Supplementary Material - Going Deeper with Lean Point Networks

Eric-Tuan Le[1] Iasonas Kokkinos[1,2] Niloy J. Mitra[1,3]

[1]University College London [2]Ariel AI [3]Adobe Research

1. Details on evaluation results

1.1. Datasets

We evaluate our networks on the point cloud segmentation task on three different datasets, ordered by increasing complexity:

- ShapeNet-Part [1]: CAD models of 16 different object categories composed of 50 labeled parts. The dataset provides $13,998$ samples for training and $2,874$ samples for evaluation. Point segmentation performance is assessed using the mean point Intersection over Union (mIoU).

- ScanNet [2]: Scans of real 3D scenes (scanned and reconstructed indoor scenes) composed of 21 semantic parts. The dataset provides $1,201$ samples for training and 312 samples for evaluation. We follow the same protocol as in [5] and report both the voxel accuracy and the part Intersection over Union (pIoU).

- PartNet [4]: Large collection of CAD models of 17 object categories composed of 251 labeled parts. The dataset provides $17,119$ samples for training, $2,492$ for validation and $4,895$ for evaluation. The dataset provides a benchmark for three different tasks: fine-grained semantic segmentation, hierarchical semantic segmentation and instance segmentation. We report on the first task to evaluate our networks on a more challenging segmentation task using the same part Intersection over Union (pIoU) as in ScanNet.

1.2. Evaluation metrics

To report our segmentation results, we use two versions of the Intersection over Union metric:

- mIoU: To get the per sample mean-IoU, the IoU is first computed for each part belonging to the given object category, whether or not the part is in the sample. Then, those values are averaged across the parts. If a part is neither predicted nor in the ground truth, the IoU of the part is set to 1 to avoid this indefinite form. The mIoU obtained for each sample is then averaged to get the final score as,

$$\text{mIoU} = \frac{1}{n_{\text{samples}}} \sum_{s \in \text{samples}} \frac{1}{n_{\text{parts}}^{\text{cat(s)}}} \sum_{p^i \in \mathcal{P}_{\text{cat(s)}}} \text{IoU}_s(p^i)$$

  with n_{samples} the number of samples in the dataset, cat(s), $n_{\text{parts}}^{\text{cat(s)}}$ and $\mathcal{P}_{\text{cat(s)}}$ the object category where s belongs, the number of parts in this category and the sets of its parts respectively. $\text{IoU}_s(p^i)$ is the IoU of part p^i in sample s.

- pIoU: The part-IoU is computed differently. The IoU per part is first computed over the whole dataset and then, the values obtained are averaged across the parts as,

$$\text{pIoU} = \frac{1}{n_{\text{parts}}} \sum_{p \in \text{parts}} \frac{\sum_{s \in \text{samples}} \text{I}_s(p^i)}{\sum_{s \in \text{samples}} \text{U}_s(p^i)}$$

  with n_{parts} the number of parts in the dataset, $\text{I}_s(p^i)$ and $\text{U}_s(p^i)$ the intersection and union for samples s on part p^i respectively.

To take into account the randomness of point cloud sampling when performing coarsening, we use the average of 'N' forward passes to decide on the final segmentation during evaluation when relevant.

Table 1. Summary of the impact of our module implants in five different networks on ShapeNet-Part. The impact is measured by four metrics: (i) memory footprint, (ii) IoU, (iii) inference time and (iv) backward time. With all tested architectures, our lean modules decrease the memory footprint while allowing small improvements in terms of IoU. The impact on inference time depends on the choice of the network but can range from positive impact to a small slowdown.

