Quadtree Convolutional Neural Networks

Pradeep Kumar Jayaraman, Jianhan Mei, Jianfei Cai, and Jianmin Zheng

Nanyang Technological University, Singapore
{pradeepkj@, jianhan001@e., ASJFCai@, ASJMZheng@}ntu.edu.sg

Abstract. This paper presents a Quadtree Convolutional Neural Network (QCNN) for efficiently learning from image datasets representing sparse data such as handwriting, pen strokes, freehand sketches, etc. Instead of storing the sparse sketches in regular dense tensors, our method decomposes and represents the image as a linear quadtree that is only refined in the non-empty portions of the image. The actual image data corresponding to non-zero pixels is stored in the finest nodes of the quadtree. Convolution and pooling operations are restricted to the sparse pixels, leading to better efficiency in computation time as well as memory usage. Specifically, the computational and memory costs in QCNN grow linearly in the number of non-zero pixels, as opposed to traditional CNNs where the costs are quadratic in the number of pixels. This enables QCNN to learn from sparse images much faster and process high resolution images without the memory constraints faced by traditional CNNs. We study QCNN on four sparse image datasets for sketch classification and simplification tasks. The results show that QCNN can obtain comparable accuracy with large reduction in computational and memory costs.

Keywords: quadtree, neural network, sparse convolution

1 Introduction

Convolutional neural networks (CNNs) are a powerful and popular method for various tasks involving the analysis of images, videos and three-dimensional objects. Most of the real world image data such as natural photographs or volumetric meshes can be represented as dense tensors, and indeed, conventional CNNs were originally proposed to optimally learn features from such data by local weight connections and parameter sharing. On the other hand, it is observed that some datasets are sparse in nature. For example, images representing freeform 2D sketches and handwriting only consist of a set of one-dimensional lines occupying a sparse subset of the 2D image plane, while point clouds or triangle meshes are only defined in a small subset of the 3D space. Unfortunately, most of the traditional CNN architectures, particularly for images, are unable to exploit the sparsity of such data, and learning from such datasets is unnecessarily inefficient in both training time and memory consumption. This is particularly of concern with the rise of deep networks that are being increasingly employed to various high resolution sparse images in applications such as sketch simplification [1], as well as to resource-scarce mobile or embedded devices.

In the case of 3D data, convolutional neural networks were originally designed by voxelizing the mesh into dense 3D tensors [2]. Due to memory and computational constraints, however, this approach does not scale to high resolutions. To alleviate this, recent works such as OctNets [3], O-CNN [4], and OGN [5] decompose the 3D meshes hierarchically into octrees and adapt CNN operations to consider the special octree structure.

Inspired by such 3D works, we present in this paper a quadtree convolutional neural network (QCNN) for efficiently learning from sparse 2D image datasets. While those 3D networks were designed to deal with 3D shapes represented by meshes or point clouds, we target general sparse images which usually have more arbitrary structure or topology. This will enable the developed method to have wide applications, especially on mobile devices where the computing power and memory are limited. Our main idea is to decompose sparse images into quadtrees, store the non-zero image pixels in the finest nodes, and design special data representation that takes the features of CPU and GPU into consideration. The computation effort will be concentrated on the areas of interest, which avoids the storage of empty pixels that do not provide meaningful information and thus reduces the memory consumption. We start with the finest level of the quadtree and perform convolutions on these nodes to compute the features, followed by pooling which downsamples the features and propagates them to the next coarser quadtree level. This operation can be stacked multiple times before finally obtaining the network output with respect to some predefined target and loss function.

Our approach has several advantages in terms of efficiency. First, since we only store non-zero pixels of the image in the bottom most level of the sparse quadtree, the storage and computational requirements are linear in the number of non-zero pixels and completely independent of image resolution. Second, it is well known that modern CPUs and GPUs are highly efficient in processing data that are contiguous in memory. Hence we use a linear quadtree representation where each level of the quadtree is stored as a linear 1D array by indexing the quadtree nodes with space-filling z-order curves. Third, we adapt CNN operations in the quadtree by considering the special data representation. Convolution that requires neighborhood access for each quadtree node in the same depth is achieved via an efficient look-up table based scheme using the Moser-de Bruijn sequence, and pooling is as simple as assigning the maximum/average of every four children nodes to their parent nodes. We demonstrate the efficiency of QCNN in terms of computational effort on several sparse image datasets for classification and sketch simplification.

