the L_b component ($\gamma, \delta = 0$).** PointNet vanilla was trained on complete point clouds (1024 points) and evaluated on ProgressiveNet's sampled points, either with or without the L_b regularization term. The results are evidently better with the L_b term.

makes the FPS completion more dominant.

For low sampling ratios this is not a concern, since FPS gives decent results. For very high sampling ratios this is also not a concern, since the task loss forces the small number of generated points to spread over the input shape even without the regularization. However, for the mid-range of sampling ratios (larger than 4 and smaller than 64) the L_b regularization term is crucial. It spreads the generated points over the input point cloud, allowing for better matching and therefore better accuracy when using the sampled points.

Turning off the L_m component ($\beta = 0$) Removing the L_m term results in 3.5-fold increase in the maximal distance between the generated points and their matched points, averaged over the test set. This leads to less tight matches, resulting in up to 7% (obtained for $k = 16$) decrease in classification accuracy when using the sampled points.

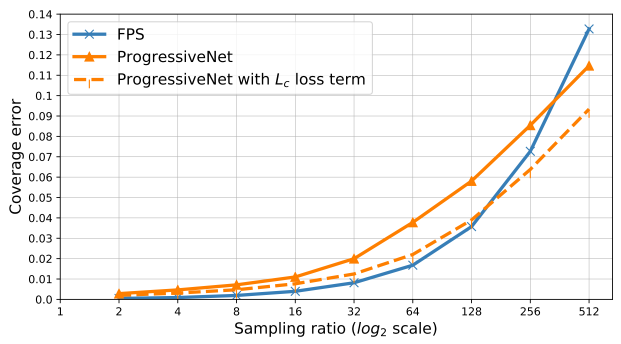

Figure 14. **Adding L_c component to the loss.** Coverage error is defined as the distance of the next FPS point. We observe that adding the L_c component closes most of the gap between FPS and ProgressiveNet.

Adding an extra regularization term FPS minimizes a geometric measure, i.e, given k points it minimizes the distance to the $k+1$ point. We regard this measure as the coverage error. Our work found it to be an inferior proxy to minimizing the task error. Nonetheless, if in some settings the coverage error is important, we can incorporate it by adding the following loss term to Equation 5:

$$L_c(G, P) = \max_{y \in P} \min_{x \in G} ||y - x||_2^2. \tag{9}$$

Figure 14 shows the coverage error as a function of the sampling ratio, for FPS and ProgressiveNet, as well as for ProgressiveNet when trained with the extra L_c loss term. We observe that adding the extra term removes most of the coverage error difference between FPS and ProgressiveNet.

PointNet's accuracy when using the sampled points did not change substantially when adding the extra term, thus we conclude that it is unnecessary to minimize the coverage error in order to get a sample that is optimized for the task. However, it is not harmful to minimize this error as well, if needed.

A.4. Time and space considerations

In Section 4.1 we mentioned the influence of S-NET on inference time and memory consumption. This section elaborates on the subject. Table 4 demonstrates how we save most of the inference time by using S-NET for a small increase in memory. When fed with a complete point cloud (with 1024 points), PointNet performs 440M floating point multiplications (FLOPs). When feeding only 16 points, this number drops to 7M. S-NET that samples from 1024 to 16 points requires 35M FLOPs. Cascading them together amounts to 42M FLOPs, which is 90% reduction in inference time, compared to running PointNet on the complete points clouds. S-NET that samples 16 points requires 0.18M parameters, which is only 5% of PointNet's memory requirement.

Network	#Parameters	FLOPs/ point cloud
PointNet-1024	3.5M	440M
PointNet-16	3.5M	7M
S-NET-16	0.18M	35M
S-NET-16 + PointNet-16	3.68M	42M

Table 4. **Time and space complexity of S-NET and PointNet**. "M" stands for million. FLOPs stands for number of floating point multiplications. S-NET-16 stands for S-NET with output size of 16 points. S-NET-16 + PointNet-16 stands for sampling 16 points with S-NET and then classifying them with PointNet. Sampling makes for a great reduction in time complexity for a modest price in terms of space complexity.

