Ursa: A Neural Network for Unordered Point Clouds Using Constellations

Mark B. Skouson, Brett Borghetti, and Robert Leishman

Department of Electrical and Computer Engineering
United States Air Force Institute of Technology
Wright Patterson AFIT, OH 45433

Abstract. This paper describes a neural network layer, named Ursa, that uses a constellation of points to learn classification information from point cloud data. Similar to new methods such as PointNet [1], this architecture works directly on d-dimensional points rather than first converting the points to a d-dimensional volume. We use an Ursa layer, followed by a series of dense layers, to classify 2D and 3D objects from point clouds. Experiments on ModelNet40 and MNIST data show classification results comparable with current methods, while reducing the training parameters by over 50 percent.

1 Introduction

A large bulk of the recent computer vision research has focused on applying artificial neural networks to 2D images. More recently, a growing research area focuses on applying neural networks to 3D physical scenes. Point clouds or point sets are a common format for representing 3D data since some sensors, including laser-based systems, collect scene data directly as point clouds. Voxelization is a straightforward way of applying powerful deep convolutional neural network techniques to point clouds, as is done in VoxNet [2] and 3DShapeNets [3]. Voxelization, however, is not always desirable because point clouds can, in many cases, represent structural information more compactly and more accurately than voxelized alternatives.

In contrast to voxelization methods, PointNet [1] and others have developed architectures that operate directly on point clouds, including ECC [4], Kd-Net [5], DGCNN [6], and KCNet [7]. This research adds to the growing body of knowledge about learning on point sets.

This paper describes a neural network layer (Ursa layer) that accepts a point cloud as input and efficiently yields a single feature vector, which is both agnostic to the ordering of the points in the point cloud and encodes the dimensional features of every point. This output feature vector is an efficient representation of the entire point cloud - an observation which can be used for classification (or other machine learning tasks) in later portions of the network. The layer's trainable parameters are centroids, and each centroid has the same dimension as a point in the point cloud. For the remainder of this paper, in order to distinguish

the centroids from the points in the point cloud, the centroids will be referred to as stars, and the collection of stars in the URSA layer will be referred to as a constellation. The output of the layer is a feature vector with length equal to the number of constellation stars, which us used in the later layers of the neural network to inform the classification output. Another important characteristic of this approach is that it does not require a preprocessing step - the Ursa layer is trained as part of the overall network structure using backpropagation and gradient descent learning.

The Ursa layer is invariant to the ordering of the input points. The Ursa layer is not inherently invariant to shift, scale, or rotation; rather, it relies on demonstrations of those types of variations (possibly through data augmentation) in the input data during training to learn these variations. The output of the Ursa layer is a global shape descriptor of the point cloud that is fed to later layers in the classifier to classify the point cloud.

Experiments on this architecture show the classification accuracy is comparible to current point cloud-based classifiers, but with a significantly smaller model size. We have tested the Ursa architecture with various distance functions and various numbers of constellation stars using MNIST (2D) data and ModelNet40 (3D) data. Experimentally, the best distance measure was dependent on the data set. We found that for both data sets, too few or too many stars degraded performance. We found the optimal number of stars for both the 2D and 3D datasets was between 265 and 512 stars in the constellation.

2 Selected Related Works

The work presented herein is informed by the PointNet [1] research and architecture. In [1], Qi, et al., introduce the concept of symmetric functions for unordered points. A symmetric function aggregates the information from each point and outputs a new vector that is invariant to the input order. Example symmetric operators are summation, multiplication, maximum, and minimum. Alternatives to a symmetric function for point order invariance would be to sort the input into a canonical order or augment the training data with all kinds of permutations. PointNet uses a 5-layer MLP to convert the inputs points to a higher-dimensional space, then uses max pooling as the symmetric function to generate a single global feature, which is then fed through 3-layer MLP for classification. The architecture we experimented on in this paper replaces PointNet's first 5 MLP layers and max pooling layer with a single Ursa layer. The Ursa layer generates a global feature and, as in PointNet, uses a 3-layer MLP for classification. As with PointNet, the Ursa layer's output is invariant to the order of the input data.

