PointRNN: Point Recurrent Neural Network for Moving Point Cloud Processing

Hehe Fan, Yi Yang
University of Technology Sydney

Abstract

In this paper, we introduce a Point Recurrent Neural Network (PointRNN) for moving point cloud processing. At each time step, PointRNN takes point coordinates \( P \in \mathbb{R}^{n \times 3} \) and point features \( X \in \mathbb{R}^{n \times d} \) as input (\( n \) and \( d \) denote the number of points and the number of feature channels, respectively). The state of PointRNN is composed of point coordinates \( P \) and point states \( S \in \mathbb{R}^{n \times d'} \) (\( d' \) denotes the number of state channels). Similarly, the output of PointRNN is composed of \( P \) and new point features \( Y \in \mathbb{R}^{n \times d''} \) (\( d'' \) denotes the number of new feature channels). Since point clouds are orderless, point features and states from two time steps cannot be directly operated. Therefore, a point-based spatiotemporally-local correlation is adopted to aggregate point features and states according to point coordinates. We further propose two variants of PointRNN, i.e., Point Gated Recurrent Unit (PointGRU) and Point Long Short-Term Memory (PointLSTM). We apply PointRNN, PointGRU and PointLSTM to moving point cloud prediction, which aims to predict the future trajectories of points in a set given their history movements. Experimental results show that PointRNN, PointGRU and PointLSTM are able to produce correct predictions on both synthetic and real-world datasets, demonstrating their ability to model point cloud sequences. The code has been released at https://github.com/hehefan/PointRNN.

1. Introduction

Most modern robot and self-driving car platforms rely on 3D point clouds for geometry perception. In contrast to RGB images, point clouds can provide accurate displacement measurements, which are generally unaffected by lighting conditions. Therefore, point cloud is attracting more and more attention in the community. However, most of the existing works focus on static point cloud analysis, e.g., classification, segmentation and detection [22, 23, 15, 20]. Few works study dynamic point clouds. Intelligent systems need the ability to understand not only the static scenes around them but also the dynamic changes in the environment. In this paper, we propose a Point Recurrent Neural Network (PointRNN) and its two variants for moving point cloud processing.

In the general setting, a point cloud is a set of points in 3D space. Usually, the point cloud is represented by the three coordinates \( P \in \mathbb{R}^{n \times 3} \) of points and their features \( X \in \mathbb{R}^{n \times d} \) (if features are provided), where \( n \) and \( d \) denote the number of points and feature channels, respectively. Essentially, point clouds are unordered sets and invariant to permutations of their points. For example, the set \( \{(1, 1, 1), (2, 2, 2), (3, 3, 3)\} \) and \( \{(3, 3, 3), (1, 1, 1), (2, 2, 2)\} \) represent the same point cloud. This irregular format data structure considerably increases the challenge to reason point clouds, making many existing achievements of deep neural networks on image and video fail to directly process point clouds.

The recurrent neural network (RNN) and its variants, e.g., Long Short-Term Memory (LSTM) [10] and Gated Recurrent Unit (GRU) [5], are well-suited to processing time series data. Generally, the (vanilla) RNN looks at a vector \( x_t \) at the time step \( t \), updates its internal state (memory) \( s_{t-1} \) to \( s_t \), and outputs a new vector \( y_t \). The behavior of RNN can be formulated as follows,

\[
y_t, s_t = \text{RNN}(x_t, s_{t-1}; \theta),
\]

where \( \theta \) denotes the parameters of RNN. Since moving point cloud is a kind of temporal sequence, we can exploit RNN to process it.

However, the conventional RNN has two severe limitations on processing point cloud sequences. On one hand,
RNN learns from one-dimensional vectors, in which the input, state and output are highly compact. It is difficult for a vector to represent an entire point cloud. Although the \( P \) and \( X \) can be flatten to one-dimensional vectors, such operation heavily damages the data structure and increases the challenge for neural networks to understand point clouds. Moreover, this operation is not independent of point permutations. If we use the global feature to represent a point cloud, the local structure will be lost. To overcome this problem, PointRNN directly takes \((P, X)\) as input. Similarly, the one-dimensional state \( s \) and output \( y \) in RNN are extended to two-dimensional \( S \in \mathbb{R}^{n \times d'} \) and \( Y \in \mathbb{R}^{n \times d''} \) in PointRNN, in which each row corresponds to a point. Besides, because \( S \) and \( Y \) depends on point coordinates, \( P \) is added into the state and output of PointRNN. The behavior of PointRNN can be formulated as follows,

\[
(P_t, Y_t), (P_t, S_t) = \text{PointRNN}((P_t, X_t), (P_{t-1}, S_{t-1}); \theta).
\]

We illustrate an RNN and a PointRNN unit in Figure 1.

One the other hand, RNN aggregates the previous state \( s_{t-1} \) and the current input \( x_t \) based on a concatenation operation (Figure 2(a)). However, because point clouds are unordered, concatenation can not be directly applied to point clouds. To solve this problem, we adopt a spatiotemporally-local correlation to aggregate \( X_t \) and \( S_{t-1} \) according to point coordinates (Figure 2(b)). Specifically, for each point in \( P_t \), PointRNN first searches its neighbors in \( P_{t-1} \). Second, for each neighbor, the feature of the query point, the state of the neighbor and the displacement from the neighbor to the query point are concatenated, which are then processed by a shared fully-connected (FC) layer. At last, the processed concatenations are reduced to a single representation by pooling.

