Scalable Approximate KNN for Latent Graph Learning

Federico Rios
Politecnico di Milano
federico.rios@mail.polimi.it

Francesco Puddu
Politecnico di Milano
francesco.puddu@mail.polimi.it

Abstract

The field of Latent Graph Learning addresses problems where learning an implicit graph over the input data can help tackle the task more effectively. During the graph creation process, the standard approach leverages a pair-wise distance matrix, which results in a scalability issue given its quadratic cost. In this paper, we illustrate the problem by analysing some significant works from the field, and then present our proposed solution. We develop an efficient approximate KNN algorithm that leverages a hierarchical data structure, and compare it to the naive approach in terms of graph quality, time-space complexity, and training performance. We show that we significantly outperform standard solutions in memory efficiency.

1. Introduction

The field of Machine Learning on graphs is now firmly established in the scientific landscape thanks to the abundance of graph-structured data in various fields such as social networks, medicine or physics. In some settings, however, the connections between data points are not explicitly known beforehand. A typical example comes from the field of biology, where interactions between molecules are often only partially known a-priori, but constitute a key element for downstream tasks.

The general problem of inferring a graph from input data motivates research in the field called Latent Graph Learning (LGL). Solutions in this field see graph creation as part of the learning pipeline, which can be analyzed on these two dimensions:

- How the graph is created from the input. i.e. how the data is connected together.
- How the graph is used for the downstream task, which typically consists of techniques from the field of Graph Representation Learning.

In this paper we focus on the first problem, addressing a well-known and widespread limitation. Intuitively, the ideal outcome of LGL is an efficient sparse graph containing all and only the most meaningful connections among the data with respect to the final task. While apparently simple, this process poses two key challenges:

- If no a-priori assumptions can be made on each node's connected neighbors, every edge must be evaluated as we have no information that allows us to exclude it.
- The act of choosing which edges to assign to each node, and which edges to avoid, amounts to a discrete decision. This discreteness poses a problem when we want to embed it as a part of a deep learning pipeline, which inherently requires continuous and differentiable operations to ensure gradient flow.

Both of these challenges can be solved by assuming a fully connected topology for the graph, which we can express as a dense adjacency matrix $\mathcal{A}$. This is in line with the idea of evaluating every possible edge, which translates to performing some operation for each element $\mathcal{A}_{i,j}$. Moreover, if we allow continuous values for the elements of $\mathcal{A}$, this approach can also be easily continuous, since no discrete decision is required.

While this is a simple and elegant solution, it is important to note that computing $\mathcal{A}$ results in quadratic time and memory complexity, since for a dataset of N nodes the size of $\mathcal{A}$ will be N^2. This scalability issue can become decisive in several areas of application of LGL techniques, such as in the point-cloud domain, in which datasets can reach sizes that are simply intractable with thise ï approach.

Our proposed solution explicitly addresses this problem by constructing a sparse graph via a hierarchical approach, efficiently approximating the result of the classical KNN algorithm. We show the results we obtained in the point-cloud domain, both in terms of performance relative to a KNN, and in terms of effectiveness for learning purposes by implementing the algorithm within a pre-existing LGL model.

2. Related work

We start with a brief overview of three solutions applying different LGL techniques on different domains.

With the aim of learning the dynamics of a system from the observed trajectories of the particles that comprise it, [4] considers the latent graph of the interactions between the particles themselves. The dataset that they work on contains the position and velocity of each particle at each discrete time step of a physics simulation. Moreover, the set of all possible types of connections is small and known a-priori, and the ground truth connections in each training example is available.

The authors implement an encoder-decoder architecture that is given the position and velocity of each particle at time t, and is trained to predict their position and velocity at time $t + 1$. Moreover, the graph that is produced by the encoder receives the ground truth connections as an additional training signal. This architecture can be viewed in Figure 1.

The encoder is modeled as a graph neural network, and the topology of the graph is taken as fully connected to simplify the task as explained in the Introduction. The node embeddings are initialized with the position and velocity of each particle, and then 2 rounds of graph convolutions are applied. The output of the encoder is a new set of node embeddings, and N^2 edge embeddings of dimension C, where C is the number of possible connection types, which is known a-priori. One of the possible types of connection encodes non-connection, to account for particles that may simply not be connected.

The logits in output for each edge are then perturbed with Gumbel noise and then passed through a softmax layer. These two steps approximate in a continuous way the process of discrete sampling of the edge type for each edge. The output of this sampling process is a vector $e_{i,j}$ of size C that is close to one hot for each edge, and the authors provide supervision on this graph by introducing a loss with respect to the ground truth edge types.

