Frame Predictions: An Unsupervised Method for Predicting Future LiDAR Scans

David Deng, Ajay Gopi, Wendi Li, Scott McCrae
davezdeng8@berkeley.edu, ajaygopi98@berkeley.edu, wendili@berkeley.edu, mccrae@berkeley.edu
UC Berkeley, Department of Electrical Engineering and Computer Sciences

Abstract

In this work, we propose a model for processing sequential lidar data and generating predictions for lidar data in the future. Using point cloud feature extractors, we construct a feed-forward network which processes a set of recent LiDAR scans and then produces a prediction for a scan $t + \varepsilon$ one time step in the future. We train the network in a self-supervised manner, comparing the generated point cloud with the actual point cloud recorded at $t + \varepsilon$ the next time step in the sequence using the Chamfer distance as our loss. Our network is feed-forward and end-to-end trainable. We achieve an average Chamfer distance of 0.133 on our validation set, surpassing the baseline Chamfer distance of 0.145 between point clouds at t and $t + 1$.

1 Introduction

In the space of perception for autonomous vehicles, much work has gone in to producing pipelines for object detection, semantic segmentation, tracking, and more. Recent works have developed methods for feature extraction on lidar point clouds [Qi et al., 2017] and sensor fusion [Meyer et al., 2019]. Until just a few months ago, researchers were constrained in their methods by the publicly available datasets. For example, KITTI does not feature rich temporal data, so it is hard to train data-hungry algorithms on previously available data. Newly available datasets from companies like Lyft Level 5, Alphabet's Waymo, and NuTonomy have more modern sensors and feature data that is captured in many series of 20 seconds each. So while previous research was mostly constrained to processing individual lidar scans, new data has opened the door to training models which use this sequential data. Since this data is new, not much work in this vein has been published yet.

Methods that incorporate unsupervised learning have the advantage of being able to use large amounts of data without requiring the time-intensive and expensive process of labelling. In that vein, we propose a method for using sequential lidar data to predict what lidar scans will look like for some small period of time in the future. Since we now have access to datasets with long sequences of data capture, we can leverage a self-supervised model of learning. Rather than relying on a generative adversarial model, we can use the vast amounts of data available to us to directly calculate a reconstruction loss on our predicted scans compared to the actual data.

Applications of this task might not be as immediately applicable as object detection networks, but they are still valuable. For instance, predicted data could be fed into a fast detection network with a low confidence threshold to produce 3D region proposals for objects in the immediate future. Or, this network could be included as an auxiliary task designed to encourage useful latent embeddings of lidar data in a larger network designed for object de-

tection or semantic segmentation.

2 Relate Works

2.1 Feature Extraction

Some of the most widely used methods for processing point cloud data in recent years have been the PointNet and PointNet++ architectures from Qi et al. The former produces a global feature vector for the entire point cloud, which can then be used for a multitude of tasks. It was improved upon in the latter work with the introduction of pointwise features, addressing a major drawback of the prior work by including a notion of local features in addition to global features. These networks have an advantage over approaches like VoxelNet [Zhou and Tuzel, 2018] or MV3D [Chen et al., 2017], which voxelize the 3D space containing the point cloud.

Some recent works have investigated fusing features from RGB camera sensors and LiDAR sensors. Methods from [Liang et al., 2018] and [Meyer et al., 2019] extract image features in addition to point cloud features, and then fuse them throughout the network to form the features which are regressed for object detection.

In addition to directly extracting features from the point cloud, papers have experimented with different representations of the data. As mentioned above, papers like VoxelNet will voxelize the point cloud into a 3D grid. Others will process the data in its Bird's Eye View representation, essentially projecting the 3D data onto the ground plane to form a 2D representation. Yet others will view the LiDAR data in the Range View, representing the data as a depth map with one-to-one and well-defined correspondences between pixels and real-world locations.

2.2 Sequential Processing

Although there is a rich body of work that processes sequential data in domains such as video, audio, or language, there is not much work which processes sequential LiDAR data as of yet. In 2D video, there has been much work on the optical flow problem, and recently FlowNet3D [Liu et al., 2019] has extended this approach to LiDAR data.

Zhou et al. have proposed a pipeline for unsupervised learning of depth in autonomous driving, which is a similar problem. We are inspired by their approach to use self-supervision via sequential data, although we use take fundamentally different approach to our loss function.