The PointRNN provides a basic component for point cloud sequence processing. Because it may encounter the same exploding and vanishing gradient problems as RNN, we further propose two variants for PointRNN, i.e., Point Gated Recurrent Unit (PointGRU) and Point Long Short-Term Memory (PointLSTM), by combining PointRNN with GRU and LSTM, respectively. We apply PointRNN, PointGRU and PointLSTM to moving point cloud prediction. Given the history movements of a point cloud, the goal of this task is to predict the future trajectories of its points. Predicting how point clouds move in future can help robots and self-driving cars to plan their actions and make decisions. Moreover, moving point cloud prediction has an innate advantage that it does not require human-annotated supervision. Based on PointRNN, PointGRU and PointLSTM, we build up sequence-to-sequence (seq2seq) [28] models for moving point cloud prediction. Experimental results on a synthetic moving MNIST point cloud dataset and two large-scale autonomous driving datasets, i.e., Argoverse [3] and nuScenes [2], show that PointRNN, PointGRU and PointLSTM produce correct predictions, confirming their ability to process moving point clouds.

2. Related Work

**Static point cloud understanding.** The dawn of point cloud has boosted a number of applications, such as object classification, object part segmentation, scene semantic segmentation [22, 23, 12, 14, 15, 30, 8, 27, 18, 33, 31, 29], reconstruction [6, 16, 34] and object detection [4, 35, 21, 24, 13, 20] in 3D space. Most recently works aim to consume point sets without transforming coordinates to regular 3D voxel grids. There exists two main challenges for static point cloud processing. First, a point cloud is in essence a set of unordered points and invariant to permutations of its points, which necessitates certain symmetrizations in computation. Second, different with images, point cloud data is irregular. Convolutions that capture local structures in images can not be directly applied to such data format. Different with these existing works, we focus on a new challenge, i.e., to model dynamics in point cloud sequences.

**RNN variants for spatiotemporal modeling.** Because the conventional RNN, GRU and LSTM learn from vectors where the representation is highly compact and the spatial structure is heavily damaged, a number of variants are proposed to model spatiotemporal sequences. For example, convolutional LSTM (ConvLSTM) [25] modifies LSTM by taking three-dimensional tensors as input and replacing FC by convolution to capture spatial local structures. Based on ConvLSTM, spatiotemporal LSTM (ST-LSTM) [32] extracts and memorizes spatial and temporal representations simultaneously by adding a spatiotemporal memory. Cubic LSTM (CubicLSTM) [7] extends ConvLSTM by splitting states into temporal states and spatial states, which are respectively generated by independent convolutions. However, these methods focus on 2D videos and can not be directly applied to 3D point cloud sequences.

3. PointRNN

In this section, we first review the standard (vanilla) RNN and then describe the proposed PointRNN in detail. The RNN is a class of deep neural networks which uses its state \( s \) to process sequences of inputs. This allows it to exhibit temporal dynamic behavior. The RNN relies on a concatenation operation to aggregate the past and the current, which is referred to as the \( \text{rnn} \) function in this paper,

\[
s_t = \text{rnn}(x_t, s_{t-1}; W, b) = W \cdot [x_t, s_{t-1}] + b, \tag{1}
\]

where \( \cdot \) denotes matrix multiplication and \([\cdot, \cdot]\) denotes concatenation. The \( W \in \mathbb{R}^{d' \times (d+d')} \) and \( b \in \mathbb{R}^{d} \) are the parameters of RNN to be learned. This operation can be implemented by fully-connected (FC) layer in deep neural
networks (shown in Figure 2(a)). Usually, RNN uses its state $s_t$ as output, i.e., $y_t = s_t$.

The conventional RNN learns from one-dimensional vectors, limiting its application to process point cloud sequences. To keep the spatial structure, we propose to directly take the coordinates $P_t \in \mathbb{R}^{n \times d}$ and features $X_t \in \mathbb{R}^{n \times d}$ as input. Similarly, the state $s_t$ and the output $y_t$ in RNN are extended to $S_t \in \mathbb{R}^{n \times (d + d')}$ and $Y_t \in \mathbb{R}^{n \times (d + d')}$ in PointRNN. The update for the point states of PointRNN at the $t\text{-th}$ time step is formulated in Eq. (2), where $W \in \mathbb{R}^{(d + d') \times (d + d' + 3)}$, $b \in \mathbb{R}^{d'}$ and $\mathcal{N}(\cdot)$ is neighborhood query.

We refer to this point-based spatiotemporally-local correlation as point-rnn. Similar to RNN, by default, PointRNN uses $S_t$ as output, i.e., $Y_t = S_t$.

The goal of the point-rnn function is to aggregate the past and the current according to point coordinates (shown in Figure 2 (b)). Given $(P_t, X_t)$ and $(P_{t-1}, S_{t-1})$, point-rnn merges $X_t$ and $S_{t-1}$ according to $P_t$ and $P_{t-1}$. Specifically, for each point in $P_t$, taking the $i\text{-th}$ point $P_{t}^i$ for instance, point-rnn first finds its neighbors in $P_{t-1}$. The neighbors potentially share the same geometry and motion information about the query point $P_{t-1}^i$. Second, for each neighbor, taking the neighbor $P_{t-1}^j$ for instance, the feature $X_{t}^j$ of the query point, the state $S_{t-1}^j$ of the neighbor, and the displacement $P_{t}^i - P_{t-1}^j$ from the neighbor to the query point are concatenated, which are subsequently processed by a shared FC layer. Third, the processed concatenations are pooled to a single representation. The output of point-rnn, i.e., $S_t$, contains the past and the current information of each point in $P_t$.

We adopt two methods for neighborhood query. The first one directly finds $k$ nearest neighbors for the query point ($k$-NN). The second one works by first finding all points that are within a radius to the query point and then sampling $k$ neighbors from the points (ball query [23]). Since a point cloud is a set of orderless points, point states or features of two time steps at the same row may correspond to different points. Without point coordinates, an independent $S_t$ or $Y_t$ is meaningless. Therefore, the point coordinates $P_t$ is added into the state and output of PointRNN.