The decoder takes as input the node embeddings produced by the decoder and the edge embeddings produced by the soft sampling process. A different graph neural network is instantiated for each connection type, meaning that one message for edge type will be passed along an edge during a graph convolution. These typed messages will be weighted by the respective $e_{i,j,c}$, where i and j are the connected nodes and c is the edge class of the message. The final node embeddings are then passed through an MLP to produce a delta position and velocity at time $t + 1$. Since when time steps are small the deltas are close to zero, the authors predict more than a single step ahead by recursively applying the model. In this way, the error given by a trivial zero difference prediction compounds heavily over multiple time steps and is therefore strongly penalized.

We find another application of LGL in the field of medicine in [3], where the disease prediction task relies on the construction of a patient similarity graph. In this case, no ground truth data is available about the correct topology of the graph, which makes this a harder problem. The available data is a set of features for each patient, and the respective ground truth diagnosis.

The authors choose to first apply an MLP to the feature vectors, to get embedding vectors that will be used to create the graph. To this end, they compute the euclidean distance of each embedding to every other embedding, run it through a sigmoid variant and take the output as an adjacency matrix $\mathcal{A}$. The result will therefore be dense, of size N^2, and will contain values between 0 and 1.

This adjacency matrix is then used to perform multiple graph convolutions, each of which yields new patient embeddings. It's worth noting that these convolutions only update the node embeddings, not the edge embeddings, since no edge embeddings are present. Moreover, given that the size of the neighborhood of each node is known and identical, the graph convolution operation collapses onto simple matrix multiplications. The final node embeddings are then used to predict probabilities for each possible diagnosis class. The full architecture can be viewed at Figure 2.

Finally, [8] (DGCNN) operates in the point cloud domain, specifically targeting classification and segmentation tasks. The edges of the graph depend (similarly to [3]) on the distance in the latent space, and thus represent a semantic link between the points. The authors frame their contribution in a more general perspective, defining a module for LGL called 'Edge Convolution' (EdgeConv) with a two-step operation:

- Application of KNN on the current representation of the points, i.e. in latent space, to create a sparse graph in which connections are solely between each node and its neighbours. Note the fact that although the resulting graph is sparse, the classical KNN algorithm requires computation of the pair-wise distance matrix, which has a quadratic cost in the number of points.

- Updating the embeddings of the points via a message passing round, where each node updates its representation by aggregating with a permutation invariant operator (e.g. maximum) the messages from its neighbours. As is typically the case with graph neural networks, a message from a given edge is generated by a neural network from the representations of the two connected points and weighted with that edge's weight.

Interestingly, in a similar way to that observed in [4], the authors stress the benefits of iterating the application of EdgeConv to improve the quality of the graph and more in general of the learning process. The stacking of this module causes the latent graph to change dynamically during both training and inference. As the authors point out, these updates make the receptive field ultimately as large as the

diameter of the point cloud, without sacrificing the sparsity of the graph. In an attempt to further interpret the meaning of iteration, the authors point out how the model is actually learning to construct the graph, rather than considering it a constant fixed in advance.

The proposed architectures for point cloud classification and segmentation see the module being stacked 4 and 3 times respectively. For classification, the final node embeddings are aggregated globally to form a one-dimensional global descriptor, from which the classification scores are generated. The segmentation architecture extends the classification one by concatenating the global descriptor and all the EdgeConv outputs for each point. It outputs per-point classification scores for the semantic labels. The full architecture is shown in Figure 3.

A recurring aspect in the LGL field, which in fact unites the works mentioned in this section, is the computation of some value for each pair of data points as part of the strategy to create the latent graph between them. For the first paper, this value is an edge embedding, while for the other two it is an euclidean distance. The creation of the graph thus entails a cost in terms of time and memory during training and inference that scales quadratically with respect to the dimensionality of the dataset. While in terms of time, the possibility of using parallel processing architectures like GPUs proves very effective in practice, the same cannot be said in terms of memory, which we can identify as one of the most significant limitations of the field.

The problem of performing Approximate Nearest Neighbour (ANN) searches is extremely transversal, we therefore find numerous proposals in the literature. As detailed in [5], one of the main strands in the field is that of approaches based on hierarchical data structures. Among them [6] is a widely used algorithm configuration method that selects the most suitable implementation among well known ANN algorithms, such as hierarchical k-means and kd-trees. Our approach, which we will discuss in the following section, can be understood as related to this methods. However, while hierarchical k-means depends on many hyperparameters, ours doesn't and therefore no additional knowledge about the structure of the data is required. Kd-tree is one of the approaches that most closely resemble our own, and we improve on it by choosing the hyperplanes that cut the hyperspace in a more data-driven way. Within the same family of solutions is [2] (ANNOY), from which our approach takes the most inspiration. However, while ANNOY is designed to be very efficient in using its index to extract KNNs, the creation of the index itself is not especially fast. This algorithm would therefore be a poor fit for our problem, since we need to build the index 4 times for each forward pass of the network.

