Sparse Convolutions on Continuous Domains for Point Cloud and Event Stream Networks

Dominic Jack[1 [0000−0002−8371−3502]], Frederic Maire[1 [0000−0002−6212−7651]], Simon Denman[1 [0000−0002−0983−5480]], and Anders Eriksson[2 [0000−0003−2652−7110]]

[1] Queensland University of Technology, QLD, Australia
thedomjack@gmail.com,{f.maire,s.denman}@qut.edu.au
[2] University of Queensland, QLD, Australia
a.eriksson@uq.edu.au

Abstract. Image convolutions have been a cornerstone of a great number of deep learning advances in computer vision. The research community is yet to settle on an equivalent operator for sparse, unstructured continuous data like point clouds and event streams however. We present an elegant sparse matrix-based interpretation of the convolution operator for these cases, which is consistent with the mathematical definition of convolution and efficient during training. On benchmark point cloud classification problems we demonstrate networks built with these operations can train an order of magnitude or more faster than top existing methods, whilst maintaining comparable accuracy and requiring a tiny fraction of the memory. We also apply our operator to event stream processing, achieving state-of-the-art results on multiple tasks with streams of hundreds of thousands of events.

Keywords: Convolution, Point Clouds, Event Cameras, Deep Learning

1 Introduction

Deep learning has exploded in popularity since AlexNet [1] achieved groundbreaking results in image classification [2]. The field now boasts state-of-the-art performance in fields as diverse as medical imaging [3], natural language processing [4], and molecular design [5].

Robotics [6] applications are of particular interest due to their capacity to revolutionize society in the near future. Driverless cars [7] specifically have attracted enormous amounts of research funding, with advanced systems being built with multi-camera setups [8], active LiDAR sensors [9], and sensor fusion approaches [10].

At the other end of the spectrum, small mobile robotics applications and mobile devices benefit from an accurate 3D understanding of the world. These platforms generally don't have access to large battery stores or computationally hefty hardware, so efficient computation is essential. Even where compute

This research was supported by the Australian Research Council through the grant ARC FT170100072.

is available, the cost of energy alone can be prohibitive, and the research community is beginning to appreciate the environmental cost of training massive power-hungry algorithms in data centres [11].

The convolution operator has been a critical component of almost all recent advances in deep learning for computer vision. However, implementations designed for use with images cannot be used for data types that are not defined on a regular grid. Consider for example event cameras, a new type of sensor which shows great promise, particularly in the area of mobile robotics. Rather than reporting average intensities of every pixel at a uniform frame rate, pixels in an event camera fire individual events when they observe an intensity change. The result is a sparse signal with very fast response time, high dynamic range and low power usage. Despite the potential, this vastly different data encoding means that a traditional 2D convolution operation is no longer appropriate.

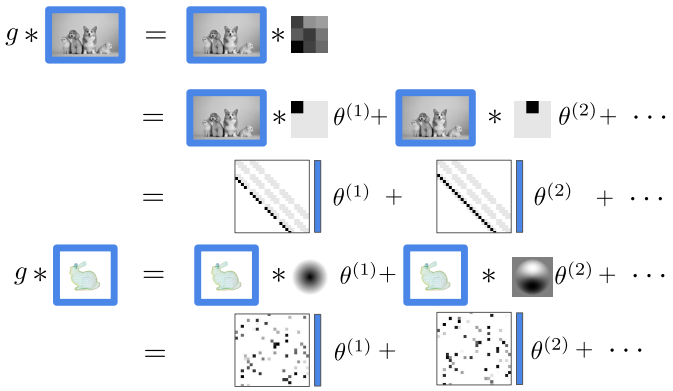


Fig. 1: Learned image convolutions can be thought of as linear combinations of static basis convolutions, where the linear combination is learned. Each basis convolution can be expressed as a sparse-dense matrix product. We take the same approach with point clouds and event streams.

In this work, we investigate how the convolution operator can be applied to two non-image input sources: point clouds and event streams. In particular, our contributions are as follows.

1. We implement a convolution operator for sparse inputs on continuous domains using only matrix products and addition during training. While others have *proposed* such an operator, we believe we are the first to *implement* one without compromising the mathematical definition of convolution.
2. We discuss implementation details essential to the feasible training and deployment of networks using our operator on modest hardware. We demon-

strate speed-ups of an order of magnitude or more compared to similar methods with a memory foot-print that allows for batch sizes in the thousands.
3. For point clouds, we discuss modifications that lead to desirable properties like robustness to varying density and continuity, and demonstrate that relatively small convolutional networks can perform competitively with much larger, more expensive networks.
4. For event streams, we demonstrate that convolutions can be used to learn features from spiking network spike trains. By principled design of our kernels, we propose two implementations of the same networks: one for learning that takes advantage of modern accelerator hardware, and another for asynchronous deployment which can provide features or inferences associated with events as they arrive. We demonstrate the effectiveness of our learned implementation by achieving state-of-the-art results on multiple classification benchmarks, including a 44% reduction in error rate on sign language classification [12].

2 Prior Work

Point Clouds Early works in point cloud processing – Pointnet [13] and Deep Sets [14] – use point-wise shared subnetworks and order invariant pooling operations. The successor to Pointnet, Pointnet++ [15] was (to the best of our knowledge) the first to take a hierarchical approach, applying Pointnet submodels to local neighborhoods.