The PointRNN provides a prototype that uses RNN to process point cloud sequence. Each component in PointRNN is necessary. However, we can design more effective spatiotemporally-local correlation methods to replace point-rnn, which can be further studied in the future.

### 4. PointGRU and PointLSTM

Because PointRNN inherits RNN, it may encounter the same exploding and vanishing gradient problems as RNN. Therefore, we propose two variants, which combine PointRNN with GRU and LSTM, respectively, to overcome these problems. We replace the concatenation operations in GRU with the spatiotemporally-local correlation operations, i.e., point-rnn, forming the PointGRU unit. The up-
dates for PointGRU at the t-th time step are formulated as follows,
\[
Z_t = \sigma(\text{point-rnn}((P_t, X_t), (P_{t-1}, S_{t-1}); W_z, b_z)),
\]
\[
R_t = \sigma(\text{point-rnn}((P_t, X_t), (P_{t-1}, S_{t-1}); W_r, b_r)),
\]
\[
\hat{S}_{t-1} = \text{point-rnn}((P_t, \text{None}), (P_{t-1}, S_{t-1}); W_s, b_s),
\]
\[
S_t = Z_t \odot S_{t-1} + (1 - Z_t) \odot \hat{S}_t,
\]
where \(Z_t\) is the update gate and \(R_t\) is the reset gate. The \(\sigma(\cdot)\) denotes the sigmoid function and \(\odot\) denotes the Hadamard product. Similar to PointRNN, the input of PointGRU is \((P_t, X_t)\), the state is \((P_t, S_t)\) and by default, the output is \((P_t, \hat{S}_t)\). Note that, besides the point-rnn function, another difference between GRU and PointGRU is that, PointGRU has an additional step \(\hat{S}_{t-1}\). The goal of this step is to weight and permute \(S_{t-1}\) according to the current input points \(P_t\). Only after this step can we perform Hadamard product between the previous state \(S_{t-1}\) and the current reset gate \(R_t\).

Similarly, the concatenation operations in LSTM are replaced with the point-rnn functions, forming the PointLSTM unit. The updates for PointLSTM at the t-th time step are formulated as follows,
\[
I_t = \sigma(\text{point-rnn}((P_t, X_t), (P_{t-1}, H_{t-1}); W_i, b_i)),
\]
\[
F_t = \sigma(\text{point-rnn}((P_t, X_t), (P_{t-1}, H_{t-1}); W_f, b_f)),
\]
\[
O_t = \sigma(\text{point-rnn}((P_t, X_t), (P_{t-1}, H_{t-1}); W_o, b_o)),
\]
\[
\hat{C}_{t-1} = \text{point-rnn}((P_t, \text{None}), (P_{t-1}, C_{t-1}); W_c, b_c),
\]
\[
C_t = F_t \odot \hat{C}_{t-1} + I_t \odot \tanh(C_t),
\]
\[
H_t = O_t \odot \tanh(C_t),
\]
where \(I_t\) is the input gate, \(F_t\) is the forget gate, \(O_t\) is the output gate, \(C_t\) is the cell state and \(H_t\) is the hidden state. The input of PointLSTM is \((P_t, X_t)\), the state is \((P_t, H_t, C_t)\) and by default, the output is \((P_t, \hat{S}_t)\). Like from GRU to PointGRU, compared with LSTM, PointLSTM has an additional step \(\hat{C}_{t-1}\), which weights and permutes \(C_{t-1}\) according to the current input points \(P_t\).

5. Moving Point Cloud Prediction

Moving point clouds provide a large amount of geometric information in scenes as well as profound dynamic changes in motions. Understanding scenes and imagining motions in 3D space are fundamental abilities for robot and self-driving car platforms. It is indisputable that an intelligent system, which is able to predict future trajectories of points in a cloud, will have these abilities. In this paper, we apply PointRNN, PointGRU and PointLSTM to moving point cloud prediction. Because this task does not require external human-annotated supervisions, models can be trained in an unsupervised manner.

5.1. Architecture

We design a basic model and an advanced model based on the seq2seq framework. The basic model (shown in Figure 3 (a)) is composed two parts, i.e., one part for encoding the given point cloud sequence and one part for predicting. Specifically, an encoding PointRNN, PointGRU or PointLSTM unit watches the input point clouds one by one. After watching the last input \(P_t\), its state is used to initialize the state of a predicting PointRNN, PointGRU or PointLSTM unit. The predicting unit then takes \(P_t\) as input and begins to make predictions. Rather than directly generating future point coordinates, the proposed models predict displacements \(\Delta P_t\) that will happen between the current step and the next step, which can be seen as 3D scene flow [17, 9].

We can stack multiple PointRNN, PointGRU or PointLSTM units to build up a multi-layer structure for hierarchical prediction. However, because all points are processed in each layer, a major problem of this structure is that it is computationally intensive, especially for high-resolution point sets. To alleviate this problem, we propose an advanced model (shown in Figure 3 (b)). This model borrows two components from PointNet++ [23], i.e., 1) a sampling operation and a grouping operation for down-sampling points and their features and 2) a feature propagation layer for up-sampling the representations associated with the intermediate points to the original points. By this down-up-sampling structure, the advanced model can take advantage of hierarchical learning, while reduce the points to be processed in each layer.

5.2. Training

There are two strategies to train recurrent neural networks. The first one uses ground truths as inputs during predicting, which is known as teacher-forcing training. The second one works using predictions generated by the network as inputs (Figure 3 (a)), which is referred to as free-running training. When using the teacher-forcing training, we find that models quickly get stuck in a bad local optima, in which \(\Delta P_t\) tends to be 0 for all inputs. Therefore, we adopt the free-running training strategy.