This work is also closely related to the KCNet architecture [7]. KCNet uses a concept similar to our Ursa constellation layer, which they call kernel correlation. Kernel correlation has been used for point set registration, including by [8]. Whereas [8] attempts to find a transformation between two sets of points to align them, Ursa and KCNet allow each point in the constellation (or kernel) to freely move and adjust during training. The KCNet architecture maintains all the layers

of PointNet, and augments them by concatenating kernel correlation information to the intermediate vectors within the 5 layers of MLP. In a forward pass in PointNet, each input point is treated independently of all other points until the global max pooling layer, but that is not the case for KCNet. KCNet uses a set of kernels that operate on local subsets of the input points using a K-nearest neighbor approach. The kernels are trained to learn local feature structures important for classification and segmentation. Thus, KCNet improves on PointNet by adding additional local geometric structure and feature information prior to global max pooling.

There are several difference between the KCNet and the Ursa-based architecture used for this paper. The KCNet kernel correlation produces a scalar value while the Ursa layer produces a vector. KCNet uses several kernels at the local level to augment the PointNet architecture. Our architecture uses a single star constellation at the global level to replace the first several layers of PointNet.

Other deep learning methods that operate on point clouds include Dynamic Graph Convolutional Neural Networks (DGCNNs) [6], Edge-Conditioned Convolution (ECC) [4], Kd-Networks [5], and OctNet [9]. These methods organize the data into graphs. In the cases of [6] and [4], the graphs are based on a vertex for each point and edges that define a relationship between the vertex and near neighbors, and weighted sum operations that operate on vertices and edges of the graph. The DGCNN architecture in [6] is quite similar to the PointNet structure, but where the multi-layer perceptron layers are replaced with Edge Convolution Layers. Both [5] and [9] use non-uniform spatial structure to partition the input space, and they also used weighted sum operations. In contrast to these methods, the learning in the Ursa layer is not stored in the weights of a weighted sum operation. Instead the learning is stored in the locations of a set of constellation stars as will be described in the next section.

This work explores the use of radial basis functions (RBFs) in point cloud classification and so bears some commonality with RBF networks [10,11,12,13]. RBFs are able to project an input space into a higher-dimensional space . It does this through a radial function, which varies with distance from a central point, and a set of vectors known as RBF centers. In common usage, a function $f(x)$ of an input vector x can be modeled as a weighted sum of a radial basis function, ϕ, of the distances between the input vector and m RBF centers, q_i:

$$f(x) \approx \sum_{i=1}^{m} \omega_i \left(\phi(||x - q_i||)\right) \tag{1}$$

where $||\cdot||$ is the L2 norm.

In general, the ω_i, the q_i, and ϕ can be selected or trained to fit the RBF network to the function. In practice, ϕ and the q_i are usually first selected, then the ω_i are adjusted or trained to fit the data. While our work herein does explore the use of RBFs, it differs from RBF networks. Rather than computing a weighted sum of RBF outputs to determine a classification of a single input vector (a point cloud), we use an RBF to transform a set of input vectors to a higher-dimensional feature vector as will be described below.

3 Method

We now describe the Ursa layer and a neural network architecture that uses a Ursa layer to classify objects. The overall classification architecture is shown in Fig. 1. It is an Ursa layer followed by a three-layer fully connected (dense) multi-layer perceptron classifier. Compared to the PointNet architecture used by several researchers [1,14,6,7], the Ursa layer replaces the first 5 MLP layers of PointNet architecture, while maintaining the last three-layer MLP portion. Maintaining an end structure similar to other methods aids in comparison.

During training we make use of data augmentation at the input and data dropout just prior to the final MLP layer. Because the final layers are straightforward, the remainder of this section will focus on only the the Ursa layer. Parameters used during implementation are discussed in Section 4.

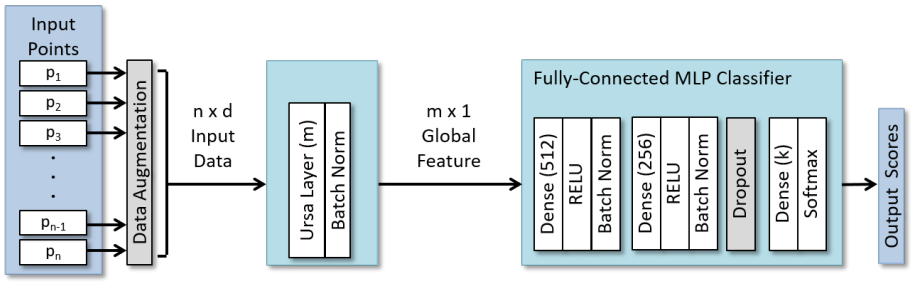


Fig. 1. The Ursa architecture. The first hidden layer is an Ursa layer. The remaining three hidden layers are fully-connected (dense) layers. Data augmentation and dropout (the gray boxes) are used only during training.