SO-Net [16] takes a similar hierarchical approach to Pointnet++, though uses a different method for sampling and grouping based on self-organizing maps. DGCNN [17] applies graph convolutions to point clouds with edges based on spatial proximity. KCNet [18] uses dynamic kernel points in correlation layers that aim to learn features that encapsulate the relationships between those kernel points and the input cloud. While most approaches treat point clouds as unordered sets by using order-invariant operations, PointCNN [19] takes the approach of learning a canonical ordering over which an order-dependent operation is applied. SpiderCNN [20] and FlexConv [21] each bring their own unique interpretation to generalizing image convolutions to irregular grids. While SpiderCNN focuses on large networks for relatively small classification and segmentation problems, FlexConv utilizes a specialized GPU kernel to apply their method to point clouds with millions of points.

Event Stream Networks Compared to standard images, relatively little research has been done with event networks. Interest has started to grow recently with the availability of a number of event-based cameras [22, 23] and publicly available datasets [23–26, 12].

A number of approaches utilize the extensive research in standard image processing by converting event streams to images [25, 27]. While these can leverage existing libraries and cheap hardware optimized for image processing, the

necessity to accumulate synchronous frames prevents them from taking advantage of many potential benefits of the data format. Other approaches look to biologically-inspired spiking neural networks (SNNs) [28–30]. While promising, these networks are difficult to train due to the discrete nature of the spikes.

Other notable approaches include the work of Lagorce *et al.* [31], who introduce hierarchical time-surfaces to describe spatio-temporal patterns; Sironi *et al.* [26], who show histograms of these time surfaces can be used for object classification; and Bi *et al.* [12], who use graph convolution techniques operating over graphs formed by connecting close events in space-time.

Sparse Convolutions Sparse convolutions have been used in a variety of ways in deep learning before. Liu *et al.*[32] and Park *et al.*[33] demonstrate improved speed from using implementations optimized for sparse kernel on discrete domains, while there are various voxel-based approaches [34–36] that look at convolutions on discrete sparse inputs and dense kernels. Other approaches involve performing dense discrete convolutions on interpolated projections [37, 38].

3 Method Overview

For simplicity, we formulate continuous domain convolutions in the context of physical point clouds in Section 3.1, before modifying the approach for event streams in Section 3.2. A summary of notation used in this section is provided in the supplementary material.

3.1 Point Cloud Convolutions

We begin by considering the mathematical definition of a convolution of a function h with a kernel g,

$$(h * g)(t) = \int_{\mathcal{D}} h(\tau) g(t - \tau)\, d\tau. \tag{1}$$

We wish to evaluate the convolution of a function with values defined at fixed points x_j in an input cloud $\mathcal{X}$ of size S, at a finite set of points x'_i in an output cloud $\mathcal{X}'$ of size S'. We denote a single feature for each point in these clouds $f \in \mathbb{R}^S$ and $f' \in \mathbb{R}^{S'}$ respectively. For the moment we assume coordinates for both input and output clouds are given. In practice it is often the case that only the input coordinates are given. We discuss choices of output clouds in subsequent sections.

By considering our convolved function h to be the sum of scaled Dirac delta functions δ centred at the point coordinates,

$$h(x) = \sum_j f_j \delta(x - x_j), \tag{2}$$

Equation 1 reduces to

$$f'_i = \sum_{x_j \in \mathcal{N}_i} f_j g(x'_i - x_j), \tag{3}$$

where $\mathcal{N}_i$ is the set of points in the input cloud within some neighborhood of the output point x'_i. We refer to pairs of points $\{x_j, x'_i\}$ where $x_j \in \mathcal{N}_i$ as an *edge*, and the difference in coordinates $\Delta x_{ij} = x'_i - x_j$ as the *edge vector*.

Like Groh *et al.*[21], we use a kernel made up of a linear combination of M unlearned basis functions p_m,

$$g(\Delta x; \theta) = \sum_m p_m(\Delta x)\theta_m, \tag{4}$$

where θ_m are learnable parameters. As with Groh *et al.*, we use geometric monomials for our basis function. Substituting this into Equation 3 and reordering summations yields

$$f'_i = \sum_m \sum_{x_j \in \mathcal{N}_i} p_m(\Delta x_{ij}) f_j \theta_m. \tag{5}$$

We note the inner summation can be expressed as a sparse-dense matrix product,

$$f' = \sum_m N^{(m)} f \theta_m, \tag{6}$$

This is visualized in Figure 1. Neighborhood matrices $N^{(m)}$ have the same sparsity structure for all m. Values $n_{ij}^{(m)}$ are given by the corresponding basis functions evaluated at edge vectors,

$$n_{ij}^{(m)} = \begin{cases} p_m(\Delta x_{ij}) & x_j \in \mathcal{N}_i, \\ 0 & \text{otherwise.} \end{cases} \tag{7}$$

Generalizing to multi-channel input and output features $F \in \mathbb{R}^{S \times Q}$ and $F' \in \mathbb{R}^{S' \times P}$ respectively, this can be expressed as a sum of matrix products,

$$F' = \sum_m N^{(m)} F \Theta^{(m)}, \tag{8}$$

where $\Theta^{(m)} \in \mathbb{R}^{Q \times P}$ is a matrix of learned parameters.