	Memory	IoU	Inference	Backward
PN++	**-76%**	**+1.2%**	**-46%**	**-83%**
DGCNN	**-69%**	**+0.9%**	**-22%**	**-49%**
SCNN	**-28%**	**+2.2%**	+27%	+193%
PointCNN	**-56%**	**+1.0%**	+35%	+71%

Table 2. Per-category performance mIoU on ShapeNet-Part (Top) and pIoU on PartNet (Bottom) based on a training on each whole dataset all at once. On ShapeNet-Part, all of our network architectures outperform PointNet++ baseline by at least +1.0%. Our deep architecture still improves the performance of its shallower counterpart by a small margin of +0.1%. On PartNet, the fine details of the segmentation and the high number of points to process make the training much more complex than previous datasets. PointNet++, here, fails to capture enough features to segment the objects properly. Our different architectures outperform PointNet++ with a spread of at least +2.0% (+5.7% increase). With this more complex dataset, deeper networks become significantly better: our Deep LPN network achieves to increase pIoU by +9.7% over PointNet++ baseline, outperforming its shallow counterpart by +2.1%.

	Tot./Av.	Aero	Bag	Cap	Car	Chair	Ear	Guitar	Knife	Lamp	Laptop	Motor	Mug	Pistol	Rocket	Skate	Table
No. Samples	13998	2349	62	44	740	3053	55	628	312	1261	367	151	146	234	54	121	4421
PN++	84.60	82.7	76.8	84.4	78.7	90.5	72.3	90.5	86.3	82.9	**96.0**	72.4	94.3	80.5	**62.8**	76.3	81.2
mRes	85.47	**83.7**	77.1	85.4	79.6	**91.2**	73.4	91.6	**88.1**	84.1	95.6	75.1	95.1	81.4	59.7	76.9	82.1
mResX	85.42	83.1	77.0	84.8	79.7	91.0	67.8	91.5	88.0	84.1	95.7	74.6	**95.4**	82.4	57.1	77.0	82.3
LPN	85.65	83.3	77.2	**87.8**	80.6	91.1	72.0	**91.8**	**88.1**	84.6	95.8	**75.8**	95.1	**83.6**	60.7	75.0	82.4
Deep LPN	**85.66**	82.8	**79.2**	82.7	**80.9**	91.1	**75.4**	91.6	**88.1**	**84.9**	95.3	73.1	95.1	83.3	61.6	**77.7**	**82.6**

	Tot./Av.	Bed	Bott	Chair	Clock	Dish	Disp	Door	Ear	Fauc	Knife	Lamp	Micro	Frid	Storage	Table	Trash	Vase
No. samples	17119	133	315	4489	406	111	633	149	147	435	221	1554	133	136	1588	5707	221	741
PN++	35.2	30.1	32.0	39.5	30.3	29.1	81.4	31.4	35.4	46.6	37.1	25.1	31.5	32.6	40.5	34.9	33.0	56.3
mRes	37.2	29.6	32.7	40.0	34.3	29.9	80.2	**35.0**	**50.0**	**56.5**	41.0	26.5	**33.9**	**35.1**	41.0	35.4	35.3	57.7
mResX	37.5	32.0	37.9	40.4	30.2	31.8	80.9	34.0	43.0	54.3	42.6	26.8	33.1	31.8	**41.2**	**36.5**	40.8	57.2
LPN	37.8	**33.2**	40.7	40.8	**35.8**	31.9	81.2	33.6	48.4	54.3	41.8	26.8	31.0	32.2	40.6	35.4	41.1	57.2
Deep LPN	**38.6**	29.5	**42.1**	**41.8**	34.7	**33.2**	**81.6**	34.8	49.6	53.0	**44.8**	**28.4**	33.5	32.3	41.1	36.3	**43.1**	**57.8**

1.3. Summary of the impact of our module

We experiment on four different networks that all exhibit diverse approach to point operation: (i) PointNet++ [6], (ii) Dynamic Graph CNN [7], (iii) SpiderCNN [8], (iv) PointCNN [3] . As detailed in Table 1, our lean blocks, being modular and generic, can not only increase the memory efficiency of that wide range of networks but can as well improve their accuracy. The effect of our blocks on inference time does vary with the type of network, from a positive impact to a small slowdown.