3. Proposed approach

For our proposed approach we extend the formulation of the latent graph creation problem presented in [8]. We consider an F-dimensional set of N points, denoted by $X = \{x_1\ ,\ ...\ ,\ x_n\} \subseteq \mathbb{R}^F$ The algorithm we propose aims to generate a directed graph $\mathcal{G} = (\mathcal{V},\ \mathcal{E})$ that locally represents the data, where $\mathcal{V} = \{1\ ,\ ...\ ,\ n\}$ and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ are the vertices and edges respectively. Note that in our setting as well, F generically denotes the dimensionality of a given layer of a deep architecture, as we aim to construct graphs in latent space as well as in input space.

The dimensions of interest in our solution are graph sparsity and space-time complexity. To obtain a sparse graph, similarly to the solution proposed in [8], we consider the degree of each vertex as an hyperparameter k. Although with this assumption the KNN algorithm used in the paper provides the exact solution to the problem, we cannot consider it for the scalability problems already described. Instead, we propose an alternative approximate version, which offers a competitive trade-off between local graph approximation and complexity.

3.1. Data Structure

Our approximate KNN exploits a hierarchical data structure, which we anticipate in this subsection to gently introduce the actual algorithm. We consider a binary tree T, in which each node $t_i \in T$ contains a dictionary with the following fields, described by resorting to the typical terminology of tree-based data structures:

- $point$: data point $x_i \in X$
- $left$: reference to the "left child" node
- $right$: reference to the "right child" node
- $parent$: reference to the "parent" node
- $axis$: an hyperplane $h_{lr} \in \mathbb{R}^F$, the role of which is deferred to the discussion of the algorithm

For ease of discussion, in the pseudocode of the algorithm we will make use of the "." operator consistently to many programming languages to denote subfields of the dictionary defined above, e.g. to assign node t_2 as the left child of node t_1 we will write: $t_1.\,left \leftarrow t_2$.

3.2. Algorithm

The proposed solution consists of two consecutive distinct procedures:

- Creation of the tree: $X \rightarrow T$
- Creation of the graph: $T \rightarrow \mathcal{G}$

See Algorithm 1 and Algorithm 2 below for the details.

Discussion. We can identify in the first algorithm a procedure for constructing a binary tree, with a hierarchical partitioning of space (a separating hyperplane is defined for each child pair). Each point x, during insertion, traverses the tree starting from the root. At every node it traverses, it encounter a hyperplane h that tells it whether it should descend on the right subtree or on the left. This choice is simply given by $h \cdot x$, which is positive on one side of the hyperplane and negative on the other. When it reaches a leaf or a node with only one child, it is added as a child node. When a node has 2 children, we compute the hyperplane that maximizes the distance to the children points, which become representatives of each side of their parent node's hyperplane. With this process, we cut the original space into smaller and smaller convex subsets as we descend into the tree. We create this tree by sampling without replacement from X until we have placed every point in the dataset.

With this strategy, we descend into the tree once for each point, building it as we go, and for each descent we pay on average a logarithmic cost in the cardinality of X. Experiments confirmed the assumption of reasonable balancing of the trees generated with this algorithm, thus supporting the claim of an $\mathcal{O}(n \log n)$ time complexity. Since the data structure stores a data point at each node, the spatial complexity is intuitively $\mathcal{O}(n)$.

Suppose we take a point x, and we traverse our tree choosing to descend along the right path when $h_t \cdot x > 0$ and the left path otherwise, as we did during the creation of the tree. We will arrive at a leaf at depth d, after making d binary choices. These d binary choices are identified by d hyperplanes, and these hyperplanes together describe the smallest subset of $\mathbb{R}^F$ that both contains x and can be expressed with our data structure. Clearly, this is the first place we should look at to find out K nearest neighbours. If we ignore the last choice, and only consider the previous $h-1$, we describe the second smallest subset of $\mathbb{R}^F$ with the two aforementioned properties. It's important to note that even if we take "wrong turns" after $h-1$ correct binary decision, we will never end up outside of the subset that the first $h-1$ hyperplanes describe.

These few considerations motivate the simple rule for building the optimal KNN given our data structure through a series of local decisions. This rule is expressed by the last 4 meaningful lines of Algorithm 2, where $first$ is the child on the same side of the hyperplane as the query node, and second is the other child.