Ma et al. propose a network which learns to complete sparse depth images in KITTI using a sequence of RGB images and sparse depth images. They use sequential data to perform self-supervision, but they break that down into a couple steps. First, they directly supervise the dense completion at points where depth was measured with LiDAR. They also estimate the change in pose between each frame in their data, then use this to calculate a photometric loss on pixels without depth supervision.

Our decision to use a feed-forward model is informed by work from Bai et al. which showed that a generic feed-forward network can outperform recurrent networks on a variety of tasks (although none of them had to do with 2D or 3D computer vision).

3 System Architecture

3.1 Data Preparation

We use the nuScenes dataset for our training and validation data. In order to prepare our data, we take two main steps. We start with the observation that many of the points in each point cloud are concentrated around the ego-vehicle and the immediately surrounding area, which tends to be the surface of the road. For example, a point cloud with about 34,000 points has about 21,000 points on the ego-vehicle and the ground around it. Faced with the need to subsample the point cloud for memory reasons, this inner set of points is safe to dis-

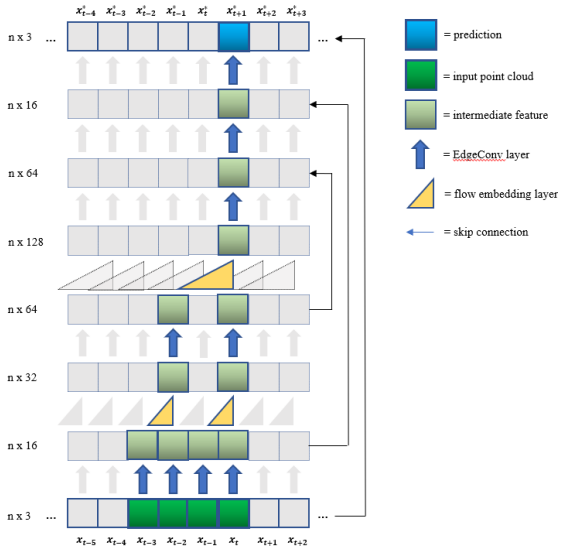


Figure 1: An overview of the proposed sequential processing pipeline.

card. These points tend not to contain useful information, and having our network predict 5,000 points on the hood of the ego-vehicle is unimportant and uninformative. Furthermore, we note that the points farthest from the vehicle tend to be on the faces of buildings or very distant objects. In the short-term, very distant objects remain distant, and it is more important to consider objects closer to the ego-vehicle. So, we discard the furthest points.

First, we sort the point cloud based on distance from the origin. This allows us to select an annular region of 12,000 points that are neither very close nor very far from the ego-vehicle. Second, we randomly subsample this resulting region by a factor of 2. This results in a point cloud with 6,000 points.

It is worth noting that this random subsampling of our data acts as a minor form of data augmentation. Although we are running this process to get a smaller point cloud, the stochastic nature of the process means that each epoch will contain slightly different training data. This could help the model generalize better, but comes at the cost of possibly messing with the Chamfer distance (see Section 3.5.3 for more detail).

For our network, we use both the training and validation sets to train our network, and we validate on the test set. We do this because we do not require annotations, and we are not submitting to a competition. This allows us to train our model on a greater variety of scenes.

3.2 Feature Extraction Layers

We use a combination of EdgeConv [Wang et al., 2019] and Flow Embedding [Liu et al., 2019] layers, from DGCNN and FlowNet3D respectively, to perform feature extraction and to generate our predicted point cloud. These two modules are a good fit for our specific use case for a couple reasons. First, because we want to go from point cloud to point cloud, it helps that these layers maintain the structure of the data, i.e. as a point set with associated features. If we were to use PointNet, we would instead have a global feature vector, which we would have to use to regress a point set from. As it turns out, DGCNN is a more expressive framework than PointNet, which is only able to express a subset of functions expressible by DGCNN. Second, these layers avoid voxelizing the 3D space of our point cloud, which allows us not to lose any resolution or fine detail in the input data. We explain how to use these layers to predict a point cloud in the next section.

We use the implementation of EdgeConv provided by the original authors, and we implement the Flow Embedding ourselves using the EdgeConv framework.