To define the Ursa layer, consider a set of n d-dimensional input points in $\Re^d$ that make up a point cloud $P = \{p_1, ..., p_n\} \subset \Re^d$. P is the input to the Ursa layer. Within the layer is a constellation of m stars, with the same dimensionality as the input points, $Q = \{q_1, ..., q_m\} \subset \Re^d$. The output of the layer is an mx1 vector $V = \{v_1, ..., v_m\} \subset \Re$. The Ursa layer converts a set of n d-dimensional points into an m-dimensional feature vector. There are mxd trainable parameters within the layer.

As mentioned earlier, RBFs have the ability to convert vectors to a higher-dimensional feature space. The Ursa layer makes use of RBFs. In fact, we investigated the use of three different candidate RBFs and compared their effectiveness. The first function explored is the Gaussian RBF, $\phi(\cdot) = \exp(-(\cdot)^2/(2\sigma^2))$, with which the relationship between P, Q, and V is

$$v_i = \sum_{j=1}^{n} \exp\left(-\frac{\|p_j - q_i\|^2}{2\sigma^2}\right), \qquad i = 1, ..., m \tag{2}$$

where σ controls the "width" of the function. In this paper, σ is considered a user-selected hyper-parameter, potentially tunable in cross-validation. So, each input point's contribution to the i-th entry in the output vector is a function its distance to the i-th constellation star, with the contribution decreasing according to the Gaussian function as the point is farther away. In other words, v_i accumulates into a single scalar value the weighted distance information between the ith star and each point in the point cloud. The summation provides the symmetry that makes the output of the layer invariant to the ordering of the input points.

The second function explored was an exponential decaying RBF, applied according to Eq. 3, where the hyper parameter λ controls the width of the function similar to σ in Eq. 2.

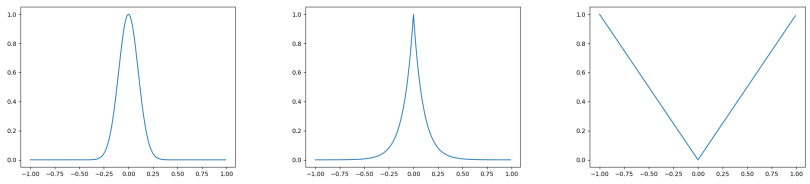
$$v_i = \sum_{j=1}^{n} \exp\left(-\lambda \left\| p_j - q_i \right\|\right), \qquad i = 1, ..., m \tag{3}$$

The exponential decay has the effect of more rapidly depreciating the contribution of each point to a star's feature output the further they are from the star's location, as can be seen in Figure 2.

Finally, we experimented with linear RBF applied according to Eq. 4.

$$v_i = \min_{1 \le j \le n} \left\| p_j - q_i \right\|, \qquad i = 1, ..., m \tag{4}$$

In this case, the symmetry is provided by the minimum function. The effect of Eq. 4 is that v_i is the distance from constellation star q_i to its input point from the point cloud. This is the RBF that provides the most efficient computation of the three we investigated. The relative effectiveness of all three measures is shown in Section 4. We refer to Eqs. 2, 3, and 4 as the Gaussian distance function, the exponential distance function, and the minimum distance function, respectively, throughout this paper.



(a) (b) (c)

Fig. 2. Visual comparison of RBFs, where the x-axis is the distance from the centroid. (a) Gaussian RBF, (b) exponential decay RBF, (c) RBF for the minimum distance function.

The Ursa layer is followed by a three-layer MLP. The non-linearity for each MLP layer is the ReLU function, except for the final layer, which uses softmax.

Each ReLU is followed by a batch normalization. The Ursa layer does not require a ReLU function afterward because the v_i in Eqs. 2, 3, and 4 are already always non-negative. Additionally, the Ursa distance measures defined by these equations are not matrix multiplies, which require a separate non-linearity afterward to enable the network to emulate non-linear function of the input; the L2 norm within the computations provides an inherent nonlinear component to the layer. All three distance measures are differentiable and are trained as part of of the overall back-propagation of the entire network. For our implementation, the gradient was computed using the standard tensorflow gradient calculations.

4 Experiments

We evaluated the described Ursa network architecture for classification of 3D and 2D objects. For 3D data, we used the ModelNet40 shape database [3]. For 2D data, we used the MNIST handwritten character recognition database [15] converted to 2D point clouds.