1.4. Detailed results from the paper

The following section provides more details on the evaluation experiments introduced in the paper. We present the per-class IoU on both ShapeNet-Part and PartNet datasets in Table 2 for each of the PointNet++ based architecture. Due to the high number of points per sample and the level of details of the segmentation, PartNet can be seen as much more complex than ShapeNet-Part.

On PartNet, the spread between an architecture with an improved information flow and a vanilla one becomes significant. Our PointNet++ based networks perform consistently better than the original architecture on each of the PartNet classes. Increasing the depth of the network allows to achieve a higher accuracy on the most complex classes such as Chairs or Lamps composed of 38 and 40 different part categories respectively. Our deep architecture is also able to better capture the boundaries between parts and thus to predict the right labels very close from part edges. When a sample is itself composed of many parts, having a deep architecture is a significant advantage.

Table 3. Per-class IoU on PartNet when training a separate network for each category, evaluated for three different architectures for *Chairs* and *Tables* (60% of the whole dataset). Our lean networks achieve here similar performance as their vanilla counterpart while delivering significant savings in memory.

		Chair	Table
DGCNN	Vanilla	**29.2 (+0.0%)**	22.5 (+0.0%)
	Lean	24.2 (-17.1%)	**28.9 (+28.4%)**
SCNN	Vanilla	30.8 (+0.0%)	**21.3 (+0.0%)**
	Lean	**31.1 (+1.0%)**	21.2 (-0.5%)
PointCNN	Vanilla	40.4 (+0.0%)	32.1 (+0.0%)
	Lean	**41.4 (+2.5%)**	**33.1 (+3.1%)**

Table 4. Memory and speed efficiency of our deep network Deep LPN with respect to two different implementations of DeepGCNs. Our network wins on all counts and successfully reduces the memory (- 75%) and increases the speed (- 48% and - 89% for the inference and the backward time respectively).

	Memory (Gb)	Inference Time (ms)	Backward Time (ms)
DeepGCN (Dense)	8.56	664	1088
DeepGCN (Sparse)	10.00	1520	445
Deep LPN	**2.18 (-75%)**	**345 (-48%)**	**67 (-85%)**

As additional reference, we provide on Table 3 the performance of our lean blocks applied to three architectures when training one network per-object category on PartNet, trained on *Chairs* and *Tables* as they represent 60% of the dataset.

For reference, we provide as well the absolute values for the efficiency of the previous networks measured by three different metrics on Table 4 and Table 5: (i) memory foot-

Table 5. Efficiency of our network architectures measured with a batch size of 8 samples or less on a Nvidia GTX 2080Ti GPU. All of our lean architectures allow to save a substantial amount of memory on GPU wrt. the PointNet++ baseline from 58% with mRes to a 67% decrease with LPN. This latter convolution-type architecture wins on all counts, decreasing both inference time (-41%) and the length of backward pass (-68%) by a large spread. Starting form this architecture, the marginal cost of going deep is extremely low: doubling the number of layers in the encoding part of the network increases inference time by 6.3% on average and the memory consumption by only 3.6% at most compared to LPN). When used in conjunction with other base architectures, similar memory savings are achieved by our blocks with low impact on inference time.

	Efficiency (%)			Memory Footprint (Gb)			Inference Time (ms)			Length Backward pass (ms)		
	ShapeNet-Part	ScanNet	PartNet	ShapeNet-Part	ScanNet	PartNet	ShapeNet-Part	ScanNet	PartNet	ShapeNet-Part	ScanNet	PartNet
PointNet++	84.60	80.5	35.2	6.80	6.73	7.69	344	238	666	173	26	185
mRes	85.47	79.4	37.2	2.09	2.93	4.03	395	379	537	54	**12**	68
mResX	85.42	79.5	37.5	2.38	3.15	4.13	441	383	583	122	26	138
LPN	85.65	**83.2**	37.8	1.65	**2.25**	**3.24**	**187**	**166**	**347**	**30**	15	**39**
Deep LPN	**85.66**	82.2	**38.6**	**1.42**	2.33	3.31	205	177	356	37	23	51