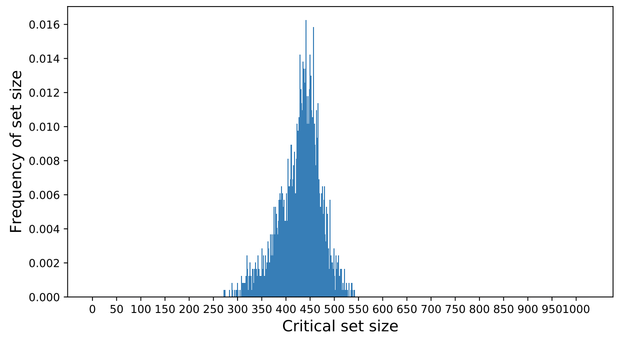

Figure 15. **Distribution of critical set sizes for PointNet vanilla over ModelNet40 test set.** The critical set size is concentrated around 429 points.

A.5. Critical set sampling

The critical set, as defined by Qi *et al.* [31], is the set of points that contributes to the max pooled feature (MPF). A proposed alternative to S-NET might be to extract the critical set and use it as the sampled point cloud. This method has three main disadvantages: first, it does not allow to control the sample size, but only sample to the size of the critical set; Second, each point cloud will be sampled to a different size, since the critical set size is not the same across the dataset, which does not allow efficient processing and storing; Third, the average critical set size of PointNet vanilla on ModelNet40 test set is 429 out of 1024 points (see Figure 15), i.e., approximately 42% of the points, which is equivalent to sampling ratio of about 2.5. On the contrary, S-NET enables control of the sample size and performs well for much larger sampling ratios. This makes the proposed alternative to S-NET not viable.

A more plausible alternative might be to sample a subset of the critical set that controls most of the features. To do so, we count the number of features that each point contributes to the max-pooled features and select the k points with most contribution. We denote this process as critical set sampling.

Sampling ratio	FPS	ProgressiveNet	Critical set sampling
1	87.3	87.3	87.3
2	85.6	85.4	**87.3**
4	81.2	82.3	**86.3**
8	68.1	**78.2**	76.0
16	49.4	**74.4**	45.6
32	29.7	**61.0**	22.5
64	16.3	**40.0**	12.5

Table 5. **Critical set sampling.** We compare the classification accuracy of PointNet vanilla for different sampling methods and sampling ratios. Critical set sampling samples the critical points that contribute to the most features to the MPF. This method is competitive for very small sampling ratios, but is not viable for larger sampling ratios.

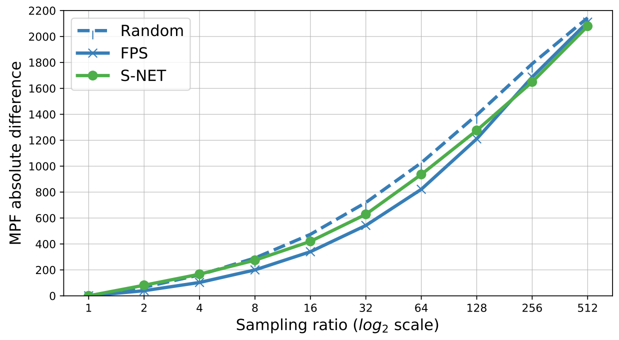

Figure 16. **Absolute difference of the max pooled feature (MPF).** Three sampling methods are compared: random, FPS and S-NET . The difference is averaged over the test set of ModelNet40. The MPF resulting from S-NET's points differs from the original MPF like random and FPS.

In Table 5 we compare this sampling method with ProgressiveNet and FPS. We see that critical set sampling performs well for very small sampling ratios, but performs poorly for larger ratios.

Relation between our sampling and the critical set In this experiment we measured the percentage of critical set points covered by S-NET's sampled points. We found that S-NET did not cover the critical set more than a random sample. To further investigate this issue, we computed the absolute difference between the MPF when feeding PointNet with sampled points and with the complete point cloud of 1024 points. The results are shown in Figure 16.

S-NET did not learn to sample the critical set. It also did not learn to reproduce the MPF. Instead, it learned to sample points that give better classification results, independently of the critical set. To explain this, we recall that the fully connected layers of PointNet, which process the MPF to infer the classification results, form a non-linear function. In addition, the MPF represents the shape class with redundancy (1024 floating point numbers to represent a class out of 40 classes). Thus, it is possible to find several different MPFs that results in the same classification. Therefore, there are multiple sets of points that gives similar classification results.