For the ModelNet40 data, we used 2048 points per object evenly sampled on mesh faces and normalized into the unit sphere as provided by [14]. To convert an MNIST image to a 2D point cloud, we used the coordinates of all pixels with values larger than 128. We used 312 points per character, which was the maximum number of pixels greater than 128 for any MNIST image. For those images with fewer than 312 points, we randomly repeated enough points from the set to reach 312 points.

During training for both 3D and 2D data, we augmented the data by scaling the shape to between 0.8 and 1.25 of the unit-sphere size with a random uniform distribution; rotating the shape between -0.18 and 0.18 radians along each angular axis with a random normal distribution (clipped) with standard deviation 0.06; shifting the shape in every dimension between -0.1 and 0.1 away from its original position with a random uniform distribution; and adding jitter between -0.05 and 0.05 to each point according to a random normal distribution (clipped) with standard deviation 0.01. Also during training we added a dropout layer with a dropout rate of 0.3 just before the last dense layer. We chose σ in Eq. 2 to be 0.1 and λ in Eq. 3 to be 10. Additional tuning of these hyper-parameters may improve performance.

The trainable parameters within the Ursa layer were initialized by randomizing them according to a uniform distribution between ± 1 in each dimension. The trainable parameters for the dense layers were initalized using the glorot uniform method. Future research may consider other Ursa layer initialization methods such as uniformly distributing across the space [12] or using information from the input data , e.g. k-means clustering techniques.

Experiments explored the three feature space transforms in Eqs. 2, 3, and 4 for several values of m, the number of constellation stars. We evaluated classification performance for $m = 32$, 64, 128, 256, 512, and 1024. We ran 10 independent tests with each of the distance measures at each value of m and plotted the average accuracy in Figs. 3 and 4.

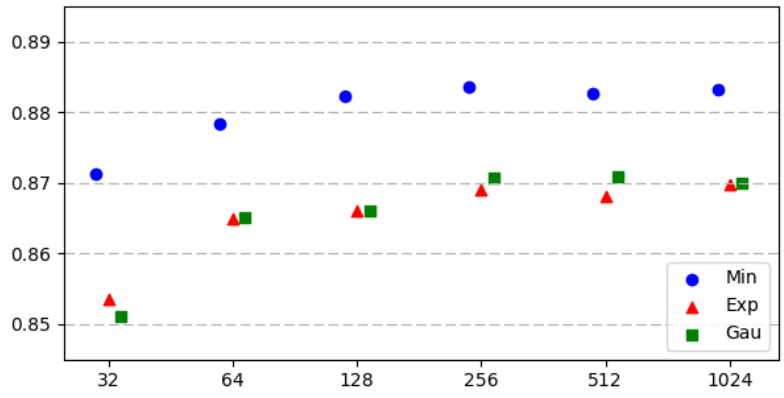


Fig. 3. The performance of the Ursa architecture on ModelNet40 for each of the distance measures with respect to the number of constellations stars. The y-axis has been zoomed in to highlight differences. The minimum distance measure slightly outperforms the other two methods. Constellations with 256 stars were best when using the minimum distance measure, 512 stars was best for the Gaussian distance measure, and 1024 was best for the exponential distance measure.

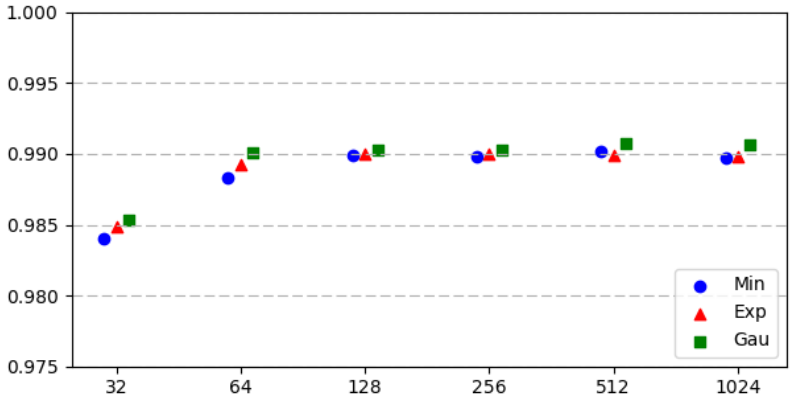


Fig. 4. The performance of the Ursa architecture on the MNIST data for each of the distance measures with respect to the number of constellations stars. The y-axis has been zoomed in to highlight difference. The Gaussian distance measure slightly outperforms the other two methods. Constellations with 512 stars were best for the minimum and Gaussian distance measures, and 256 was best for the exponential distance measure.