Please refer to our repository [1] for the full implementation. The module was developed in Cython [1] to optimize the execution time while remaining easily embeddable in existing Python-based models.

[1]The code is available at github.com/francescopuddu/approximate_knn

4. Experiments

We validate our proposed solution with two sets of experiments. In the first, we compare it with a standard implementation of the KNN algorithm (namely the one used in [7]) to assess how the approximation accuracy depends on the specifics of the setting. In the second, we focus on the practical case of a DGCNN architecture trained for the point cloud part segmentation task, evaluating the improvements when the original KNN implementation is replaced with our proposed method. With the experiments, we aim to show the properties of our solution as a generically applicable tool to create latent graphs efficiently, showing the aforementioned downstream task as a significant example where it can unlock new orders of magnitude of input complexity. We implement our algorithm both in vanilla Python using Numpy, and in Cython for faster execution.

Datasets. We perform our experiments on the standard ShapeNetPart benchmark [9], made specifically for the part segmentation task. This dataset contains detailed per-point labeling of 31963 models in 16 shape categories, spanning different types of real-world objects. For Experiment 3 only we resort to randomly generated point cloud instances to span over various input dimensionalities.

4.1. Comparison with Atandard KNN

We analyse this comparison through two metrics, where the comparison data (used as ground truth) come from the exact implementation of KNN.

- $Intersection\ over\ Union$ (IoU) : we evaluate it between the computed neighbourhoods. We aim at maximizing this metric,
- $Average\ Distance\ from\ Centroid$ (ADC) : it considers the average distance between centroids and neighbours. We aim at minimizing this metric, obviously considering that since we compute the ratio to ground truth results, the optimal value is 1.

In all experiments, the final results of both metrics are aggregated by averaging over all data points. The experiments cover three significant variables involved in the creation of latent graphs, namely cardinality, dimensionality and the k hyperparameter. Where it is not the test subject, the value of k is set as in the original paper (40). The default values for cardinality and dimensionality are instead those of the ShapeNetPart pointclouds (2048 points, 3 dimensions).

Experiment 1 : Dependence from Cardinality We applied different KNN strategies to point cloud samples of increasing size to test the dependence of the two metrics on

the number of points, without noticing a statistically significant trend in the scores. The results are shown in Figures 4 and 5.

Experiment 2 : Dependence from K Consistently with the previous experiment, we observe that repeating the graph construction operation as k varies does not result in any substantial trend in the scores for either metric. The results are shown in Figures 6 and 7.

Experiment 3 : Dependence from Dimensionality To test at different orders of magnitude dimensionality, we had to resort to randomly generated point clouds, as sampling from ShapeNetPart would limit us to three-dimensional vectors. The results we obtained show a decrease in performance as the number of dimensions increases. We consider this result a pessimistic estimate in light of the curse of dimensionality problem for distances computed between random high-dimensionality vectors. The results are shown in Figures 8 and 9.

4.2. Extending the DGCNN Architecture

We implement an extended version of the DGCNN architecture using our algorithm for the latent graph creation, and we compare the results with the ones of the original version. Both models trained with the same set of hyperparameters, identical to those in the paper except for batch size which is reduced to 2 to comply with GPU memory limitations, and learning rate which needs to be adjusted given the reduction in batch size. We run grid search on the learning rate and select 0.000316. We run our experiments on a desktop PC with an NVIDIA Geforce GTX 970 4GB graphics card, an AMD Ryzen 7 3800X processor and 16GB of RAM. Any interpretation of the results cannot disregard the fact that while the classical KNN algorithm can be implemented as a vectorial operation and thus computed on parallel hardware, the same does not hold for our algorithm which, given its inherently sequential nature, relies instead on the processor and main memory. In this difference we highlight a further trade-off to discuss in the following sections.

Experiment 4 : Computational Time We compared the execution time to process one batch through the DGCNN architecture. Although the Cython implementation results in an extremely consistent speed-up compared to its NumPy counterpart, the practical advantage of being able to implement the standard algorithm on a GPU proves decisive in its favour as the size of the point cloud increases. In the absence of batch-level parallelisation, our algorithm reports execution times in the same order of magnitude as the standard one, with a relative increase limited to 3ms per batch (around 20%). We run experiment both on a batches and single point clouds, with similar results, and report the data for the point cloud experiments in Figure 12.