2 Related Work

2.1 Hierarchical Data Representation

The quadtree [6] and octree [7] are hierarchical representations of 2D and 3D spatial data, respectively, and generalizations of the binary tree. They have been extensively used in various graphics and image processing applications such as

collision detection, ray tracing, and level-of-details [8–10]. It is common to implement quadtrees and octrees using pointers. However, for representing data hierarchically for CNN training purposes, this is infeasible since CPUs and GPUs are efficient in processing contiguous array data. Linear quadtrees or octrees [11], where 2D/3D node indices in each level of the tree are converted to 1D indices using space-filling curves, are more relevant to our application.

2.2 Sparse Convolutional Neural Networks

Using sparsity can result in higher resolution inputs to be processed efficiently. However, there are only a few network architectures that exploit sparsity. Initially, CNNs were employed to process 3D data by voxelizing the meshes into 3D dense volumetric tensors [2]. Since this representation has a high computation and memory cost, the input resolution had to be restricted to around 30^3. Graham proposed a sparse version of the CNN for 2D image [12] and 3D voxel [13] data that only performs convolutions on non-zero sites and their neighbors within the receptive field of the kernel. Nevertheless, the approach becomes inefficient when a large number of convolution layers are placed in between the pooling layers since the feature map dilates after each convolution. The feature dilation problem was recently handled by Graham and Maaten [14] by restricting convolutions only on the non-zero sites. Their works require additional book-keeping for indexing the non-zero pixels for each layer's output, as well as efficient hash table implementations.

Quadtree/octree structures on the other hand can be computed in one shot and clearly define the structure of the data beforehand, independently of the convolution or pooling parameters in the network. Additionally, they can be linearly represented as a simple contiguous array, thanks to their regular structure. Moreover, simply conforming the feature maps to the quadtree/octree structure is sufficient to significantly prevent feature dilation. To support high resolution 3D data, Riegler et al. [15] combined octree and a grid structure, and limited CNN operations to the interior volume of 3D shapes. While this is efficient compared to using dense voxels, storing the interior volume of 3D surface data is still wasteful. Wang et al. [4] only considered the surface voxels of the 3D data in the octree representation and drastically improved memory and computational costs in performing CNN operations. Similar to octrees, our work introduces the quadtree structure for efficiently learning from sparse image data.

3 Quadtree Convolution Neural Network

3.1 Motivation

Consider a general scenario where a dense n-dimensional tensor used to represent some input that is to be fed into a convolutional neural network. This tensor could represent grayscale images ($n = 2$), color images ($n = 3$), voxels from 3D points clouds or surfaces ($n = 3$), etc. Sparsity arises in an n-dimensional tensor

whenever it is used to represent a lower-dimensional ($< n$) manifold. Examples include a set of freehand pen strokes that are 1-manifolds in 2D grayscale images and triangle meshes that are 2-manifolds stored in 3D volumes. Even if the object of interest only occupies a small portion of the space it resides in, the storage costs of a representing such objects with a dense tensor grows in the order of n with increasing resolution, as does the computational cost of applying convolutions to extract feature maps. For example, in this paper where we mainly consider sparse grayscale image data ($n = 2$), an $N \times N$ image requires a storage cost of N^2, and convolving $M \times M$ kernels to compute C feature maps requires M^2N^2C multiply-accumulate (MACC) operations (assuming unit stride), both of which are of quadratic complexity in the total number of pixels.

By representing the image as a quadtree which is only subdivided when non-zero pixels exist in a quadrant, the non-zero pixels (without the loss of generality) in the input image correspond to the nodes in the finest quadtree level. Hence, the storage requirement of the image data is roughly N_{nz} denoting the number of non-zero pixels (where $N_{nz} \ll N^2$). If we restrict the convolutions to these non-zero pixels, then we need $M^2N_{nz}C$ MACC operations. This process is of linear complexity in the number of pixels stored in the quadtree level and independent of the image resolution N.

There are several advantages of using a quadtree to hierarchically represent the image data for convolutional neural networks. First, the quadtree can be efficiently computed once for each image, and its structure then remains fixed throughout the forward and backward passes and requires no further bookkeeping. Its structure defines the locations of the non-zero pixels hierarchically and the nodes that are non-empty in each level. Convolutions are performed on the pixels that correspond to the nodes in the bottommost level of the quadtree, resulting in feature maps that fit in the same level. By simply restricting computations to the sparse quadtree nodes, we can ensure that feature maps do not dilate in the deeper layers of the network even when repeated convolution layers are stacked, hence retaining the sparse nature of the input. Second, since the quadtree structure is by definition hierarchical, downsampling and upsampling features can be performed easily and efficiently. Pooling downsamples the feature map such that it can be stored in the previous level of the quadtree, and is carried out by assigning the maximum or average of the children nodes at the current level into their parent node in the previous coarser level. Upsampling can be performed similarly by traversing the quadtree in the opposite direction.