		Efficiency (%)			Memory Footprint (Gb)			Inference Time (ms)			Length Backward pass (ms)		
		ShapeNet-Part	ScanNet	PartNet	ShapeNet-Part	ScanNet	PartNet	ShapeNet-Part	ScanNet	PartNet	ShapeNet-Part	ScanNet	PartNet
DGCNN	Vanilla	82.59	74.5	20.5	2.62	7.03	9.50	41	194	216	41	82	104
	Lean	**83.32**	**75.0**	**21.9**	**0.81**	**3.99**	**5.77**	**32**	**158**	**168**	**21**	**45**	**57**
SCNN	Vanilla	79.86	72.9	17.9	1.09	4.33	5.21	**22**	**279**	**142**	**45**	**99**	**249**
	Lean	**81.61**	**73.2**	**18.4**	**0.79**	**3.25**	**3.33**	28	281	150	132	443	637
PointCNN	Vanilla	83.60	77.2	25.0	4.54	5.18	6.83	**189**	**229**	**228**	**109**	**71**	**77**
	Lean	**84.45**	**80.1**	**27.0**	**1.98**	**3.93**	**5.55**	256	278	263	186	225	208

print, (ii) inference time and (iii) length of backward pass. Our lean architectures consistently reduce the memory consumption of their vanilla couterparts while having a very low impact on inference time. When compared to DeepGCNs, our Deep LPN architecture wins on all counts by achieving the same performance while requiring less memory (-75%) and shorter inference (-48%) and backward (-89%) time.

2. Design of our architectures

In this section, we provide more details about how we design our lean architectures to ensure reproducible results for the following architectures, (i) PointNet++ [6], (ii) Dynamic Graph CNN [7], (iii) SpiderCNN [8], (iv) PointCNN [3] . We implement each networks in Pytorch following the original code in Tensorflow and we implant our blocks directly within those networks.

2.1. PointNet++ based architectures

To keep things simple and concise in this section, we adopt the following notations:

- S(n): Sampling layer of n points;
- rNN(r): query-ball of radius r;
- MaxP: Max Pooling along the neighborhood axis;
- $\bigoplus$: Multi-resolution combination;
- Lin(s): Linear unit of s neurons;
- Drop(p): Dropout layer with a probability p to zero a neuron.

Inside our architectures, every downsampling module is itself based on FPS to decrease the resolution of the input point cloud. To get back to the original resolution, upsampling layers proceed to linear interpolation (Interp) in the spatial space using the $K_u = 3$ closest neighbors. To generate multiple resolutions of the same input point cloud, a downsampling ratio of 2 is used for every additional resolution.

2.1.1 PointNet++

In all our experiments, we choose to report the performance of the multi-scale PointNet++ (MSG PN++) as it is reported to beat its alternative versions in the original paper on all tasks. We code our own implementation of PointNet++ in Pytorch and choose the same parameters as in the original code.

For segmentation task, the architecture is designed as follow:

Encoding1:

$$\text{S}(512)\rightarrow \begin{bmatrix} \text{rNN}(.1) \rightarrow \text{mLP}([32,32,64]) \rightarrow \text{MaxP} \\ \text{rNN}(.2) \rightarrow \text{mLP}([64,64,128]) \rightarrow \text{MaxP} \\ \text{rNN}(.4) \rightarrow \text{mLP}([64,96,128]) \rightarrow \text{MaxP} \end{bmatrix} \bigoplus$$

Encoding2:

$$\text{S}(128)\rightarrow \begin{bmatrix} \text{rNN}(.2) \rightarrow \text{mLP}([64,64,128]) \rightarrow \text{MaxP} \\ \text{rNN}(.4) \rightarrow \text{mLP}([128,128,256]) \rightarrow \text{MaxP} \\ \text{rNN}(.8) \rightarrow \text{mLP}([128,128,256]) \rightarrow \text{MaxP} \end{bmatrix} \bigoplus$$