Experiment 5 : Peak Memory Consumption We compared the peak memory consumption reached during the computation of a graph as the number of points increases. Although the matter is evidently resolved by considering the different orders of magnitude of the spatial complexities (quadratic in the standard implementation, linear in ours), it is interesting to report a practical consequence of this difference. While comparing the the memory consumption to compute point clouds of the same size of the dataset ones (2^{11} points) results in a ratio between of approximately 57:1, for orders of magnitude greater than 2^{15} the standard implementation's peak is such that it requires excessive hardware resources, i.e. more than 64GB of available memory. For the same number of points, we require less than 20MB. The results are shown in Figures 10 and 11.

Experiment 6 : Performance Gap To assess the influence of neighbourhood approximation on training effectiveness, we compare the performance of two models trained with the same training settings but different latent graph creation strategies. We note that although the model featuring our algorithm suffers a performance gap, this is quantified as moderate and does not affect the learning trend observed in the original conditions. Due to limited availability of computational resources, we were only able to compare the two models over a smaller number of epochs (20), and could not assess the final convergence value. Refer to Figures 13 and 14 for further details.

Results and discussion. Notwithstanding the need for further analysis of the performance at high-dimensions and the convergence of the training process, the results indicate that the proposed algorithm can be considered a competitive alternative to existing methods for computing approximate KNN. Furthermore, the proven performance within LGL architectures is significant in itself, as it allows for extremely more memory-efficient training pipelines.

Future directions of research could be the parallelization of the tree creation process over multiple batches, and of the KNN extraction process over all points. Moreover, adding multiple trees to our data structure could reasonably improve the quality of the extracted KNNs.

5. Conclusions

We analyze 3 prominent papers in the field of Latent Graph Learning, and find a common challenge in the quadratic scaling of their complexity with regard to the size of the dataset. We address this challenge by developing a

novel approximate KNN algorithm based on a hierarchical data structure. We test our algorithm in terms of adherence to the ideal neighborhood, time and memory consumption, and training results. We prove that with regards to memory consumption, we significantly improve over the approaches used in the aforementioned papers, while remaining competitive in terms of training results and time performance.

Acknowledgements

We would like to thank J. Masci for the useful initial directions he gave us in the field of Latent Graph Learning.

References

[1] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011. 4

[2] E. Bernhardsson. Annoy. https://github.com/spotify/annoy, 2015. 3

[3] L. Cosmo, A. Kazi, S.-A. Ahmadi, N. Navab, and M. Bronstein. Latent-graph learning for disease prediction. In *Medical Image Computing and Computer Assisted Intervention – MICCAI 2020*, pages 643–653. Springer International Publishing, 2020. 2, 8

[4] T. Kipf, E. Fetaya, K.-C. Wang, M. Welling, and R. Zemel. Neural relational inference for interacting systems, 2018. 2, 8

[5] W. Li, Y. Zhang, Y. Sun, W. Wang, W. Zhang, and X. Lin. Approximate nearest neighbor search on high dimensional data — experiments, analyses, and improvement (v1.0), 2016. 3

[6] D. A. Suju and H. Jose. Flann: Fast approximate nearest neighbour search algorithm for elucidating human-wildlife conflicts in forest areas, 2017. 3

[7] A. Tao. dgcnn. https://github.com/AnTao97/dgcnn.pytorch, 2022. 4

[8] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon. Dynamic graph cnn for learning on point clouds, 2018. 2, 3, 8

[9] L. Yi, V. G. Kim, D. Ceylan, I.-C. Shen, M. Yan, H. Su, C. Lu, Q. Huang, A. Sheffer, and L. Guibas. A scalable active framework for region annotation in 3d shape collections. *SIGGRAPH Asia*, 2016. 4

Algorithm 1 Creation of the Tree

Require: X
Ensure: T
  $t_0 \leftarrow ()$
  $T \leftarrow \{t_0\}$
  $P \leftarrow random\ permutation\ of\ [1...n]$
  while $\mid Z \mid \leq N$ **do**
    $i \leftarrow sample\ without\ replacement\ from\ P$
    ADD NODE $(i,\ 1)$
  end while
  return T

  function ADD NODE $(p,\ c)$
    ▷ p is the index of point to insert in T
    ▷ c is the index of candidate parent
    if $t_c.\ axis \neq Null$ **then**
      $t_p \leftarrow (x_p,\ Null,\ Null,\ t_c\ \vec{0})$
      $t_c.\ left \leftarrow t_p$
      return
    end if
    if $t_c.\ left = Null \vee t_c.\ right = Null$ **then**
      $t_c.\ axis \leftarrow axis\ of\ segment\ (x_c\ x_p)$
      $t_p \leftarrow (x_p,\ Null,\ Null,\ t_c\ \vec{0})$
      if $t_c.\ axis \cdot x_p > 0$ **then**
        $t_c.\ right \leftarrow t_c.\ left$
        $t_c.\ left \leftarrow t_p$
      else
        $t_c.\ right \leftarrow t_p$
      end if
      return
    end if
    if $t_c.\ axis \cdot x_p > 0$ **then**
      ADD NODE $(p,\ index\ of\ x_c.\ right)$
    else
      ADD NODE $(p,\ index\ of\ x_c.\ left)$
    end if
  end function