Since point clouds are unordered, point-to-point loss functions can not directly apply to compute the difference between the prediction and ground truth. Loss functions should be invariant to the order of input points. In this paper, we adopt Chamfer Distance (CD) and Earth Movers Distance (EMD). The CD between the point set \(P\) and \(P'\) is defined as follows,
\[
L_{CD}(P, P') = \sum_{p \in P} \min_{p' \in P'} ||p - p'||^2 + \sum_{p' \in P'} \min_{p \in P} ||p' - p||^2.
\]
Basically, this loss function is a nearest neighbour distance metric that bidirectionally measures the error in two sets. Every point in \(P\) is mapped to the nearest point in \(P'\), and
vice versa. The EMD from $P$ to $P'$ is defined as follows,
\[
\mathcal{L}_{EMD}(P, P') = \min_{\phi : P \rightarrow P'} \sum_{p \in P} ||p - \phi(p)||^2,
\]
where $\phi : P \rightarrow P'$ is a bijection. The EMD calculates a point-to-point mapping between two point clouds. The overall loss is as follows,
\[
\mathcal{L}(P, P') = \alpha \mathcal{L}_{CD}(P, P') + \beta \mathcal{L}_{EMD}(P, P'),
\]
where the hyperparameter $\alpha, \beta \geq 0$.

6. Experiments

We conduct experiments on one synthetic moving MNIST point cloud dataset and two large-scale real-world datasets, i.e., Argoverse [3] and nuScenes [2]. Models are trained for 200k iterations using the Adam [11] optimizer with a learning rate of $10^{-5}$. Gradients are clipped in the range $[-5, 5]$. The $\alpha$ and $\beta$ in Eq. (7) are set to 1.0. We follow point cloud reconstruction [1, 34, 19] to adopt CD and EMD for quantity evaluation. Model details are listed in Table 1. Max pooling is used for our models. For the two real-world datasets, because they contain too many points to be processed by the basic model, we only evaluate the advanced model. We randomly synthesize or sample 5,000 sequences from test subsets of these datasets as the test data.

We design three baselines for comparison. 1) Copy last input. This method does not make predictions and just copies the last input in the given sequence as outputs. Models that outperform this method can prove their effectiveness to model point cloud sequence. 2) PointNet++ · LSTM. This method first uses set abstraction (SA) layers from PointNet++ [23] to encode a point cloud to a global feature. Then, LSTMs take the global features as inputs and output predicted features. At last, like the advanced model, FP layers are used to propagate features to predicted displacements. 3) PointCNN · ConvLSTM. The PointCNN [15] provides an $\mathcal{X}$-Conv operator that can weight and permute input points and features before they are processed by a typical convolution. Therefore, PointCNN and ConvLSTM can be combined for point cloud sequence processing by using $\mathcal{X}$-Conv to permute ConvLSTM states. We evaluate this combination with the advanced model.

6.1. Moving MNIST Point Cloud

Experiments on the synthetic Moving MNIST point cloud dataset can provide some basic understanding of the behavior of the proposed recurrent units. To synthesize moving MNIST digit sequences, we use a generation process similar to that described in [26]. Each synthetic sequence consists of 20 consecutive point clouds, with 10 for inputs and 10 for predictions. Each point cloud contains one or two potentially overlapping handwritten digits moving and bouncing inside a $64 \times 64$ area. Pixels whose brightness values (ranged from 0 to 255) are less than 16 are removed. Locations of the remaining pixels are transformed to $(x, y)$ coordinates. The $z$-coordinate is set to 0 for all points. We randomly sample 128 points for one digit and 256 points for two digits.
Besides the three baselines, we also compare our methods with two video prediction models, i.e., ConvLSTM [25] and CubicLSTM [7]. Essentially, the Moving MNIST point cloud dataset is 2D. Digit point clouds can be first voxelized to images and then apply video-based methods to process them. Specifically, a pixel in the voxelization image is set to 1 if there exists a point at the position. Otherwise, the pixel is set to 0. In this way, a 2D point cloud sequence is converted to a video. For training, we use the binary cross entropy loss to optimize each output pixel. For evaluation, because the number of output points is usually not consistent with that of input points, we collect points whose brightness are in top 128 (for one digit) or top 256 (for two digits) as the output point cloud.

Experimental results are listed in Table 2 (our methods are with ball query). The kNN results are listed in Table 3. The performance of advanced models (ball query) with different numbers of layers is reported in Table 4. Two prediction examples are visualized in Figure 4 (our models are with the advanced architecture and ball query). PointRNN, PointGRU and PointLSTM are superior to other methods. For example, in Table 2, the CD of advanced PointLSTM on one-digit prediction is 1.16, less than ConvLSTM by 56.93, CubicLSTM by 8.35, PointNet++ · LSTM by 174.1 and PointCNN · ConvLSTM by 14.21.

Figure 4. Visualization of moving MNIST point cloud prediction. Left: one moving digit. Right: two moving digits.

| Method                  | #params | One digit  |          | Two digits |          |
|-------------------------|---------|------------|----------|------------|----------|
|                         |         | FLOPs      | CD       | EMD        | FLOPs    | CD       | EMD      |
| Copy last input         | -       | -          | 262.46   | 15.94      | -        | 140.14   | 15.18    |
| ConvLSTM [25] (voxel-based) | 2.16   | 345.26     | 58.09    | 8.85       | 345.26   | 13.02    | 5.99     |
| CubicLSTM [7] (voxel-based) | 6.08   | 448.88     | 9.51     | 4.20       | 448.88   | 6.19     | 4.42     |
| PointNet++ · LSTM       | 2.50    | 1.01       | 175.26   | 15.86      | 1.94     | 100.08   | 15.10    |
| PointCNN · ConvLSTM     | 2.97    | 1.43       | 15.37    | 5.45       | 2.86     | 49.92    | 10.31    |
| PointRNN (ours)         | 0.27    | 5.32       | 2.65     | 2.79       | 10.05    | 16.98    | 6.62     |
| PointGRU (ours)         | 0.96    | 14.84      | 1.43     | 2.03       | 29.73    | 7.29     | 4.87     |
| PointLSTM (ours)        | 1.22    | 24.74      | 1.40     | 2.00       | 49.55    | 49.92    | 10.31    |