Encoding3:

$$\text{S}(1)\rightarrow \text{mLP}([256,512,1024]) \rightarrow \text{MaxP}$$

Decoding1: $\text{Interp}(3)\rightarrow \text{mLP}([256,256])$

Decoding2: $\text{Interp}(3)\rightarrow \text{mLP}([256,128])$

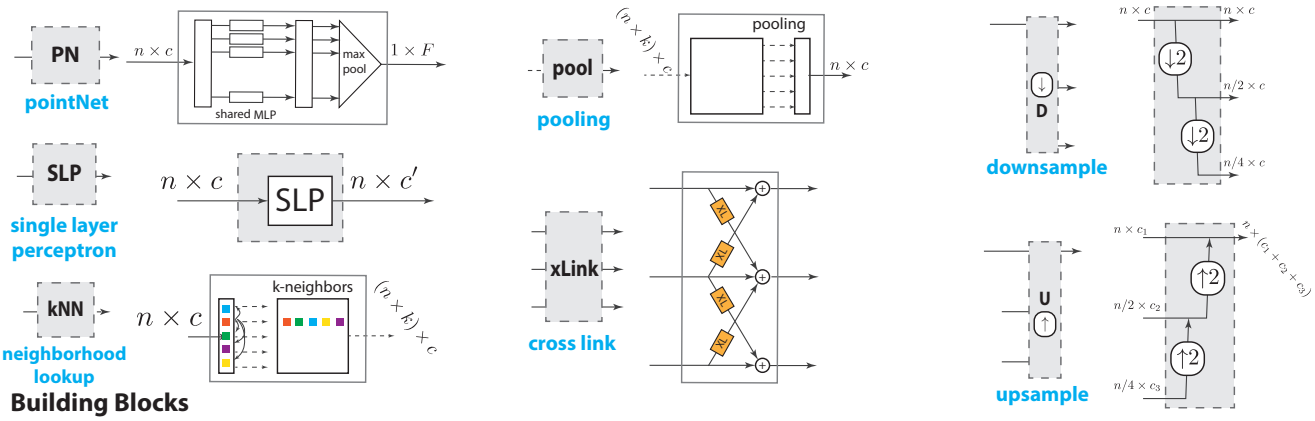

Figure 1. Elementary building blocks for point processing. Apart from standard neighborhood lookup, pooling and SLP layers, we introduce cross-link layers across scales, and propose multi-resolution up/down sampling blocks for point processing. PointNet module combines a stack of shared SLP (forming an MLP) to lift individual points and then performs permutation-invariant local pooling.

Decoding3: Interp(3)$\rightarrow$ mLP($[128, 128]$)
Classification: Lin(512)$\rightarrow$ Drop(.7)$\rightarrow$ Lin($\text{nb}_{\text{classes}}$)
We omit here skiplinks for sake of clarity: they connect encoding and decoding modules at the same scale level.

2.1.2 mRes

The mRes architecture consists in changing the way the sampling is done in the network to get a multi-resolution approach (see Fig. 2). We provide the details only for the encoding part of the network as we keep the decoding part unchanged from PointNet++.

Encoding1:

$$\begin{bmatrix} S(512) \rightarrow \text{rNN}(.1) \rightarrow \text{mLP}([32, 32, 64]) \rightarrow \text{MaxP} \\ S(256) \rightarrow \text{rNN}(.2) \rightarrow \text{mLP}([64, 64, 128]) \rightarrow \text{MaxP} \\ S(128) \rightarrow \text{rNN}(.4) \rightarrow \text{mLP}([64, 96, 128]) \rightarrow \text{MaxP} \end{bmatrix} \bigoplus$$

Encoding2:

$$\begin{bmatrix} S(128) \rightarrow \text{rNN}(.2) \rightarrow \text{mLP}([64, 64, 128]) \rightarrow \text{MaxP} \\ S(96) \rightarrow \text{rNN}(.4) \rightarrow \text{mLP}([128, 128, 256]) \rightarrow \text{MaxP} \\ S(64) \rightarrow \text{rNN}(.8) \rightarrow \text{mLP}([128, 128, 256] \rightarrow \text{MaxP} \end{bmatrix} \bigoplus$$