Algorithm 2 Creation of the Graph

Require: $T,\ k$
Ensure: $\mathcal{G}$
  $\mathcal{V} \leftarrow \{1\ ,\ ...\ ,\ n\}$
  $\mathcal{E} \leftarrow \{\}$
  for $i \leftarrow 1$ to N **do**
    KNN $(i,\ i)$
  end for
  $\mathcal{G} \leftarrow (\mathcal{V}, \mathcal{E})$
  return $\mathcal{G}$

  function KNN $(p,\ c)$ *
    ▷ p is the index of point to insert in $\mathcal{G}$
    ▷ c is the index of candidate neighbour
    if $t_c.\ axis \cdot x_p\ < 0$ **then**
      $first \leftarrow\ t_c.\ left$
      $second \leftarrow\ t_c.\ right$
    else
      $first \leftarrow\ t_c.\ right$
      $second \leftarrow\ t_c.\ left$
    end if
    if $first \neq Null$ **then** KNN ($first,\ p$)
    if $t_c.\ point \neq Null$ **then** $\mathcal{E} \leftarrow \mathcal{E} + (p, c)$
    if $second \neq Null$ **then** KNN ($second,\ p$)
    if $caller\ \neq\ parent\ node$ **then** KNN ($t_c.\ parent,\ p$)
  end function

* If at any moment the neighbourhood reaches size k, then the procedure returns to the original upstream caller. We omit implementation details for readability.

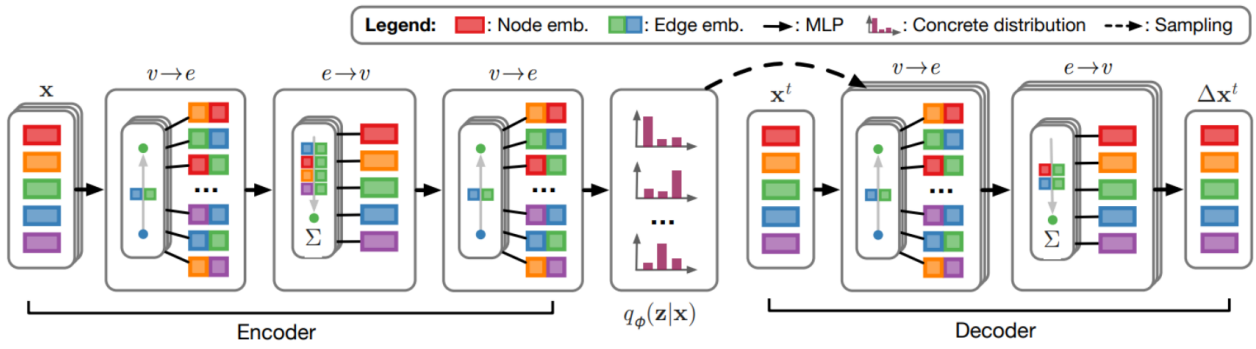

Figure 1: The architecture presented in [4], courtesy of the authors.

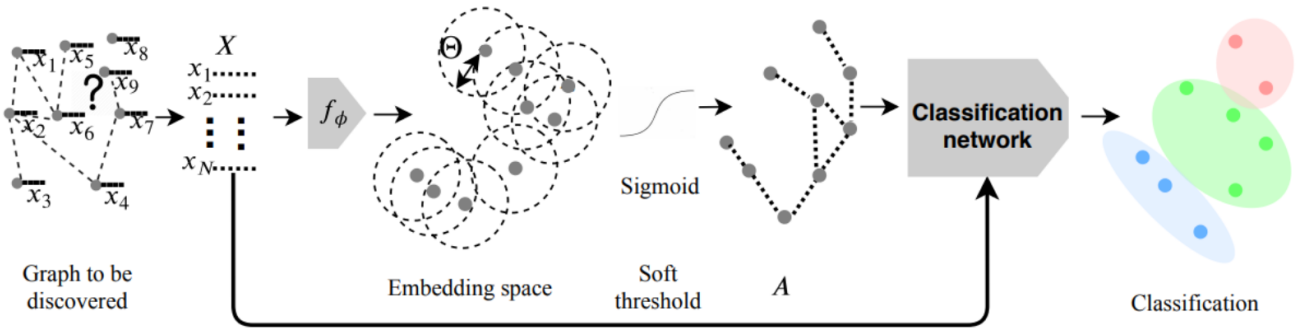

Figure 2: The architecture presented in [3], courtesy of the authors.