The elegance of this representation should not be understated. $N^{(m)}$ is a sparse matrix defined purely by relative point coordinates and choice of basis functions. $\Theta^{(m)}$ is a dense matrix of parameter weights much like traditional convolutional layers, and F' and F are feature matrices with the same structure, allowing networks to be constructed in much the same way as image CNNs.

We now identify three implementations with analogues to common image convolutions. A summary is provided in Table 1.

Down-Sampling Convolutions Convolutions in which there are fewer output points than input points and more output channels than input channels are more efficiently computed left-to-right, *i.e.* as $(N^{(m)}F)\Theta^{(s)}$. These are analogous to conventional strided image convolutions.

Up-Sampling Convolutions Convolutions in which there are more output points than input points and fewer output channels than input channels are more efficiently computed right-to-left, i.e. $N^{(m)}\left(F\Theta^{(m)}\right)$. These are analogous to conventional fractionally strided or transposed image convolutions.

Featureless Convolutions The initial convolutions in image CNNs typically have large receptive fields and a large increase in the number of filters. For point clouds, there are often no input features at all – just coordinates. In this instance the convolution reduces to a sum of kernel values over the neighborhood. In the multi-input/output channel context this is given by

$$Z = \tilde{G}\Phi_0, \tag{9}$$

where $\Phi_0 \in \mathbb{R}^{S\times Q}$ is the learned matrix and $\tilde{G} \in \mathbb{R}^{N'\times S}$ is a dense matrix of summed monomial values

$$\tilde{g}_{is} = \sum_j \hat{n}_{ij}^{(m)}. \tag{10}$$

	Opt. Cond.	Form	Mult. Adds	Mem.
In Place	$Q = P$ $S' = S$	$\sum_m N^{(m)} F\Theta^{(m)}$	$MP(E + SP)$	SP
Down-Sample	$Q < P$ $S' < S$	$\sum_m \left(N^{(m)} F\right)\Theta^{(m)}$	$MQ(E + S'P)$	$S'Q$
Up-Sample	$Q > P$ $S' > S$	$\sum_m N^{(m)}\left(F\Theta^{(m)}\right)$	$MP(E + SQ)$	SP
Featureless	$F = 1$	$\tilde{G}\Phi_0$	MSP	-

Table 1: Time complexity of different point cloud convolution operations and theoretical space complexity of intermediate terms (Mem). The matrix product for in place convolutions can be evaluated in either order.

Neighborhoods To be consistent with the mathematical definition of convolution, the neighborhood of each point should be fixed, which precludes the use of k-nearest neighbors (kNN), despite its prevalence in the literature [21, 20, 15, 39]. The obvious choice of a neighborhood is a ball. Equation 8 can be implemented trivially using either kNN or ball neighborhoods, though from a deep learning perspective each neighborhood has its own advantages and disadvantages.

Predictable computation time: The sparse-dense matrix products have computation proportional to the number of edges. For kNN this is proportional to the output cloud size, but is less predictable when using ball-searches.

Robustness to point density: Implementations based on each neighborhood react differently to variations in point density. As the density increases, kNN implementations shrink their receptive field. On the other hand, ball-search implementations suffer from increased computation time and output values proportional to the density.

Discontinuity in point coordinates: Both neighborhood types result in operations that are discontinuous in point coordinates. kNN convolutions are discontinuous as the k^{th} and $(k+1)^{\text{th}}$ neighbors of each point pass each other. Ball-search convolutions have a discontinuity at the ball-search radius.

Symmetry: Connectedness in ball-neighborhoods is symmetric – i.e. if $x'_i \in \mathcal{N}_j$ then $x_j \in \mathcal{N}_i$ for neighborhood functions with the same radius. This means the neighborhood matrix N_{IJ} between sets $\mathcal{X}_I$ and $\mathcal{X}_J$ is related to the reversed neighborhood by $N_{IJ} = N_{JI}^T$ (up to a possible difference in sign due to the monomial value). This allows for shared computation between different layers.

Transposed Neighborhood Occupancy: For kNN, all neighborhood matrices are guaranteed to have k entries in each row. This guarantees there will be no empty rows, and hence no empty neighborhoods. Ball search neighborhoods do not share this property, and there is no guarantee points will have any neighbors. This is important for transposed convolutions, where points may rely on neighbors from a lower resolution cloud to seed their features.

Subsampling Thus far we have remained silent as to how the S' output points making up $\mathcal{X}'$ are chosen. In-place convolutions can be performed with the same input and output clouds, but to construct networks we would like to reduce the number of points as we increase the number of channels in a similar way to image CNNs. We adopt a similar approach to Pointnet++ [15] in that we sample a set of points from the input cloud. Pointnet++ [15] selects points based on the first S' points in iterative farthest point (IFP) ordering, whereby a fixed number of points are iteratively selected based on being farthest from the currently selected points. For each point selected, the distance to all other points has to be computed, resulting in an $\mathcal{O}(S'S)$ implementation.

To improve upon this, we begin by updating distances only to points within a ball neighborhood – a neighborhood that may have already been computed for the previous convolution. By storing the minimum distances in a priority queue, this sampling process still gives relatively uniform coverage like the original IFP, but can be computed in $\mathcal{O}(S'\bar{k})$, where $\bar{k}$ is the average number of neighbors of each point.