3.3 Temporal Processing

Our approach for processing sequential data is to use a fully feed-forward, rather than recurrent, model. Several issues crop up when processing different scans of LiDAR at once, such as varying numbers of points in each point cloud as well as the unordered nature of the data. Our solutions to these are discussed below.

Due to the nature of LiDAR sensors, not every recorded point cloud has the same number of points. A car might be going over a hill, for instance, meaning at some times the front of the car is facing the air, and at others it faces the ground. This issue is conveniently solved by subsampling our data to have the same number of points each time. Before settling on subsampling, we had planned on adopting the convention that our predicted LiDAR frame yt+1 will have the same number of frames as the most recent input, yt. Note how the Chamfer distance loss function does not rely on both point clouds having the same amount of points.

As mentioned, we opt for a feed-forward model. Our network accepts four point clouds as input (Figure 1), and uses these four to generate the next point cloud. We note that our network does not necessarily need more data than the past few LiDAR frames, since the world will not drastically change between what has happened in the past 0.2 seconds and what happens in the next 0.05 seconds. At slow speeds, objects will not move much between frames, and at high speeds, trajectories do not change much between frames. That said, more frames might help implicitly predict the trajectory of objects.

We are able to generate one point cloud from four inputs by leveraging the behavior of the Flow Embedding layer. Although it takes in two point clouds, it only outputs features for one. Functionally, this allows us to reduce our working data by one point cloud, and this process is repeated until we are left with only one point cloud.

We are able to map our data into a learned latent representation by using EdgeConv layers along with Flow Embedding. By specifying the input and output dimensions for EdgeConv, we are able to map our data from the original 3 xyz features into a higher-dimensional representation, and then map back down to 3 xyz dimensions. Interestingly, each Flow Embedding layer can be extended to process an arbitrary number of features, not just 3. So, in our implementation, each layer models a flow between point clouds within their latent representation, rather than just in 3D.

Although this dimensional mapping erases explicit semantic meaning of the features along the way, we enforce the fact that the resulting final 3 dimensions are indeed xyz coordinates by minimizing the Chamfer distance between the prediction and the ground truth. Because the ground truth features are xyz coordinates, minimizing loss encourages our prediction to have these same features. The loss is discussed more in Section 3.5.

3.4 Point Cloud Generation

As mentioned in Section 3.2, we predict a lidar frame $y_{t\hat{+}1}$ such that the cardinality of $|y_{t\hat{+}1}| = |\hat{y}_t|$. To this effect, rather than generate a point cloud from noise input as one might in a GAN-based model, we rely on the EdgeConv operation to perturb the points in our point clouds. This process is made slightly easier by the fact that we subsample our point clouds to have identical sizes.

In addition, to improve the backpropagation and encourage the network to regress back to xyz coordinates, we add several skip, or residual connections across the network at different levels of feature abstraction. Specifically, we add the input to the output of the final layer, the output of the first layer to the features before the last layer, and so on. We implement these connections using addition because our features are all pointwise and we maintain dimensionality across the skip connections. Refer to Figure 1. for more details.

3.5 Loss

3.5.1 An Ideal Scenario

The following is a description of what we ideally would have used for our network if we were not memory constrained..

Our network uses a combined loss function. One component is the Chamfer distance between the predicted LiDAR frame and the ground truth, and

the other is perceptual loss measured by the l^2-norm distance between PointRCNN features extracted from the predicted frame and the ground truth. Formally, we have:

$$L_{total} = \lambda_C L_{Chamfer} + \lambda_P L_{Perceptive} \quad (1)$$

$$L_{Chamfer} = \frac{1}{|y|} \sum_{a \in Y} \min_{b \in \hat{y}} \|a - b\|^2 + \frac{1}{|\hat{y}|} \sum_{b \in \hat{y}} \min_{a \in y} \|a - b\|^2 \quad (2)$$

$$L_{Perceptive} = \|h(y) - h(\hat{y})\|^2 \quad (3)$$

where $h(\cdot)$ is the feature extractor from PointRCNN. PointRCNN has many stages of feature extraction and representation, and we choose to use the output of their so-called "Point Cloud Encoder", which combines semantic features as well as local spatial features. This gives us a global feature vector, which is amenable to analysis for perceptual loss, since it avoids possible correspondence issues when using pointwise feature comparison.