Table 2. Prediction error (CD and EMD), #params (million) and FLOPs (billion) on the moving MNIST point cloud dataset.

| Method                  | #params | One digit  |          | Two digits |          |
|-------------------------|---------|------------|----------|------------|----------|
|                         |         | FLOPs      | CD       | EMD        | FLOPs    | CD       | EMD      |
| PointRNN (basic)        | 0.27    | 5.32       | 2.65     | 2.79       | 10.05    | 16.98    | 6.62     |
| PointGRU (basic)        | 0.96    | 14.84      | 1.43     | 2.03       | 29.73    | 7.29     | 4.87     |
| PointLSTM (basic)       | 1.22    | 24.74      | 1.40     | 2.00       | 49.55    | 49.92    | 10.31    |
| PointRNN (advanced)     | 0.36    | 0.77       | 1.78     | 2.32       | 1.54     | 11.44    | 5.87     |
| PointGRU (advanced)     | 1.04    | 2.00       | 1.18     | 1.85       | 4.00     | 5.79     | 4.49     |
| PointLSTM (advanced)    | 1.30    | 3.18       | 1.16     | 1.78       | 6.37     | 5.18     | 4.21     |

Table 3. Prediction error of PointRNN, PointGRU and PointLSTM with kNN on the moving MNIST point cloud dataset.

| Method                  | #params | One digit  |          | Two digits |          |
|-------------------------|---------|------------|----------|------------|----------|
|                         |         | FLOPs      | CD       | EMD        | FLOPs    | CD       | EMD      |
| PointRNN (basic)        | 0.025   | 0.176      | 22.12    | 7.79       | 0.354    | 29.92    | 8.30     |
| PointGRU (basic)        | 0.051   | 0.454      | 19.74    | 7.64       | 0.912    | 19.41    | 6.99     |
| PointLSTM (basic)       | 0.060   | 0.710      | 17.34    | 7.20       | 1.427    | 18.37    | 7.14     |
| PointRNN (advanced)     | 0.109   | 0.485      | 14.54    | 6.42       | 0.972    | 16.51    | 6.62     |
| PointGRU (advanced)     | 0.268   | 1.224      | 8.88     | 5.33       | 2.453    | 8.22     | 5.13     |
| PointLSTM (advanced)    | 0.328   | 1.962      | 1.38     | 4.53       | 3.931    | 7.82     | 4.96     |

Table 4. Performance of advanced models (ball query) with different numbers of layers on the moving MNIST point cloud dataset.
Voxel-based models are not sensitive to sparse point clouds. There is only 128 points per point cloud area, which is too sparse for voxel-based methods. According to Table 2 and Figure 4, the predictions of ConvLSTM and CubicLSTM on one moving digit are worse than those on two moving digits, which is counterintuitive. One possible reason is that, for one-digit prediction, there is only 128/(64 × 64) = 3.125% point area, which is too sparse for voxel-based methods.

Compared with the basic architecture, the advanced architecture significantly reduces computation. For example, in Table 2, for the one-digit prediction, the FLOPs of the basic PointGRU is 14.84 billion and the FLOPs of the advanced PointGRU is only 2.00 billion.

The advanced architecture also achieves lower prediction error than the basic architecture. For example, in Table 2, for the two-digit prediction, the CD of the basic PointRNN is 16.08 and the CD of the advanced PointRNN is 11.44.

The ball query is superior to the kNN. For example, when applying the advanced PointLSTM model to the two-digit prediction, the EMD of the ball query is 4.21 (in Table 2) and the EMD of the kNN is 5.13 (in Table 3).

Hierarchical structure by stacking multiple layers can effectively improve accuracy. For example, on the one-digit prediction, the CDs of the advanced PointGRU with one, two and three layers are 3.72, 1.48 and 1.18, respectively.

Learning via global features fails to make predictions. In Table 2, although the PointNet++·LSTM combination obtains lower CDs than the copy-last-input method, their EMDs are similar. In Figure 4, PointNet++·LSTM fails to predict neither appearance nor motion.

Point-based models consume less FLOPs than voxel-based models. Taking the two-digit prediction for instance, the basic PointLSTM model consumes only 49.55 billion FLOPs, while the voxel-based ConvLSTM consumes 345.26 billion FLOPs. The reason is that, the FLOPs of point-based models only depends on the number of input points, while voxel-based models have to process the entire space no matter how many points are actually in the space.

Voxel-based models are not sensitive to sparse point clouds. According to Table 2 and Figure 4, the predictions of ConvLSTM and CubicLSTM on one moving digit are worse than those on two moving digits, which is counterintuitive. One possible reason is that, for one-digit prediction, there is only 128/(64 × 64) = 3.125% point area, which is too sparse for voxel-based methods.

### 6.2. Argoverse and nuScenes

Argoverse [3] and nuScenes [2] are two large-scale autonomous driving datasets. Argoverse data is collected by a fleet of autonomous vehicles in Pittsburgh (86km) and Miami (204km). The nuScenes data is recorded in Boston and Singapore, with 15h of driving data (242km travelled at an average of 16km/h). These datasets are collected by multiple sensors, including LiDAR, RADAR, camera, etc. In this paper, we only use the data from LiDAR sensors, without any human-annotated supervision, for moving point cloud prediction. Details about Argoverse and nuScenes are listed in Table 5.