We also propose to terminate searching once this set of neighborless candidates has been exhausted, rather than iterating for a fixed number of steps. We refer to this as *rejection sampling*. This results in point clouds of different sizes, but leads to a more consistent number of edges in subsequent neighborhood matrices. It also guarantees all points in the input cloud will have a neighbor in the output cloud. We provide pseudo-code for these algorithms and illustrations in the supplementary material.

Weighted Convolutions To address both the discontinuity at the ball radius and the neighbor count variation inherent to using balls, we propose using a

weighted average convolution by weighting neighboring values by some continuous function w which decreases to zero at the ball radius,

$$\hat{n}_{ij}^{(m)} = \frac{1}{W_i} w_{ij} n_{ij}^{(m)} \tag{11}$$

where $w_{ij} = w(|\Delta x_{ij}|)$ and $W_i = \sum_j w_{ij}$. We use $w(x) = 1 - x/r$ for our experiments, where r is the search radius.

Comparison to Existing Methods We are not the first to propose hierarchical convolution-like operators for learning on point clouds. In this section we look at a number of other implementations and identify key differences.

Pointnet++ [15] and SpiderCNN [20] each use feature kernels which are non-linear with respect to the learned parameters. This means these methods have a large memory usage which increases as they create edge features from point features, before reducing those edge features back to point features.

Pointnet++ claims to use a ball neighborhood – and show results are improved using this over kNN. However, their implementation is based on a truncated kNNsearch with fixed k, meaning padding edges are created in sparse regions and meaningful edges are cropped out in dense regions. The cropping is partially offset by the use of max pooling over the neighborhood and IFP ordering, since the first k neighbors found are relatively spread out over the neighborhood. As discussed however, IFP is $\mathcal{O}(SS')$ in time, but removing this means results in the truncated ball search will no longer necessarily be evenly distributed. Also, the padding of sparse neighborhoods leads to an inefficient implementation, as edge features are computed despite never being used.

FlexConv [21] present a very similar derivation to our own. However, they implement Equation 5 with a custom GPU kernel that only supports kNN.

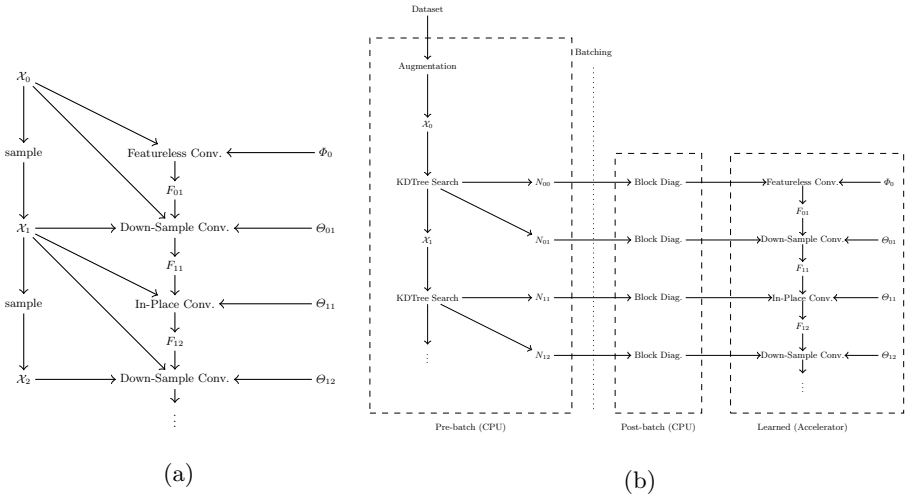
On the whole, we are unable to find any existing learned approaches that perform true ball searches, nor make any attempt to deal with the discontinuity inherent to kNN. We accept models are capable of learning robustness to such discontinuities, but feel enforcing it at the design stage warrants consideration.

Data Pipeline There are two aspects of the data processing that are critical to the efficient implementation of our point cloud convolution networks.

Neighborhood Preprocessing The neighborhood matrices $N^{(m)}$ are functions of relative point coordinates and the choice of unlearned basis functions – they do not depend on any learned parameters. This means they can be pre-computed, either online on CPUs as the previous batch utilizes available accelerators, or offline prior to training. In practice we only pre-compute the neighborhood indices and calculate the relative coordinates and basis functions on the accelerator. This additional computation on the accelerator(s) takes negligible time and reduces the amount of memory that needs to be stored, loaded and shipped.

Ragged Batching During the batching process, the uneven number of points in each cloud for each example can be concatenated, rather than padded to a uniform size, and sparse matrices block diagonalized. For environments where fixed-sized inputs are required, additional padding can occur at the *batch* level, rather than the individual example level, where variance in the average size will be smaller.

Unlike standard dataset preprocessing, our networks require network-specific preprocessing – preprocessing dependent on *e.g.* the size of the ball searches at each layer, the number of layers *etc.* To facilitate testing and rapid prototyping, we developed a meta-network module for creating separate pre- and post-batch preprocessing, while simultaneously building learned network stages based on a single conceptual network architecture. This is illustrated in Figure 2.



(a) (b)

Fig. 2: (a) Conceptual network vs (b) separate computation graphs.