Meanwhile, Chamfer distance measures alignment between point clouds. As you can see from the provided formula, it measures distance from each point in one point cloud to the nearest point in the second point cloud. On its own, this operation is not symmetric, i.e. $Chamfer(a, b) \neq Chamfer(b, a)$. So, we take the average of the Chamfer distance calculated bi-directionally. Minimizing this Chamfer loss will drive the predicted point cloud closer to the ground truth, with a global minimum reached when the two point clouds are identical.

The problem of determining a sort of reconstruction error between our predicted frame and the ground truth is difficult. Unlike 2D images, we cannot use metrics like MSE, since our data is unordered and does not necessarily have one-to-one point correspondences. We also cannot use an l_p loss metric like we do for perceptual loss, since the vectors representing our point clouds suffer from the aforementioned correspondence issue.

3.5.2 Compromise

Although we would have liked to use a combined loss function for our network, we were constrained by GPU memory. In reality, the perceptual loss function is impractical for our use case. Because it requires taking a feature from PointRCNN, we would have to run PointRCNN on our predicted and ground-truth point clouds to generate these features. This requires loading the entire model into GPU memory, in essence requiring us to store and run two large models concurrently. It is both memory- and time-intensive. Faced with this reality, we have decided not to incorporate the perceptual loss from PointRCNN.

3.5.3 Viability of Chamfer Distance

In class, a student made a good point that Chamfer distance relies on our two point clouds having relatively reasonable alignment in order to serve as a good distance metric. PointRCNN does not have such a requirement. In that sense, perceptual loss is helpful – it can more easily distinguish a jumbled mess from a well-formed point cloud with semantic meaning. So, using only Chamfer distance might mean that our network has less informative gradients to work with when training begins. (That said, we empirically note that Chamfer distance provides a high value for highly dissimilar point clouds, as discussed in our presentation.)

In practice, we notice very high Chamfer distances at the beginning of model training (order of 1000), although they rapidly approach more reasonable values (order of 1). At the outset, our network has no inbuilt priors to encourage semantically meaningful or otherwise well-formed point clouds, so this high Chamfer distance is encouraging. This implies that Adi's concern, while valid, is not crippling the network. It is also worth noting that the high sample rate of LiDAR sweeps in the nuScenes dataset (20 Hz) means that each subsequent point cloud is pretty similar to its predecessor, so Chamfer distance is a reasonable metric in

our use case.

4 Experiments

We train our model on the nuScenes dataset using the Adam optimizer with cosine annealing, processing 8 frames of LiDAR at a time. Our model is trained end to end on an Nvidia RTX 2080 Ti.

We pretrain our model to learn to perform an identity mapping from the point cloud sequence ending at time t to the point cloud at time t. Giving our network good weights to start with allows it to focus on small-scale fixes, rather than needing to learn how to represent meaningful point clouds from scratch.

Since there is no real prior state of the art for our specific problem, we will instead try to provide insights into the performance of our model by comparing it to a naive baseline of simply using the point cloud at time t as the prediction. Below are our results:

	Naive baseline	Ours
Chamfer Distance (m)	0.145	**0.133**

The table shows the average Chamfer distance in the validation set between the ground truth and the naive baseline, and the ground truth and our method. We calculate the error for predictions one time step in the future. As you can see, our model outperforms the naive method, despite the naive method already being a relatively strong baseline. Using the baseline over our model would lead to a 9.02% increase in Chamfer distance.

5 Discussion

Overall, we are happy with the model we were able to construct. That said, it is frustrating that we have had to subsample the point clouds so aggressively (a factor of 5 to 6 on average) and that we were not able to train using the perceptual loss. We feel as though both of these compromises potentially hold back the performance of our model, but we are currently constrained by hardware. Un-doing these compromises would be very interesting future work.

One promising avenue for future work would be incorporating some sort of sensor fusion. Image features from RGB cameras could help the network better understand the scene, and data from radar sensors might help predict objects' trajectories. Both these modalities could be incorporated into the self-supervised architecture. Including object detection, tracking, or semantic segmentation results may also help inform the model how the scene will behave over time. That is, knowledge about the motion of objects might allow the network to more intelligently move groups of points in a cohesive manner.