5 Results and Discussion

An analysis of Figs. 3 and 4 shows a general trend of performance improving as the constellation grows to 256 stars, but leveling out or worsening after 512 stars. In our tests, the best average performance was with 256 or 512 stars for all sets of experiments except for the exponential distance function on ModelNet40 data, which was best with 1024 stars. It is interesting that our experiments did not show any clear difference in the number of stars needed based on the dimensionality of the data (3D vs 2D), number of points per shape (2048 vs 312), or number of possible classes (40 vs 10).

The minimum distance function performed better on the ModelNet40 data, while the Gaussian distance function performed slightly better on the MNIST data. This may be a function of the number of points per shape. The minimum function may be more effective in situations with many points. It is interesting to note that PointNet [1] performed better using max pooling rather average pooling or an attention weighted sum to provide the symmetry (point order invariance) property.

Table 1 shows how the evaluated Ursa network compares to other classification methods on the same data sets. Ursa achieved impressive results, especially considering the Ursa model uses far fewer parameters than any other method we compared. While some more sophisticated methods outperformed Ursa, this paper demonstrates the viability, effectiveness, and potential of the Ursa layer and the constellation approach to pattern learning.

Table 1. Classification results and model size comparisons for various methods. Accuracy results for the Ursa method are mean of 10 independent runs for each data set. Best Ursa single run performances are in parentheses. Accuracy results for the other methods are adapted from [7] and [6]

	ModelNet40 Accuracy (in percent)	MNIST Accuracy (in percent)	Model Size (x1M params)
LeNet5 [15]	–	99.2	–
3DShapeNets [3]	84.7	–	–
VoxNet [2]	85.9	–	–
Subvolume [16]	89.2	–	–
ECC [3]	87.4	99.4	–
PointNet (Baseline) [1]	87.2	98.7	0.8
PointNet [1]	89.2	99.2	3.5
PointNet++ [14]	90.7	99.5	1.0
KCNet [7]	90.0	99.3	0.9
Kd-Net [5]	91.8	99.1	2.0
DGCNN [6]	92.2	–	1.8
Ursa (Ours)	88.4 (88.9)	99.1 (99.2)	0.4

The 2D data was used to demonstrate the movement over time of the Ursa constellation stars during training (see Figs. 5 and 6). We did this using both the minimum distance measure of Eq. 4 and the Gaussian distance measure of Eq. 2, as shown in Figs. 5 and 6, respectively. These figures show the constellation stars adjusting over time to span the space. The constellation resulting from the minimum distance measure appears more compact in the center and the stars are more spread out toward the outside. On the other hand, the Gaussian distance measure constellation is more uniformly distributed throughout the space. Also, the resulting range is larger for the minimum distance measure than for the Gaussian distance measure.

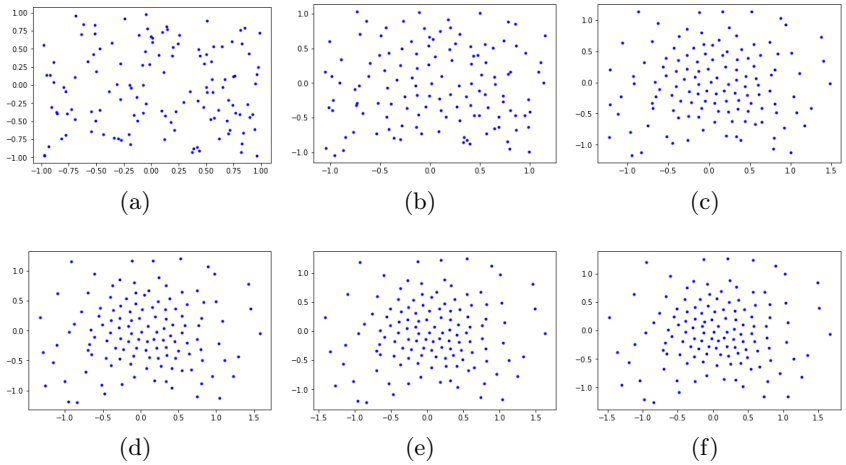


Fig. 5. Depiction of the Ursa constellation stars and their positions during training for for m=128 and using the minimum distance measure. Sub-figures show (a) random initialization, (b) after 10 training epochs, (c) after 100 epochs, (d) after 200 epochs, (e) after 300 epochs, and (f) after 500 epochs.