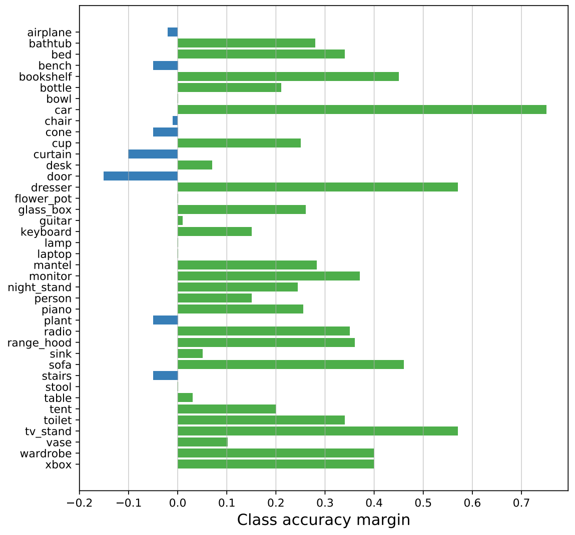

Figure 17. **Class accuracy margin.** PointNet was trained on the complete point clouds (1024 points) and evaluated on sampled point clouds of $k = 64$ points, using either FPS or S-NET sampling. S-NET achieves better results on 27 classes, with an average margin of 26%, while FPS achieves better results on 8 classes with an average margin of just 6%.

To further stress this point, we evaluated PointNet accuracy on the complementary set of the critical set for each shape, and the accuracy only drops by 1.5%, from 89.2% to 87.7%. We conclude that the critical set is not that critical.

A.6. Class accuracy

Up to this point we reported instance classification accuracy (also regarded as overall accuracy). Now we analyze the per class accuracy behaviour. Figure 17 shows the per-class difference in accuracy between using 64 S-NET and 64 FPS points. S-NET achieves superior results in 27 out of 40 classes (67.5%) while FPS is better in only 8 classes (20%). The results are equal for the remaining 5 classes. The average margin on the classes with superior S-NET results is 26%, compared to just 6% average margin for the classes with better FPS results. Notably, FPS achieves higher accuracy on the door class with a margin of 15% (80% vs 65%), while S-NET achieves higher accuracy on the car class with a margin of 75% (91% vs 16%).

B. Visual examples

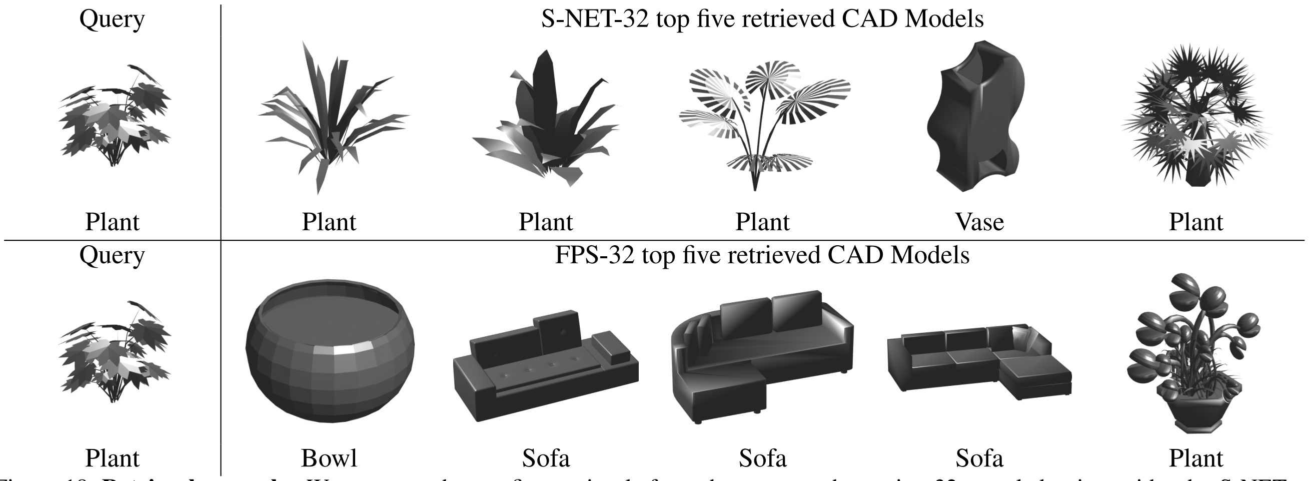
Query
S-NET-32 top five retrieved CAD Models
Plant
Plant
Plant
Plant
Vase
Plant
Query
FPS-32 top five retrieved CAD Models
Plant
Bowl
Sofa
Sofa
Sofa
Plant