3.2 Event Stream Convolutions

Event streams from cameras can be thought of as 3D point clouds in (x, y, t) space. However, only the most fundamental of physicists would consider space and time equivalent dimensions, and in practice their use cases are significantly different. For event cameras in particular,

- spatial coordinates of events are discrete and occur on a fixed size grid corresponding to the pixels of the camera;
- the time coordinate is effectively continuous and unbounded; and
- events come in time-sorted order.

We aim to formulate a model with the following requirements:

- *Intermediate results*: we would like our model to provide a stream of predictions, updated as more information comes in, rather than having to wait for the end of a sequence before making an inference.
- *Run indefinitely*: we would like to deploy these models in systems which can run for long periods of time. As such, our memory and computational requirements must be $\mathcal{O}(1)$ and $\mathcal{O}(E)$ respectively, with respect to the number of events.

Unfortunately, these requirements are difficult to enforce while making good use of modern deep learning frameworks and hardware accelerators. That said, just because we desire these properties in our end system does not mean we need them during training. By using convolutions with a domain of integration extending *backwards in time only* and using an exponentially decaying kernel, we can train using accelerators on sparse matrices and have an alternative deployment implementation which respects the above requirements.

Formally, we propose neighborhoods defined over small spatial neighborhood of size M_u – similar to image convolutions – extending backwards in time – with a kernel given by

$$g(u, \Delta t) = \sum_v \exp(-\lambda_{uv} \Delta t) \theta_{uv}, \tag{12}$$

where u corresponds to the pixel offset between events and v sums over some fixed temporal kernel size M_v, λ_{uv} is a learned temporal decay term enforced to be positive, θ_{uv} is a learned parameter and u extends over the spatial neighborhood. A temporal domain of integration extending backwards in time only ensures $\Delta t \geq 0$, hence we ensure the effects of events on features decay over time.

Dual Implementations For training, the kernel function of Equation 12 can be used in Equation 5 and reduced to a form similar to Equation 8, where $M = M_u M_v$. This can be implemented in the same way as our point cloud convolutions. Unfortunately, this requires us to construct the entire sparse matrix, removing any chance of getting intermediate results when they are relevant, and also breaks our $\mathcal{O}(1)$ memory constraint with respect to the number of events.

As such, we additionally propose a deployment implementation that updates features at pixels using exponential moving averages in response to events. As an input events come in, we decay the current values of the corresponding pixel by the time since it was last updated and add the new input features. When the features for an output event are required, the features of the pixels in the receptive field can be decayed and then transformed by $\Theta^{(uv)}$, and reduced like a standard image convolution. Formally, we initialize $\mathbf{z}_x^{(uv)} = \mathbf{0} \in \mathbb{R}^Q$ and $\tau_x = 0$ for all pixels x. For each input event (x, t) with features $\mathbf{f}$, we perform the following updates:

$$\mathbf{z}_x^{(uv)} \leftarrow \mathbf{f} + \exp(-\lambda_{uv}(t - \tau_x))\mathbf{z}_x^{(uv)} \tag{13}$$

$$\tau_x \leftarrow t. \tag{14}$$

Features $\mathbf{f}'$ for output event at (x', t') can thus be computed by

$$\mathbf{f}' = \sum_{u,v} \exp(-\lambda_{uv}(t' - \tau_{x'-u}))\mathbf{z}_{x'-u}^{(uv)\ T}\Theta^{(uv)}. \tag{15}$$

This requires $\mathcal{O}(M_u M_v Q)$ operations per input event, $\mathcal{O}(M_u M_v PQ)$ operations per output event and $\mathcal{O}(M_u M_v Q)$ space per pixel. Alternatively, the linear transform can be applied to $\mathbf{f}$ during the $\mathbf{z}_x^{(uv)}$ update (equivalent to up-sampling convolutions) for subtly different space and computational requirements. Either way, our requirements are satisfied.

Subsampling As with our point cloud formulation, we would like a hierarchical model with convolutions joining multiple streams with successively lower resolution and higher dimensional features. We propose using an unlearned leaky-integrate-and-fire (LIF) model due to the simplicity of the implementation and its prevalence in SNN literature [40].

LIF models transform input spike trains by tracking a theoretical voltage at each location or "neuron". These voltages exponentially decay over time, but are increased discontinuously by input events in some receptive field. If the voltage at a location exceeds a certain threshold, the voltage at that neuron is reset, and an output event is fired. SNNs generally learn the sensitivity of each output neuron to input spikes. We take a simpler approach, using a fixed voltage increase of $1/n$ as a result of an input spike, where n is the number of output neurons affected by the input event. Note we do not suggest this is optimal for our use case – particularly the unlearned nature of it – though we leave additional investigation of this idea to future work.

4 Experiments

We perform experiments on various classification tasks across point clouds and event streams. We provide a brief overview of network structures here. Model diagrams and technical details about the training procedure are provided in the supplementary material.

We investigate our point cloud operator in the context of ModelNet40 [41], a 40-class classification problem with 9840 training examples and 2468 testing examples. We use the first 1024 points provided by Pointnet++ [15] and use the same point dropout, random jittering and off-axis rotation, uniform scaling and shuffling data augmentation policies.