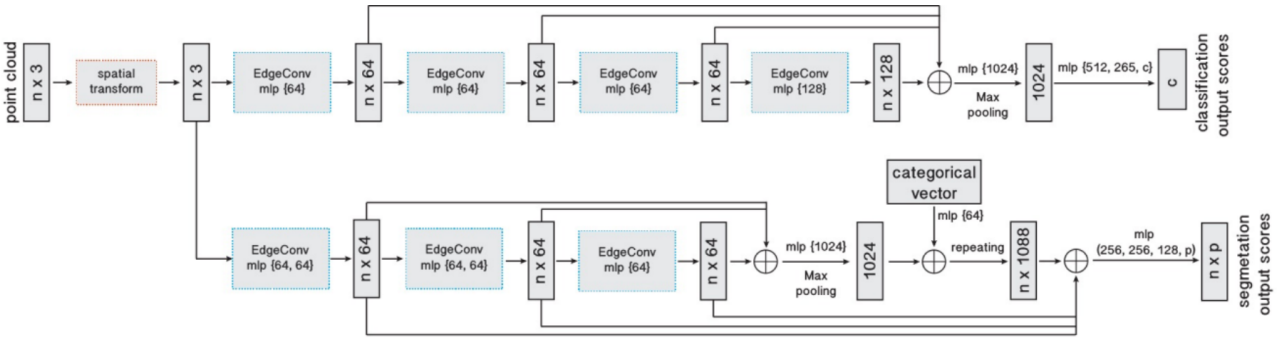

Figure 3: The architecture presented in [8], courtesy of the authors.

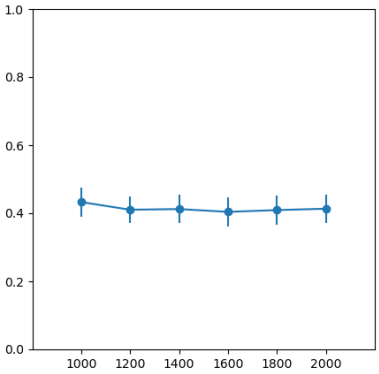

Figure 4: The IoU score (y) against the point cloud size (x).

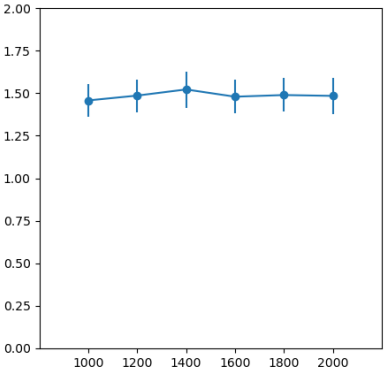

Figure 5: The ADC score (y) against the point cloud size (x).

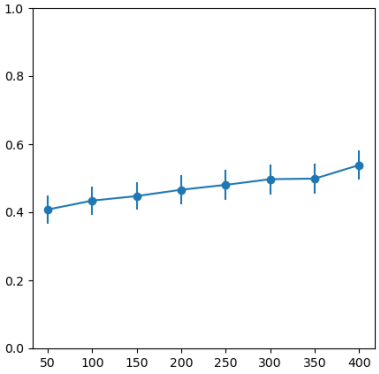

Figure 6: The IoU score (y) against the k hyperparameter (x).

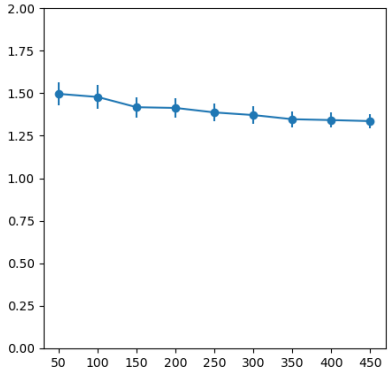

Figure 7: The ADC score (y) against the k hyperparameter (x).

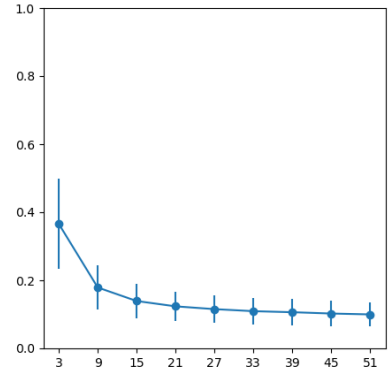

Figure 8: The IoU score (y) against the point cloud dimensionality (x).