Figure 18. **Retrieval example.** We compare the top five retrievals from the test set when using 32 sampled points, either by S-NET or FPS. The sampled points were processes by PointNet (that was trained on complete point clouds of 1024 points) and its penultimate layer was used as a shape descriptor. Retrieval was done based on L_2 distance on this shape descriptor. S-NET was trained with PointNet for classification, no additional training was done for retrieval. When using S-NET, four out of five retrieved shapes are correct (plant). For FPS, only the fifth retrieved shape is correct.

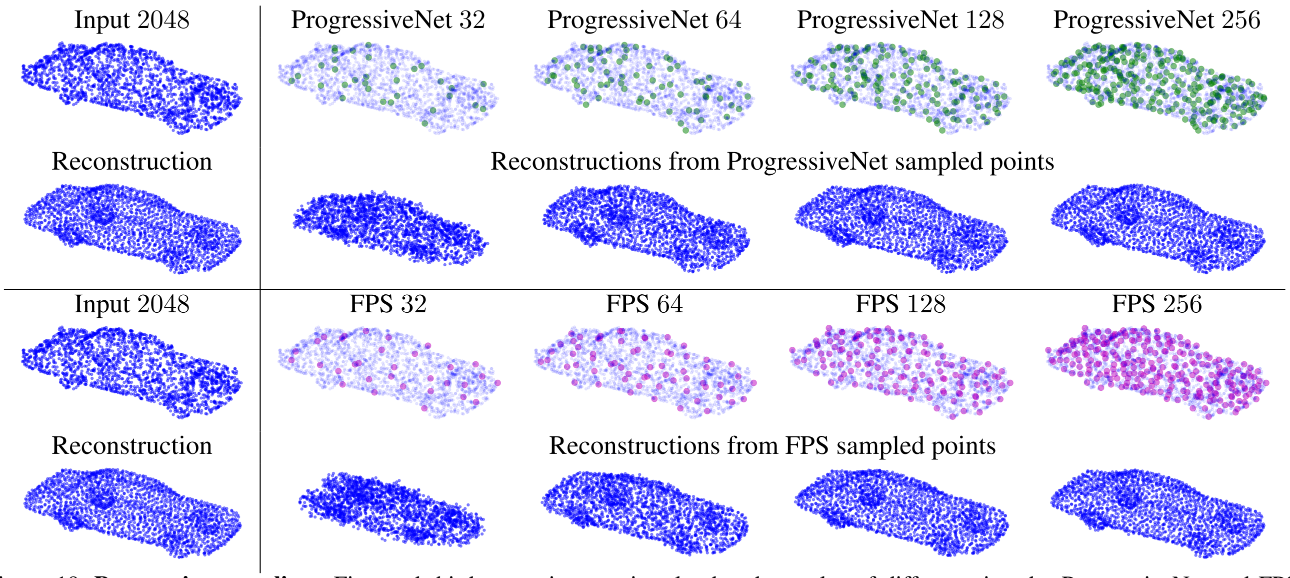

Figure 19. **Progressive sampling.** First and third rows: input point cloud and samples of different sizes by ProgressiveNet and FPS, respectively. Second and fourth rows: reconstruction from the input point cloud and from the corresponding samples. The sampled points are enlarged for visualization purpose. Even at a sample size as low as 64 points, the reconstruction from ProgressiveNet's points is visually similar to the reconstruction from the complete input point cloud. On the contrary, it takes four times more FPS points to achieve this level of similarity.

C. Extension - Progressive Autoencoder

We extended the training concept of ProgressiveNet for training a progressive autoencoder, named ProgressiveAE. That is an autoencoder whose output points are ordered according to their contribution to the reconstruction of the input point cloud.

Motivation The scheme of ProgressiveAE fits naturally to a Client-Server scenario and its benefits are three-fold: lower communication load, lower memory footprint and level-of-detail control.

Suppose that the encoder is located at the server. Instead of sending the whole point cloud (2048 3D points, which are 6144 floats) to the client, the server sends only the latent vector of 128 floats (98% reduction in communication load). The client possesses only the decoder part of ProgressiveAE, which is a progressive decoder.