Encoding3:
S(1)$\rightarrow$ mLP($[256, 512, 1024]$) $\rightarrow$ MaxP

Starting from this architecture, we add Xlink connections between each layer of each mLP to get our mResX architecture. A Xlink connection connects two neighboring resolutions to merge information at different granularity. On each link, we use a sampling module (either downsampling or upsampling) to match the input to the target resolution. We use two alternatives for feature combination: (i) concatenation, (ii) summation. In the later case, we add an additional sLP on each Xlink to map the input feature dimension to the target. To keep this process as lean as possible, we position the SLP at the coarser resolution, i.e. before the upsampling module or after the downsampling module.

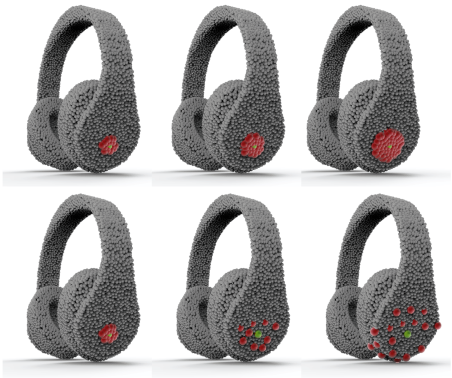

Figure 2. Comparison of multi-scale processing (top) with multi-resolution processing (down): multi-resolution processing allows us to process larger-scale areas while not increasing memory consumption, making it easier to elicit global context information.

2.1.3 LPN

Our convPN module can be seen as a point counterpart of 2D image convolution. To do so, the convPN module replaces the MLP with its pooling layer by a sequence of SLP-Pooling modules.

To simplify the writing, we adopt the additional notations:

- *Sampling* block $S([s_1, s_2, .., s_n]^T)$ where we make a sampling of s_i points on each resolution i. When only one resolution is available as input, the block $S([., s_1, s_2, ..., s_{n-1}]^T)$ will sequentially downsample the input point cloud by s_1, s_2, .. points to create the desired number of resolutions.

- *Convolution* block $C([r_1, r_2, ..., r_n]^T)$ is composed itself of three operations for each resolution i: neighbor-

hood lookup to select the r_iNN for each points, an sLP layer of the same size as its input and a max-pooling.

- *Transition* block $T([t_1, t_2, ..., t_n]^T)$ whose main role is to change the channel dimension of the input to the one of the convolution block. An sLP of ouput dimension t_i will be apply to the resolution i.

Residual connections are noted as *.
Encoding1:

$$S\begin{bmatrix}.\\512\\256\end{bmatrix} \rightarrow T\begin{bmatrix}32\\64\\64\end{bmatrix} \rightarrow C^*\begin{bmatrix}.1\\.2\\.4\end{bmatrix} \rightarrow T\begin{bmatrix}32\\64\\96\end{bmatrix} \rightarrow C^*\begin{bmatrix}.1\\.2\\.4\end{bmatrix} \rightarrow T\begin{bmatrix}64\\128\\128\end{bmatrix} \rightarrow C^*\begin{bmatrix}.1\\.2\\.4\end{bmatrix} \rightarrow S\begin{bmatrix}512\\256\\128\end{bmatrix} \rightarrow \bigoplus$$

Encoding2:

$$S\begin{bmatrix}.\\128\\96\end{bmatrix} \rightarrow T\begin{bmatrix}64\\128\\128\end{bmatrix} \rightarrow C^*\begin{bmatrix}.2\\.4\\.8\end{bmatrix} \rightarrow C^*\begin{bmatrix}.2\\.4\\.8\end{bmatrix} \rightarrow T\begin{bmatrix}128\\256\\256\end{bmatrix} \rightarrow C^*\begin{bmatrix}.2\\.4\\.8\end{bmatrix} \rightarrow S\begin{bmatrix}128\\96\\64\end{bmatrix} \rightarrow \bigoplus$$

Encoding3:
$\mathrm{S}(1) \rightarrow \mathrm{mLP}([256, 512, 1024]) \rightarrow \mathrm{MaxP}$

Note here that there is no *Transition* block between the first two C blocks in the Encoding2 part. This is because those two *Convolution* blocks work on the same feature dimension.