Predicting moving point clouds on real-world driving datasets is considerably challenging. Content of a long driving point cloud sequence may change dramatically. Because we can not predict what are not provided in the given inputs, models are asked to make short-term prediction on the driving datasets. Each driving log is considered as a driving log for training, with 5 for inputs and 5 for predictions. Since Argoverse and nuScenes are high-resolution, using all points requires considerable computation and running memory. Therefore, for each cloud, we only use the points whose coordinates are in the range [−5m, 5m]. Then, we randomly sample 1,024 points for training.
points from the range as inputs and ground truths. Batch size is set to 4.

Experimental results are listed in Table 6 (our methods are with ball query). The result details over time are illustrated in Figure 5. The $k$NN results are listed in Table 7.

PointRNN, PointGRU and PointLSTM are superior to PointNet++ · LSTM and PointCNN · ConvLSTM. For example, in Table 6, the CD of advanced PointRNN on Argoverse is 0.2789, less than PointNet++ · LSTM by 0.1037 and PointCNN · ConvLSTM by 0.0668. In Figure 5, our models achieve lower prediction error than PointNet++ · LSTM and PointCNN · ConvLSTM at most time steps.

The $k$NN and ball query achieve similar accuracy on Argoverse and nuScenes. For example, when applying the advanced PointLSTM model to Argoverse, the EMD of the ball query is 0.8892 (in Table 6) and the EMD of the $k$NN is 0.8856 (in Table 7).

We visualize two prediction examples in Figure 6 (our models are with the ball query). For the first example, PointNet++ · LSTM and PointCNN · ConvLSTM fail to predict car A or car B to move backward, while PointRNN, PointGRU and PointLSTM can predict the movement of car B correctly and the movement of car A with a little error at the rear. For the second example, PointNet++ · LSTM fails to predict the movements of the pedestrian or the two cars. The PointCNN · ConvLSTM model can predict the pedestrian to a certain degree, but fails to predict the cars. PointRNN, PointGRU and PointLSTM can predict the movements of both the pedestrian and the cars.

We also visualize the predicted scene flow ($\Delta P$) by the advanced PointLSTM model (ball query) in Figure 7. The point whose flow magnitude $||\Delta p||$ is less than 0.01 m is removed. The model outputs reasonable scene flow. This suggests a promising method to learn 3D scene flow from point clouds in an unsupervised manner.

7. Conclusion

This paper proposes a PointRNN, PointGRU and PointLSTM for point cloud sequence processing. Experimental results on moving point cloud prediction demonstrate their ability to model point cloud sequences. The proposed units have the potential for other temporal-related applications, such as 3D action recognition based on point cloud and sequential scene semantic segmentation. More effective spatiotemporally-local correlation methods can be studied to improve PointRNN, PointGRU and PointLSTM.
**PointRNN: Point Recurrent Neural Network for Moving Point Cloud Processing**

**SUPPLEMENTARY MATERIAL**

## 1. PointGRU and PointLSTM

In this section, we provide detailed breakdowns of PointGRU and PointLSTM in the main paper by comparing them with the conventional GRU and LSTM, respectively.

### 1.1. PointGRU

To solve the exploding and vanishing gradient problems of the standard RNN, GRU introduces a update gate $z_t \in \mathbb{R}^d$ and a reset gate $r_t \in \mathbb{R}^d$ to decide what information should be passed to the output. The update gate helps the model to determine how much of the past information (from previous time steps) needs to be passed along to the future. The reset gate is used to decide how much of the past information to forget. The current state $\tilde{s}_t \in \mathbb{R}^d$ uses the reset gate to store the relevant information from the past. The final state $s_t \in \mathbb{R}^d$ uses the update gate to determine what to collect from the current state $\tilde{s}_t$ and what from the previous state $s_{t-1}$.

![PointGRU and PointLSTM](Figure1.png)

**Figure 1.** PointGRU and PointLSTM. By default, the $Y_t$ of PointGRU is set to $S_t$ and the $Y_t$ of PointLSTM set to $H_t$.

| Gate/State | GRU | PointGRU |
|------------|-----|----------|
| update gate | $z_t = \sigma(r\text{nn}(x_t, s_{t-1}; W_z, b_z))$ | $S_t = \sigma(\text{point-rnn}((P_t, X_t), (P_{t-1}, S_{t-1}); W_z, b_z))$ |
| reset gate  | $r_t = \sigma(r\text{nn}(x_t, s_{t-1}; W_r, b_r))$ | $S_{t-1} = \text{point-rnn}((P_t, \text{None}), (P_{t-1}, S_{t-1}); W_r, b_r)$ |
| state       | $\tilde{s}_t = \tanh(W_s \cdot [x_t, r_t \odot s_{t-1}] + b_s)$ | $S_t = \tanh(W_s \cdot [x_t, r_t \odot S_{t-1}] + b_s)$ |
|            | $s_t = z_t \odot \tilde{s}_t + (1 - z_t) \odot \tilde{s}_t$ | $S_t = Z_t \odot S_{t-1} + (1 - Z_t) \odot \tilde{s}_t$ |