We construct two networks based loosely on Resnet [42]. Our larger model consists of an in-place convolution with 32 output channels, followed by 3 alternating down-sampling and in-place residual blocks, with the number of filters increasing by a factor of 2 in each down-sampling block. Our in-place ball radii start at 0.1125 and increase by a factor of 2 each level. Our down-sample radii are $\sqrt{2}$ larger than the previous in-place convolution. This results in sampled point clouds with roughly 25% of the original points on average, roughly 10

neighbors per in-place output point and 20 neighbors per down-sample output point. After our final in-place convolution block we use a single point-wise convolution to increase the number of filters by a factor of 4 before max pooling across the point dimension. We feed this into a single hidden layer classifier with 256 hidden units. All convolutions use monomial basis functions up to 2nd order. We use dropout, batch normalization and L2 regularization throughout. Our smaller model is similar, but skips the initial in-place convolution and has half the number of filters at each level. Both are trained using a batch size of 128 using Adam optimizer [43] and with the learning rate reduced by a factor of 5 after 20 epochs without an improvement to training accuracy.

For event streams, we consider 5 classification tasks – N-MNIST and N-Caltech101 from Orchard *et al.*[24], MNIST-DVS and CIFAR10-DVS from Serrano *et al.*[23]) and ASL-DVS from Bi *et al.*[12].

All our event models share the same general structure, with an initial 3x3 convolution with stride 2 followed by alternating in-place resnet/inception-inspired convolution blocks and down-sample convolutions (also 3x3 with stride 2), finishing with a final in-place block. We doubled the number of filters and the LIF decay time at each down sampling.

The result is multiple streams, with each event in each stream having its own features. The features of any event in any stream could be used as inputs to a classifier, but in order to compare to other work we choose to pool our streams by averaging over (x, y, t) voxels at our three lowest resolutions. For example, our CIFAR-10 model had streams with learned features at 64×64 down to 4×4. We voxelized the 16×16 stream to $16 \times 16 \times 4$, the 8×8 stream into an $8 \times 8 \times 2$ grid and the final stream into a $4 \times 4 \times 1$. Each voxel grid beyond the first receives inputs from the lower resolution voxel grid (via a strided $2 \times 2 \times 2$ voxel convolution), and from the average of the event stream. In this way, examples with relatively few events that result in zero events at the final stream still resulted in predictions (empty voxels are assigned the value $\mathbf{0}$). Hyperparameters associated with stream propagation (decay rate, spike threshold and reset potential) were hand-selected via an extremely crude search that aimed to achieve a modest number of events in the final stream for most examples. These hyperparameters, along with further details on training and data augmentation are provided in the supplementary material.

5 Results

5.1 Point Clouds

We begin by benchmarking our implementations of Equation 8. We implement the outer summation in two ways: a parallel implementation which unstacks the relevant tensors and computes matrix-vector products in parallel, and a map-reduce variant which accumulates intermediate values. Both are written entirely in the high-level python interface to Tensorflow 2.0.

We compare with the work of Groh *et al.*[21] who provide benchmarks for their own tensorflow implementation, as well as a custom CUDA implementation

that only supports kNN. Our implementations are written entirely in the high-level Tensorflow 2 python interface and can handle arbitrary neighborhoods. Computation time and memory requirements are shown in Table 2. Values do not include neighborhood calculations. Despite our implementation being more flexible, our forward pass is almost an order of magnitude faster, and our full training pass is sped up more than 60-fold. Our implementation does require more memory. We also see significant improvements by using Tensorflow's accelerated linear algebra just-in-time (JIT) compilation module, particularly in terms of memory usage.

	Time (ms)		GPU Mem (Mb)	
	Forward	Backward	Forward	Backward
TF [21]	1829	2738	34G	63G
Custom [21]	24.0	265.0	**8.4**	**8.7**
$(NF)\Theta$	2.9	5.1	57.3	105.0
$(NF)\Theta$-JIT	**2.7**	5.0	41.0	41.0
$N(F\Theta)$	2.9	4.3	56.0	56.2
$N(F\Theta)$-JIT	**2.7**	**4.1**	40.0	49.0

Table 2: Equation 8 implementations vs. FlexConv benchmarks on an Nvidia GTX-1080Ti. $M = 4$, $P = Q = 64$, $S = S' = 4096$, 9 neighbors and batch size of 8. Backward passes compute gradients w.r.t. learned parameters and input features (F and Θ). JIT rows correspond to just-in-time compiled implementations excluding compile time.

Next we look at training times and capacity of our model on the ModelNet40 classification task using 1024 input points. Table 3 shows performance at various possible batch sizes and training times for our standard model compared to various other methods. For fair comparison, we do not use XLA compilation.

Clearly our model runs significantly faster than those we compare to. Just as clear is the fact that our models which compute neighborhood information online are CPU-constrained. This preprocessing is not particularly slow – a modest 8-core desktop is capable of completing the 7 neighborhood searches and 3 rejection samplings associated with each example on our large model at over 800 Hz, which results in training that is still an order of magnitude faster than the closest competitor – but in the context of an accelerator-based training loop that runs at up to 3000 Hz this is a major bottleneck.

One might expect such a speed-up to come at the cost of inference accuracy. Top-1 accuracy is given in Table 4. We observe a slight drop in performance compared to recent state-of-the-art methods, though our large model is still competitive with well established methods like Pointnet++. Our small model performs distinctly worse, though still respectably.