6 Conclusion

This paper has presented an Ursa neural network layer and demonstrated its effectiveness and viability for classification of point cloud data. The Ursa layer stores information in the form of constellation points, rather than a set of multiplicative weights in a matrix. While other more sophisticated methods achieved higher classification rates on the data sets we tested, all other methods we compared used at least twice the model parameters of the baseline Ursa network.

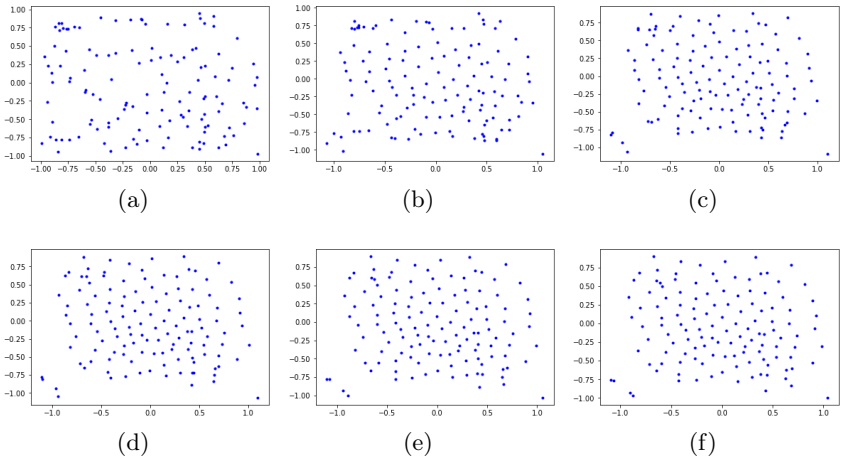


Fig. 6. Depiction of the Ursa constellation stars and their positions during training for for m=128 and using the Gaussian distance measure. Sub-figures show (a) random initialization, (b) after 10 training epochs, (c) after 100 epochs, (d) after 200 epochs, (e) after 300 epochs, and (f) after 500 epochs.

References

1. C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "PointNet: Deep learning on point sets for 3D classification and segmentation," in *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-Janua, pp. 77–85, 2017.
2. D. Maturana and S. Scherer, "VoxNet: A 3D Convolutional Neural Network for real-time object recognition," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 922–928, IEEE, 9 2015.
3. Z. Wu, "3d shapenets: A deep representation for volumetric shapes," in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015*, p. 1912–1920, 2015.
4. M. Simonovsky and N. Komodakis, "Dynamic edge-conditioned filters in convolutional neural networks on graphs," in *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, pp. 29–38, 2017.
5. R. Klokov and V. Lempitsky, "Escape from Cells: Deep Kd-Networks for the Recognition of 3D Point Cloud Models," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2017-Octob, pp. 863–872, 2017.
6. Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic Graph CNN for Learning on Point Clouds," in *arXiv:1801.07829*, 2018.
7. Y. Shen, C. Feng, Y. Yang, and D. Tian, "Mining Point Cloud Local Structures by Kernel Correlation and Graph Pooling," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2018.
8. Y. Tsin and T. Kanade, "A Correlation-Based Approach to Robust Point Set Registration," in *European Conference on Computer Vision*, 2004.

9. G. Riegler, A. O. Ulusoy, and A. Geiger, "OctNet: Learning deep 3D representations at high resolutions," in *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-Janua, pp. 6620–6629, 2017.
10. M. D. M. D. Buhmann, *Radial basis functions : theory and implementations*. Cambridge University Press, 2003.
11. M. Orr, *Introduction to Radial Basis Function Networks*. 1996.
12. D. S. Broomhead and D. Lowe, "Multivariable Functional Interpolation and Adaptive Networks," *Complex Systems*, vol. 2, pp. 3221–355, 1988.
13. S. Chen, C. Cowan, and P. Grant, "Orthogonal least squares learning algorithm for radial basis function networks," *IEEE Transactions on Neural Networks*, vol. 2, pp. 302–309, 3 1991.
14. C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space," in *Neural Information Processing Systems Conference*, 2017.
15. Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
16. C. R. Qi, H. Su, M. Niessner, A. Dai, M. Yan, and L. J. Guibas, "Volumetric and Multi-View CNNs for Object Classification on 3D Data," *Proceedings - 29th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016*, 2016.