| Gate/State | LSTM | PointLSTM |
|------------|-------|-----------|
| input gate | $i_t = \sigma(r\text{nn}(x_t, h_{t-1}; W_i, b_i))$ | $C_t = \sigma(\text{point-rnn}((P_t, X_t), (P_{t-1}, H_{t-1}); W_i, b_i))$ |
| forget gate| $f_t = \sigma(r\text{nn}(x_t, h_{t-1}; W_f, b_f))$ | $F_t = \sigma(\text{point-rnn}((P_t, X_t), (P_{t-1}, H_{t-1}); W_f, b_f))$ |
| output gate| $o_t = \sigma(r\text{nn}(x_t, h_{t-1}; W_o, b_o))$ | $O_t = \sigma(\text{point-rnn}((P_t, X_t), (P_{t-1}, H_{t-1}); W_o, b_o))$ |
| cell state | $\tilde{c}_t = \tanh(r\text{nn}(x_t, h_{t-1}; W_c, b_c))$ | $C_{t-1} = \text{point-rnn}((P_t, \text{None}), (P_{t-1}, C_{t-1}); W_c, b_c)$ |
| hidden state| $h_t = o_t \odot \tanh(c_t)$ | $C_t = F_t \odot C_{t-1} + I_t \odot \tilde{c}_t$ |

| Gate/State | LSTM | PointLSTM |
|------------|-------|-----------|
| output gate | $o_t = \sigma(r\text{nn}(x_t, h_{t-1}; W_o, b_o))$ | $O_t = \sigma(\text{point-rnn}((P_t, X_t), (P_{t-1}, H_{t-1}); W_o, b_o))$ |
| hidden state| $h_t = o_t \odot \tanh(c_t)$ | $H_t = O_t \odot \tanh(C_t)$ |

| Table 1. Comparison between GRU and PointGRU. |
| Table 2. Comparison between LSTM and PointLSTM. |

The update gate $z_t$ and the reset gate $r_t$ of GRU are calculated by the $r\text{nn}$ function in the main paper, which aggregates the previous state $s_{t-1}$ and the current input $x_t \in \mathbb{R}^d$ with concatenation. By default, GRU uses its state $s_t$ as output $y_t$. To apply GRU to point cloud, we replace the input in the reset and update gates with the proposed point-$r\text{nn}$ function, forming the PointGRU unit. The comparison of update steps between GRU and PointGRU are shown in Table 1. Because point clouds are unordered, PointGRU leverages an additional step $S_{t-1}$ to weight and permute $S_{t-1}$ according to the current input points $P_t$. Similar to PointRNN, the input of PointGRU is $(P_t, X_t)$, the state is $(P_t, S_t)$ and the output is $(P_t, Y_t)$ (shown in Figure 1(a)). By default, $Y_t$ is set to $S_t$.

### 1.2. PointLSTM

The conventional LSTM unit is composed of a cell state $c_t \in \mathbb{R}^d$, a hidden state a cell state $h_t \in \mathbb{R}^d$, an input gate $i_t \in \mathbb{R}^d$, an output gate $o_t \in \mathbb{R}^d$ and a forget gate $f_t \in \mathbb{R}^d$. The cell state acts as an accumulator of the sequence or the
temporal information over time. Specifically, the current input $x_t$ will be integrated to $c_t$ if the input gate $i_t$ is activated. Meanwhile, the previous cell state $c_{t-1}$ may be forgotten if the forget gate $f_t$ turns on. Whether $c_t$ will be propagated to the hidden state $h_t \in \mathbb{R}^d$ is controlled by the output gate $o_t$. By default, LSTM uses its hidden state $h_t$ as output $y_t$.

Similar to the gates in GRU, the cell state $c_t$, input gate $i_t$, output gate $o_t$ and forget gate $f_t$ of LSTM are calculated by the run function. We replace run in LSTM with the point-run function, forming the PointLSTM. The comparison of update steps between LSTM and PointLSTM are shown in Table 2. Like from GRU to PointGRU, PointLSTM has an additional step $\tilde{C}_{t-1}$ to weight and permute $C_{t-1}$ according to the current input points $P_t$. The input of PointLSTM is $(P_t, X_t)$, the state is $(P_t, H_t, C_t)$ and the output is $(P_t, Y_t)$ (shown in Figure 1(b)). By default, $Y_t$ is set to $H_t$.

2. More Experiments

In this section we provide more experimental results to validate and analyze the proposed PointRNN, PointGRU and PointLSTM.

2.1. Prediction on Argoverse and nuScenes with Different Ball Query Radiuses

In the main paper, the search radiuses of the architecture for Argoverse and nuScenes are fixed as $S(0.5) \rightarrow PU(1.0) \rightarrow SG(1.0) \rightarrow PU(2.0) \rightarrow SG(2.0) \rightarrow PU(4.0)$. In this section, we evaluate another two radius settings, to explore the impact of ball query radius. Note that, the number of neighbor (i.e., $k$) in each component is not changed.

- Setting 1: $S(0.05) \rightarrow PU(0.1) \rightarrow SG(0.1) \rightarrow PU(0.2) \rightarrow SG(0.2) \rightarrow PU(0.4)$.
- Setting 2: $S(0.15) \rightarrow PU(0.3) \rightarrow SG(0.3) \rightarrow PU(0.6) \rightarrow SG(0.6) \rightarrow PU(1.2)$.
- Setting 3: $S(0.25) \rightarrow PU(0.5) \rightarrow SG(0.5) \rightarrow PU(1.0) \rightarrow SG(1.0) \rightarrow PU(2.0)$.

The results are listed in Table 3. The radius of ball query have an obvious impact on the prediction accuracy. For example, the CD of PointRNN with setting 1 on Argoverse is 0.2875, while that with setting 2 is 0.3001. This suggests that, for ball query, it can effectively reduce prediction error by carefully choosing search radiuses.

| Method    | Argoverse CD | Argoverse EMD | nuScenes CD | nuScenes EMD |
|-----------|--------------|---------------|-------------|--------------|
| Setting 1 | 0.2875       | 0.8906        | 0.0625      | 0.3750       |
| PointRNN  |              |               |             |              |
| PointGRU  | 0.3292       | 0.9834        | 0.0620      | 0.3733       |
| PointLSTM | 0.3556       | 0.9610        | 0.0617      | 0.3743       |
| Setting 2 | 0.3001       | 0.8924        | 0.0610      | 0.3724       |
| PointRNN  |              |               |             |              |
| PointGRU  | 0.3084       | 0.9803        | 0.0623      | 0.3719       |
| PointLSTM | 0.3036       | 0.9248        | 0.0598      | 0.3727       |
| Setting 3 | 0.2798       | 0.8893        | 0.0610      | 0.3719       |
| PointRNN  |              |               |             |              |
| PointGRU  | 0.2839       | 0.9388        | 0.0606      | 0.3718       |
| PointLSTM | 0.3370       | 0.9590        | 0.0607      | 0.3727       |

Table 3. Prediction error with different ball query radiuses on Argoverse and nuScenes.