Model	Batch Size	Epoch time (s)	
		Online	Offline
SpiderCNN [20]	24	196	-
Pointnet++ [15]	32	56	-
	64	56	-
PointCNN [19]	32	35	-
	64	33	-
	128	33	-
Ours (large)	32	12.9	6.80
	64	11.8	5.10
	128	11.2	4.13
	1024	12.4	3.60
	4096	11.5	**3.35**
Ours (small)	32	12.0	4.35
	64	11.4	2.83
	128	11.2	2.06
	1024	11.4	**1.38**
	4096	11.5	1.41
	9840	13.0	1.39

Table 3: Time to train 1 epoch of ModelNet40 classification on an Nvidia GTX-1080Ti. Online/offline refers to preprocessing.

Model	Reported/Best	Mean
Ours (small)	88.77	87.94
Pointnet [13]	89.20	88.65
KCNet [18]	91.00	89.62
DeepSets [14]	90.30	89.71
Pointnet++ [15]	90.70	90.14
Ours (large)	91.08	90.34
DGCNN [17]	92.20	91.55
PointCNN [19]	92.20	91.82
SO-Net [16]	**93.40**	**92.65**

Table 4: Top-1 instance accuracy on ModelNet40, sorted by mean of 10 runs according to Koguciuk *et al.*[44], for our large model with batch size 128. Reported/Best are those values reported by other papers, and the best of 10 runs for our models.

5.2 Event Camera Streams

Table 5 shows results for our method on the selected classification tasks. We see minor improvements over current state-of-the art methods on the straightforward MNIST variants, though acknowledge the questionable value of such minor improvements on datasets like these. We see a modest improvement on CIFAR-10, though perform relatively poorly on N-Caltech101. Our ASL-DVS model significantly out-performs the current state-of-the-art, with a 44% reduction in error rate. We attribute the greater success on this last dataset compared to others to the significantly larger number of examples available during training (∼80,000 vs ∼10,000).

Model	N-MNIST	MNIST-DVS	CIFAR-DVS	NCaltech101	ASL-DVS
HATS [26]	99.1	98.4	52.4	64.2	-
RG-CNN [12]	99.0	98.6	54.0	**65.7**	90.1
Ours	**99.2**	**99.1**	**56.6**	63.0	**94.6**

Table 5: Top-1 classification accuracy (%) for event stream classification tasks.

References

1. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C.J.C., Bottou, L., Weinberger, K.Q., eds.: Advances in Neural Information Processing Systems 25. Curran Associates, Inc. (2012) 1097–1105
2. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In: 2009 IEEE conference on computer vision and pattern recognition, Ieee (2009) 248–255
3. Erickson, B.J., Korfiatis, P., Akkus, Z., Kline, T.L.: Machine learning for medical imaging. Radiographics **37** (2017) 505–515
4. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
5. Elton, D.C., Boukouvalas, Z., Fuge, M.D., Chung, P.W.: Deep learning for molecular design-a review of the state of the art. Molecular Systems Design & Engineering (2019)
6. Pierson, H.A., Gashler, M.S.: Deep learning in robotics: a review of recent research. Advanced Robotics **31** (2017) 821–835
7. Grigorescu, S., Trasnea, B., Cocias, T., Macesanu, G.: A survey of deep learning techniques for autonomous driving. arXiv preprint arXiv:1910.07738 (2019)
8. Heng, L., Choi, B., Cui, Z., Geppert, M., Hu, S., Kuan, B., Liu, P., Nguyen, R., Yeo, Y.C., Geiger, A., et al.: Project autovision: Localization and 3d scene perception for an autonomous vehicle with a multi-camera system. In: 2019 International Conference on Robotics and Automation (ICRA), IEEE (2019) 4695–4702
9. Li, B., Zhang, T., Xia, T.: Vehicle detection from 3d lidar using fully convolutional network. arXiv preprint arXiv:1608.07916 (2016)
10. Gao, H., Cheng, B., Wang, J., Li, K., Zhao, J., Li, D.: Object classification using cnn-based fusion of vision and lidar in autonomous vehicle environment. IEEE Transactions on Industrial Informatics **14** (2018) 4224–4231
11. García-Martín, E., Rodrigues, C.F., Riley, G., Grahn, H.: Estimation of energy consumption in machine learning. Journal of Parallel and Distributed Computing **134** (2019) 75–88
12. Bi, Y., Chadha, A., Abbas, A., Bourtsoulatze, E., Andreopoulos, Y.: Graph-based spatial-temporal feature learning for neuromorphic vision sensing. arXiv preprint arXiv:1910.03579 (2019)
13. Qi, C.R., Su, H., Mo, K., Guibas, L.J.: Pointnet: Deep learning on point sets for 3d classification and segmentation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. (2017) 652–660
14. Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R.R., Smola, A.J.: Deep sets. In: Advances in neural information processing systems. (2017) 3391–3401
15. Qi, C.R., Yi, L., Su, H., Guibas, L.J.: Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In: Advances in Neural Information Processing Systems. (2017) 5099–5108
16. Li, J., Chen, B.M., Hee Lee, G.: So-net: Self-organizing network for point cloud analysis. In: Proceedings of the IEEE conference on computer vision and pattern recognition. (2018) 9397–9406
17. Wang, Y., Sun, Y., Liu, Z., Sarma, S.E., Bronstein, M.M., Solomon, J.M.: Dynamic graph cnn for learning on point clouds. ACM Transactions on Graphics (TOG) **38** (2019) 146