According to the available memory resources and the required level-of-detail, the client may hold only the parameters corresponding to the first c output points of the decoder and reconstruct c points instead of n. For example, if the client capacity allows only reconstruction of up to 256 points, it will hold the parameters needed to calculate the first 256 output points. This amounts to more than 80% reduction in memory consumption (number of parameters) on the client side, compared to holding the complete 2048-points decoder.

Furthermore, the client can choose in real time to reconstruct an even smaller point cloud to reduce computation load. For example, calculating the first 128 output points results in almost 90% reduction in the number of floating point operations (FLOPs), compared to reconstructing the complete point cloud.

Figure 20 summarizes the time and space requirements for reconstructing point clouds of different sizes. It is much more efficient to reconstruct a point cloud to the required size than to reconstruct the complete point cloud and to then sample it to the required size.

Implementation The architecture of ProgressiveAE is the same as that of the autoencoder proposed by Achlioptas *et al.* [1], referred to as BaselineAE. The input and output of both autoencoders consist of $n = 2048$ points.

Similar to the training of ProgressiveNet, we trained ProgressiveAE with several loss terms. Each term computes the Chamfer distance between the first c point of output and the n points of the input. The overall loss is the sum of all loss terms. For this experiment we used loss terms for $C_s = \{16, 32, \ldots, 2048\}$. The loss for training the BaselineAE was Chamfer distance between the input and the output point clouds of n points. All other training conditions are the same as those detailed in Section A.1.

Results To measure the reconstruction performance, we took the first c points from ProgressiveAE and computed

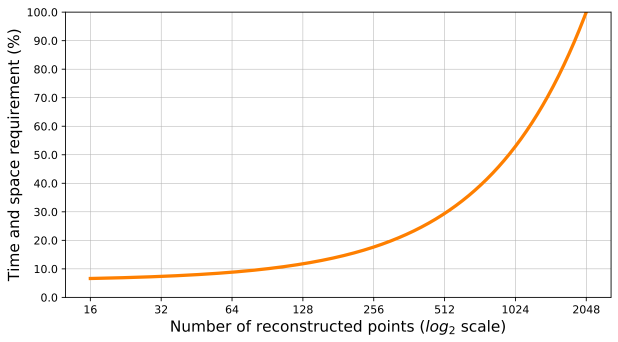

Figure 20. **Progressive decoder time and space requirements.** While a traditional autoencoder reconstructs a point cloud of a fixed size, ProgressiveAE enables real time level-of-detail management, allowing for a great reduction in inference time. In addition, the progressive decoder may hold only the parameters needed to reconstruct a point cloud smaller then the original, allowing for a reduction in memory as well.

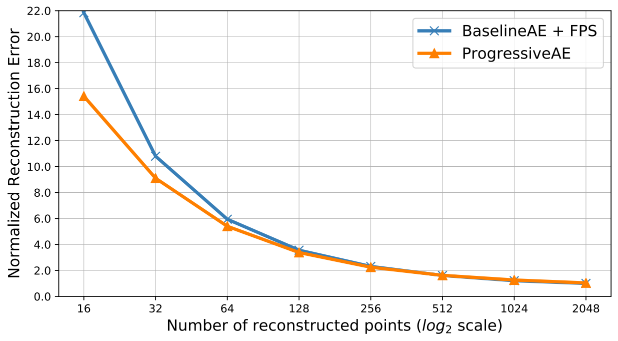

Figure 21. **Progressive autoencoder.** The Normalized reconstruction error (NRE) was computed on the test split of the data, the normalization factor is the reconstruction error when using the BaselineAE to reconstruct from complete point cloud (2048 points). ProgressiveAE has equal or better reconstruction error.

the reconstruction error as Chamfer distance from the input point cloud. As an alternative reconstruction approach with c points, we sampled c points with FPS from the output of BaselineAE. We find that ProgressiveAE has equal or better reconstruction error for for any reconstruction size (see Figure 21).

In an additional experiment, we took the encoder parameters, learned by BaselineAE, and trained only the decoder of ProgressiveAE (the variables of the layers after the max-pool operation). Interestingly, the reconstruction error for ProgressiveAE in this experiment was almost the same as in the end-to-end training of ProgressiveAE. This means that a progressive decoder can be trained to work with the encoder of an existing autoencoder, without paying any cost in terms of reconstruction quality.