2.2. Average pooling

In the main paper, PointRNNs, PointGRUs and PointLSTMs are with max pooling. In this section, we investigate average pooling on the moving MNIST point cloud dataset. The results are reported in Table 4. Compared with the moving MNIST point cloud results in the main paper, max pooling is better than average pooling.

| Method   | 1NN One digit | 1NN Two digits | Ball query One digit | Ball query Two digits |
|----------|---------------|----------------|----------------------|----------------------|
|          | CD | EMD | CD | EMD | CD | EMD | CD | EMD |
| Basic    |     |     |     |     |     |     |     |     |
| PointRNN | 10.81 | 4.82 | 25.89 | 8.58 | 4.85 | 3.54 | 23.04 | 7.45 |
| PointGRU | 4.82 | 3.36 | 20.42 | 7.85 | 2.44 | 2.50 | 13.10 | 6.05 |
| PointLSTM | 5.14 | 3.41 | 28.34 | 9.69 | 2.58 | 2.54 | 13.63 | 6.19 |
| Advanced |     |     |     |     |     |     |     |     |
| PointRNN | 4.04 | 3.16 | 16.86 | 6.72 | 2.52 | 2.73 | 13.50 | 6.50 |
| PointGRU | 2.70 | 2.62 | 12.50 | 6.27 | 1.51 | 2.06 | 8.79 | 5.27 |
| PointLSTM | 2.75 | 2.58 | 14.13 | 6.58 | 1.65 | 2.14 | 18.82 | 5.56 |

Table 4. Prediction error of PointRNN, PointGRU and PointLSTM with average pooling on moving MNIST point cloud.

2.3. More Visualization Examples

In this section, we provide more visualization examples on the moving MNIST point cloud, Argoverse and nuScenes datasets. Our methods are with the ball query.
| Inputs | Ground truths |
|--------|---------------|
| ![ConvLSTM (voxel)](image) | ![ConvLSTM (voxel)](image) |
| ![CubicLSTM (voxel)](image) | ![CubicLSTM (voxel)](image) |
| ![PointNet++ · LSTM](image) | ![PointNet++ · LSTM](image) |
| ![PointCNN · ConvLSTM](image) | ![PointCNN · ConvLSTM](image) |
| ![PointRNN (basic)](image) | ![PointRNN (basic)](image) |
| ![PointRNN (advanced)](image) | ![PointRNN (advanced)](image) |
| ![PointGRU (basic)](image) | ![PointGRU (basic)](image) |
| ![PointGRU (advanced)](image) | ![PointGRU (advanced)](image) |
| ![PointLSTM (basic)](image) | ![PointLSTM (basic)](image) |
| ![PointLSTM (advanced)](image) | ![PointLSTM (advanced)](image) |

Figure 2. Visualization of moving MNIST point cloud prediction with one digit. For PointRNN, PointGRU and PointLSTM, both the basic model and the advanced model make correct appearances and motions.
| Inputs | Ground truths |
|--------|--------------|
| ![ConvLSTM (voxel)](image) | ![ConvLSTM (voxel)](image) |
| ![CubicLSTM (voxel)](image) | ![CubicLSTM (voxel)](image) |
| ![PointNet++ · LSTM](image) | ![PointNet++ · LSTM](image) |
| ![PointCNN · ConvLSTM](image) | ![PointCNN · ConvLSTM](image) |
| ![PointRNN (basic)](image) | ![PointRNN (basic)](image) |
| ![PointRNN (advanced)](image) | ![PointRNN (advanced)](image) |
| ![PointGRU (basic)](image) | ![PointGRU (basic)](image) |
| ![PointGRU (advanced)](image) | ![PointGRU (advanced)](image) |
| ![PointLSTM (basic)](image) | ![PointLSTM (basic)](image) |
| ![PointLSTM (advanced)](image) | ![PointLSTM (advanced)](image) |

Figure 3. Visualization of moving MNIST point cloud prediction with two digits. For PointRNN, PointGRU and PointLSTM, the advanced model generates clearer digits than the basic model.
Figure 4. Visualization of moving point cloud prediction on Argoverse. (a) In the bird’s-eye view example, two cars are moving backward. The PointRNN, PointGRU and PointLSTM correctly predict them to move backward. (b) In the worm’s-eye view example, a truck and a car are moving backward. At last, the car moves out of the field of visualization. The PointRNN, PointGRU and PointLSTM correctly predict the truck and the car to move backward. At the last time step, predictions of the truck made by PointNet++ · LSTM and PointCNN · ConvLSTM are severely distorted.
Figure 5. Visualization of moving point cloud prediction on nuScenes. (a) In the bird’s-eye view example, two cars are moving backward. The PointRNN, PointGRU and PointLSTM correctly predict them to move backward. In the worm’s-eye view example, there are also two cars which are moving backward. (b) In each example, there are also two cars that are moving backward. At last, car A moves out of the field of visualization. The PointRNN, PointGRU and PointLSTM correctly predict car B to move backward and car A to gradually move out of the field of visualization.