18. Shen, Y., Feng, C., Yang, Y., Tian, D.: Mining point cloud local structures by kernel correlation and graph pooling. In: Proceedings of the IEEE conference on computer vision and pattern recognition. (2018) 4548–4557
19. Li, Y., Bu, R., Sun, M., Wu, W., Di, X., Chen, B.: Pointcnn: Convolution on x-transformed points. In: Advances in Neural Information Processing Systems. (2018) 820–830
20. Xu, Y., Fan, T., Xu, M., Zeng, L., Qiao, Y.: Spidercnn: Deep learning on point sets with parameterized convolutional filters. CoRR **abs/1803.11527** (2018)
21. Groh, F., Wieschollek, P., Lensch, H.P.A.: Flex-convolution (deep learning beyond grid-worlds). CoRR **abs/1803.07289** (2018)
22. Posch, C., Matolin, D., Wohlgenannt, R.: A qvga 143 db dynamic range frame-free pwm image sensor with lossless pixel-level video compression and time-domain cds. IEEE Journal of Solid-State Circuits **46** (2010) 259–275
23. Serrano-Gotarredona, T., Linares-Barranco, B.: A 128 *times* 128 1.5% contrast sensitivity 0.9% fpn 3 μs latency 4 mw asynchronous frame-free dynamic vision sensor using transimpedance preamplifiers. IEEE Journal of Solid-State Circuits **48** (2013) 827–838
24. Orchard, G., Jayawant, A., Cohen, G.K., Thakor, N.: Converting static image datasets to spiking neuromorphic datasets using saccades. Frontiers in neuroscience **9** (2015) 437
25. Maqueda, A.I., Loquercio, A., Gallego, G., García, N., Scaramuzza, D.: Event-based vision meets deep learning on steering prediction for self-driving cars. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. (2018) 5419–5427
26. Sironi, A., Brambilla, M., Bourdis, N., Lagorce, X., Benosman, R.: Hats: Histograms of averaged time surfaces for robust event-based object classification. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. (2018) 1731–1740
27. Nguyen, A., Do, T.T., Caldwell, D.G., Tsagarakis, N.G.: Real-time 6dof pose relocalization for event cameras with stacked spatial lstm networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops. (2019) 0–0
28. Bohte, S.M., Kok, J.N., La Poutre, H.: Error-backpropagation in temporally encoded networks of spiking neurons. Neurocomputing **48** (2002) 17–37
29. Russell, A., Orchard, G., Dong, Y., Mihalas, Ş., Niebur, E., Tapson, J., Etienne-Cummings, R.: Optimization methods for spiking neurons and networks. IEEE transactions on neural networks **21** (2010) 1950–1962
30. Cao, Y., Chen, Y., Khosla, D.: Spiking deep convolutional neural networks for energy-efficient object recognition. International Journal of Computer Vision **113** (2015) 54–66
31. Lagorce, X., Orchard, G., Galluppi, F., Shi, B.E., Benosman, R.B.: Hots: a hierarchy of event-based time-surfaces for pattern recognition. IEEE transactions on pattern analysis and machine intelligence **39** (2016) 1346–1359
32. Liu, B., Wang, M., Foroosh, H., Tappen, M., Pensky, M.: Sparse convolutional neural networks. In: Proceedings of the IEEE conference on computer vision and pattern recognition. (2015) 806–814
33. Park, J., Li, S., Wen, W., Tang, P.T.P., Li, H., Chen, Y., Dubey, P.: Faster cnns with direct sparse convolutions and guided pruning. arXiv preprint arXiv:1608.01409 (2016)
34. Graham, B., van der Maaten, L.: Submanifold sparse convolutional networks. arXiv preprint arXiv:1706.01307 (2017)

35. Graham, B., Engelcke, M., Van Der Maaten, L.: 3d semantic segmentation with submanifold sparse convolutional networks. In: Proceedings of the IEEE conference on computer vision and pattern recognition. (2018) 9224–9232
36. Choy, C., Gwak, J., Savarese, S.: 4d spatio-temporal convnets: Minkowski convolutional neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. (2019) 3075–3084
37. Jampani, V., Kiefel, M., Gehler, P.V.: Learning sparse high dimensional filters: Image filtering, dense crfs and bilateral neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. (2016) 4452–4461
38. Su, H., Jampani, V., Sun, D., Maji, S., Kalogerakis, E., Yang, M.H., Kautz, J.: Splatnet: Sparse lattice networks for point cloud processing. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. (2018) 2530–2539
39. Boulch, A.: Convpoint: continuous convolutions for point cloud processing. Computers & Graphics (2020)
40. Koch, C., Segev, I., et al.: Methods in neuronal modeling: from ions to networks. MIT press (1998)
41. Wu, Z., Song, S., Khosla, A., Yu, F., Zhang, L., Tang, X., Xiao, J.: 3d shapenets: A deep representation for volumetric shapes. In: Proceedings of the IEEE conference on computer vision and pattern recognition. (2015) 1912–1920
42. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2016) 770–778
43. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
44. Koguciuk, D., Chechliński, Ł., El-Gaaly, T.: 3d object recognition with ensemble learning—a study of point cloud-based deep learning models. In: International Symposium on Visual Computing, Springer (2019) 100–114