We also add Xlinks inside each of the C blocks. In this architecture, the features passing through the Xlinks are combined by summation and follow the same design as for mResX.

In the case of SLPs, using the on-the-fly re-computation of the neighborhood features tensor has a significant positive impact on both the forward and backward pass by means of a simple adjustment. Instead of applying the SLP on the neighborhood features tensor, we can first apply the SLP on the flat feature tensor and then reconstruct the neighborhood just before the max-pooling layer (Algorithm 1). The same can be used for the backward pass (see Algorithm 2).

2.1.4 Deep LPN

Our deep architecture builds on LPN to design a deeper architecture. For our experiments, we double the size of the encoding part by repeating each convolution block twice. For each encoding segment, we position the sampling block after the third convolution block, so that the first half of the convolution blocks are processing a higher resolution point cloud and the other half a coarser version.

Algorithm 1: Low-memory grouping - Forward pass

Data: Input features tensor $\mathcal{T}_f$ $(N \times R^D)$, input spatial tensor $\mathcal{T}_s$ $(N \times R^3)$ and indices of each point's neighborhood for lookup operation $\mathcal{L}$ $(N \times K)$
Result: Output feature tensor $\mathcal{T}_f^o$ $(N \times R^{D'})$

begin
  /* Lifting each point/feature to $R^{D'}$ */
  $\mathcal{T}_{f'} \longleftarrow$ **SLP**$_f(\mathcal{T}_f)$
  $\mathcal{T}_{s'} \longleftarrow$ **SLP**$_s(\mathcal{T}_s)$
  /* Neighborhood features $(N \times R^{D'} \rightarrow N \times R^{D'} \times (K+1))$ */
  $\mathcal{T}_{f'}^K \longleftarrow$ **IndexLookup**$(\mathcal{T}_{f'}, \mathcal{T}_{s'}, \mathcal{L})$
  /* Neighborhood pooling $(N \times R^{D'} \times (K+1) \rightarrow N \times R^{D'})$ */
  $\mathcal{T}_{f'}^o \longleftarrow$ **MaxPooling**$(\mathcal{T}_{f'}^K)$
  FreeMemory$(\mathcal{T}_{s'}, \mathcal{T}_{f'}, \mathcal{T}_{f'}^K)$
  return $\mathcal{T}_{f'}^o$
end

Algorithm 2: Low-memory grouping - Backward pass

Data: Input features tensor $\mathcal{T}_f$ $(N \times R^D)$, input spatial tensor $\mathcal{T}_s$ $(N \times R^3)$, gradient of the output $\mathcal{G}_{out}$ and indices of each point's neighborhood for lookup $\mathcal{L}$ $(N \times K)$
Result: Gradient of the input $\mathcal{G}_{in}$ and gradient of the weights $\mathcal{G}_w$

begin
  /* Gradient Max Pooling $(N \times R^{D'} \rightarrow N \times R^{D'} \times (K+1))$ */
  $\mathcal{G}_{out}^{mp} \longleftarrow$ **BackwardMaxPooling**$(\mathcal{G}_{out})$
  /* Flattening features $(N \times R^{D'} \times (K+1) \rightarrow N \times R^{D'})$ */
  $\mathcal{G}_{out}^{fl} \longleftarrow$ **InverseIndexLookup**$(\mathcal{G}_{out}^{mp}, \mathcal{L})$
  /* Gradient wrt. input/weight */
  $\mathcal{G}_w, \mathcal{G}_{in} \longleftarrow$ **BackwardSLP**$(\mathcal{T}_f, \mathcal{T}_s, \mathcal{G}_{out}^{fl})$
  FreeMemory$(\mathcal{T}_f, \mathcal{T}_s, \mathcal{G}_{out}, \mathcal{G}_{out}^{mp}, \mathcal{G}_{out}^{fl})$
  return $(\mathcal{G}_{in}, \mathcal{G}_w)$
end

2.2. DGCNN based architecture

Starting from the authors' exact implementation, we swap each edge-conv layer, implemented as an MLP, by a sequence of single resolution convPN blocks. This set of convPN blocks replicates the sequence of layers used to design the MLPs in the original implementation.