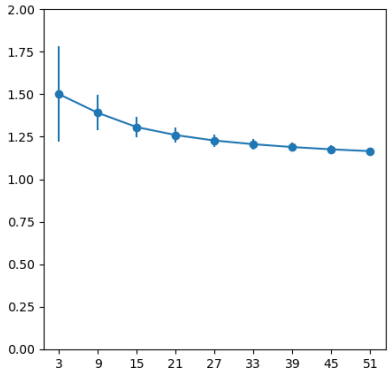

Figure 9: The ADC score (y) against the point cloud dimensionality (x).

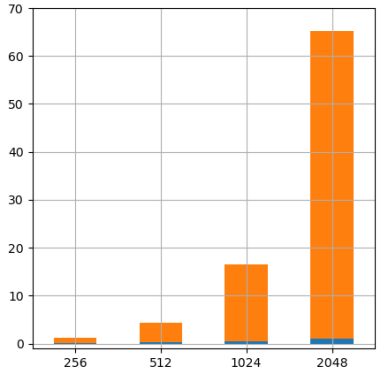

Figure 10: Memory consumption (MB) at different point cloud sizes. Our algorithm (blue) against standard KNN (orange).

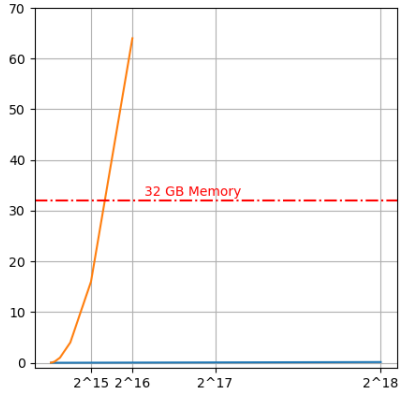

Figure 11: Memory consumption (MB) at different point cloud sizes. Our algorithm (blue) against standard KNN (orange).

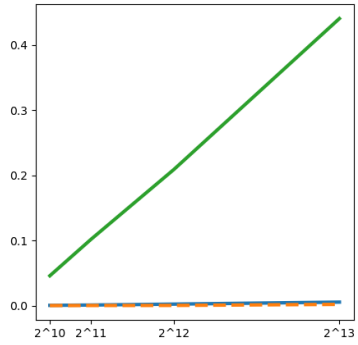

Figure 12: Execution time (seconds) at different point cloud sizes. Our algorithm appears both in the Cython implementation (blue) and the NumPy one (green) against standard KNN (orange).

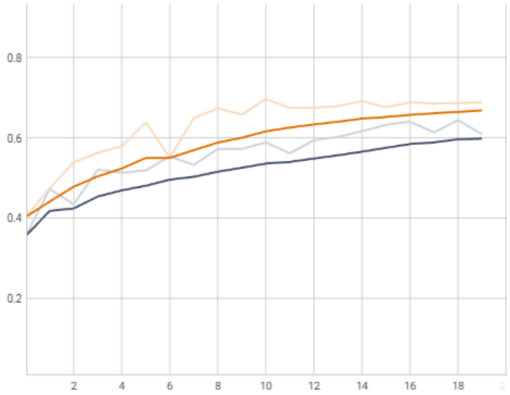

Figure 13: Balanced test accuracy during training. Our algorithm (blue) against standard KNN (orange).

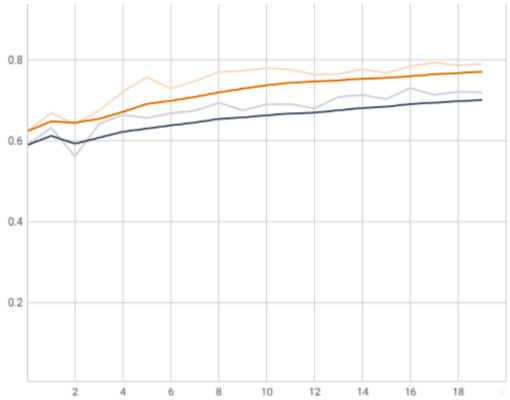

Figure 14: Average test IoU during training. Our algorithm (blue) against standard KNN (orange).

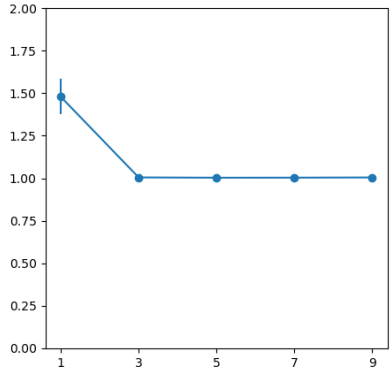

Figure 15: The ADC score (y) against the forest size (x).