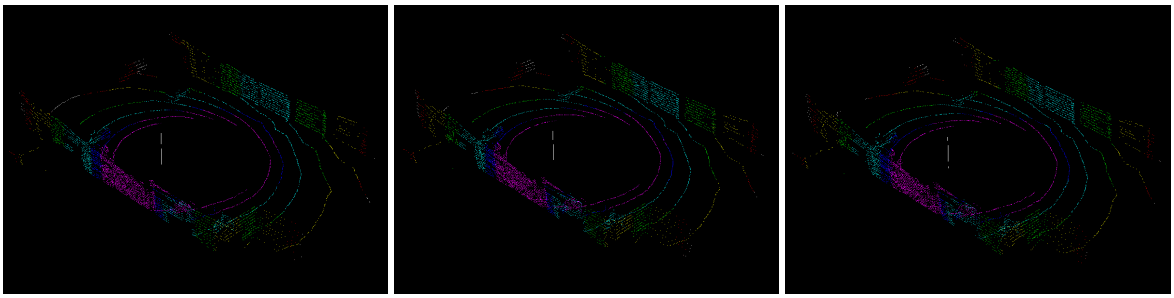
We think this work could be used as a part of a region proposal network for 3D object detection using point clouds. It would be interesting to implement this and see how performance is affected.

As for more concrete future work on this project, we are interested in experimenting with the use and placement of skip connections. We hypothesize that adding a skip connection from xyz input coordinates to layers after Flow Embedding is useful because the Flow Embedding can calculate a displacement without necessarily remembering the original coordinates. Thus, giving future layers access to both displacement and original location is useful; we would like to try to quantify exactly how informative these skip connections are.

6 Conclusion

6.1 Summary

In this paper, we propose a model for predicting future LiDAR frames given a set of previously seen frames. As far as we know, this problem has no solutions in the literature. We opt to use EdgeConv and Flow Embedding operations to map our input



(a) Point cloud at time t (b) Point cloud at time $t+1$ (c) Prediction for time $t+1$

Figure 2: Point cloud at time t, $t+1$, and prediction for $t+1$ from left to right. Notice subtle similarities between $t+1$ and the prediction that are not present between t and $t+1$.

data into a learned feature space. We use Chamfer distance between predictions and ground truth in a self-supervised manner, taking advantage of the rich temporal data provided by the nuScenes dataset.

6.2 Student Contributions

Throughout the course of this project, we held weekly meetings to touch base and discuss the trajectory of our work. To this end, many design choices were reached through group discussion and consensus. In terms of specific implementation, David spearheaded the network training process. Wendi contributed to the data loading and sampling pipeline. Ajay contributed to implementations of our loss functions. Scott wrote the paper and presentation, and helped with data loading and loss function construction. We all pitched in to help each other fix bugs.

References

Shaojie Bai, J Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018.

Xiaozhi Chen, Huimin Ma, Ji Wan, Bo Li, and Tian Xia. Multi-view 3d object detection network for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1907–1915, 2017.

Ming Liang, Bin Yang, Shenlong Wang, and Raquel Urtasun. Deep continuous fusion for multi-sensor 3d object detection. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 641–656, 2018.

Xingyu Liu, Charles R Qi, and Leonidas J Guibas. Flownet3d: Learning scene flow in 3d point clouds. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 529–537, 2019.

Fangchang Ma, Guilherme Venturelli Cavalheiro, and Sertac Karaman. Self-supervised sparse-to-dense: Self-supervised depth completion from lidar and monocular camera. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 3288–3295. IEEE, 2019.

Gregory P Meyer, Jake Charland, Darshan Hegde, Ankit Laddha, and Carlos Vallespi-Gonzalez. Sensor fusion for joint 3d object detection and semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2019.

Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for

3d classification and segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 652–660, 2017.

Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E Sarma, Michael M Bronstein, and Justin M Solomon. Dynamic graph cnn for learning on point clouds. *ACM Transactions on Graphics (TOG)*, 38(5):146, 2019.

Tinghui Zhou, Matthew Brown, Noah Snavely, and David G Lowe. Unsupervised learning of depth and ego-motion from video. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1851–1858, 2017.

Yin Zhou and Oncel Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4490–4499, 2018.