To allow the use of residual links, a transition block is placed before each edge-conv layer to match the dimension of both ends of the residual links.

2.3. SpiderCNN based architecture

A SpiderConv block can be seen as a bilinear operator on the input features and on a non-linear transformation of

the input points. This non-linear transformation consists of changing the space where the points live in.

In the original architecture, an SLP is first applied to the transformed points to compute the points' Taylor expansion. Then, each output vector is multiplied by its corresponding feature. Finally a convolution is applied on the product. Therefore, the neighborhood features can be built *on-the-fly* within the block and deleted once the outputs are obtained. We thus modify the backward pass to reconstruct the needed tensors when needed for gradient computation.

2.4. PointCNN based architecture

For PointCNN, we modify the χ-conv operator to avoid having to store the neighborhood features tensors for the backward pass. To do so, we make several approximations from the original architecture.

We replace the first MLP used to lift the points by a sequence of convPN blocks. Thus, instead of learning a feature representation per neighbor, we retain only a global feature vector per representative point.

We change as well the first fully connected layer used to learn the χ-transformation matrix. This new layer now reconstructs the neighborhood features *on-the-fly* from its inputs and deletes it from memory as soon as its output is computed. During the backward pass, the neighborhood features tensor is easily rebuilt to get the required gradients.

We implement the same trick for the convolution operator applied to the transformed features. We further augment this layer with the task of applying the χ-transformation to the neighborhood features once grouped.

Finally, we place transition blocks between each χ-conv layer to enable residual links.

2.5. Implementation details

In all our experiments , we process the dataset to have the same number of points N for each sample. To reach a given number of points, input pointclouds are downsampled using the furthest point sampling (FPS) algorithm or randomly upsampled.

We keep the exact same parameters as the original networks evaluated regarding most of parameters.

To regularize the network, we interleave a dropout layer between the last fully connected layers and parameterize it to zero 70% of the input neurons. Finally, we add a weight decay of 5e-4 to the loss for all our experiments.

All networks are trained using the Adam optimizer to minimize the cross-entropy loss. The running average coefficients for Adam are set to 0.9 and 0.999 for the gradient and its square, respectively.

References

[1] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. ShapeNet: An Information-Rich 3D Model Repository. Technical Report arXiv:1512.03012 [cs.GR], Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015. 1

[2] Angela Dai, Angel X. Chang, Manolis Savva, Maciej Halber, Thomas Funkhouser, and Matthias Nießner. Scannet: Richly-annotated 3d reconstructions of indoor scenes. In *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*, 2017. 1

[3] Yangyan Li, Rui Bu, Mingchao Sun, Wei Wu, Xinhan Di, and Baoquan Chen. Pointcnn: Convolution on x-transformed points. 2018. 2, 3

[4] Kaichun Mo, Shilin Zhu, Angel X. Chang, Li Yi, Subarna Tripathi, Leonidas J. Guibas, and Hao Su. Partnet: A large-scale benchmark for fine-grained and hierarchical part-level 3d object understanding. *CoRR*, abs/1812.02713, 2018. 1

[5] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *CVPR*, 1(2):4, 2017. 1

[6] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In *NIPS*, pages 5099–5108, 2017. 2, 3

[7] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E Sarma, Michael M Bronstein, and Justin M Solomon. Dynamic graph cnn for learning on point clouds. *arXiv preprint arXiv:1801.07829*, 2018. 2, 3

[8] Yifan Xu, Tianqi Fan, Mingye Xu, Long Zeng, and Yu Qiao. Spidercnn: Deep learning on point sets with parameterized convolutional filters. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 87–102, 2018. 2, 3