3.2 Representing Images as Linear Quadtrees

We use a linear quadtree to decompose the input image, where nodes at each level are stored in a contiguous array for convenient and efficient processing in both CPU and GPU, as opposed to a pointer based quadtree. An image of dimension $2^\ell \times 2^\ell$ can be decomposed into an ℓ-level quadtree. Each of the nodes at level $l \in [1, \ldots, \ell]$ can be represented as a list of 1D indices. A common strategy to linearize indices is the interleaved bit representation. For example, given a 2D index, say $(x = 4, y = 5)$ of a quadtree node from level 3, which is $(100_2, 101_2)$ in

binary, the linear quadtree index is given by interleaving each of the binary digits corresponding to y and x alternatively, yielding $110010_2 = 50$. This linearization has two advantages: First, it is locality preserving as opposed to row-column indexing and ensures higher cache hits when looking up neighbors during CNN operations since they are mapped to nearby locations in 1D. Second, this indexing maps every four quadtree nodes sequentially in 1D memory, which leads to easy and efficient implementations of downsampling/upsampling operations.

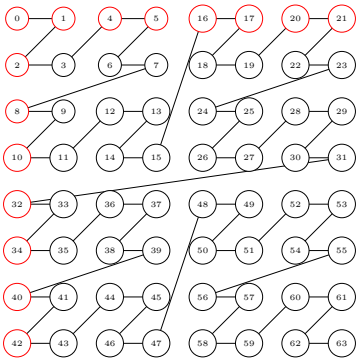


Fig. 1: Z-order indexing: 2D quadtree nodes (level $l = 3$ in this example) are linearized into a 1D array using Z-order curves as shown by the index values in circles. The 1D indices along the topmost row (colored red) are a Moser-de Bruijn sequence $(0, 4^0, 4^1, 4^0 + 4^1, \ldots)$, and those along the leftmost column are simply the same sequence scaled by 2.

Note that the interleaved coordinate representation is a space filling z-order curve of order l which maps 2D coordinates to 1D indices. One can observe from the z-order curves that their path follows a regular sequence. The top row highlighted in red in Fig. 1, is a Moser-de Bruijn sequence in which each number is a sum of unique powers of 4. The left column is the same sequence scaled by 2. We generate a 1D lookup table for the sequence in the top row $t : \mathbb{Z}_{\geq} \to \mathbb{Z}_{\geq}$ defined as: $t(0) = 0$, and $t(i) = (t(i-1) + 0\text{xaaaaaaab})$ `&` $0\text{x}55555555$, assuming that the quadtree node indices are represented with 32-bit unsigned integers. From this, the z-order index can be obtained as $z(x, y) = t(x) \,|\, (t(y) \ll 1)$, where `|` and `<<` are the bitwise or and left shift operators, respectively. For example, $z(4, 2) = t(4) \,|\, (t(2) \ll 1) = 16 \,|\, (4 \ll 1) = 24$. This lookup table is always generated of size 2^ℓ, which denotes the width or height of the quadtree in the maximum depth, and reused for all the computations including those in the coarser levels.

3.3 Quadtree CNN Operations

Data structure To facilitate CNN operations such as convolution and pooling on the quadtree, we employ a custom data structure that is different from

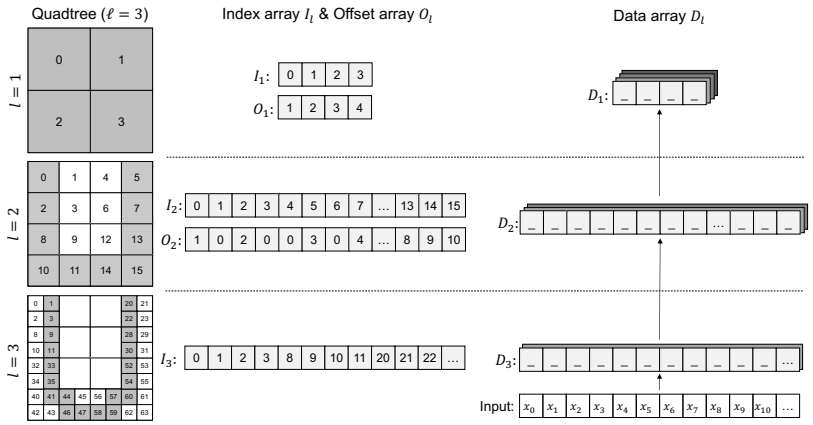


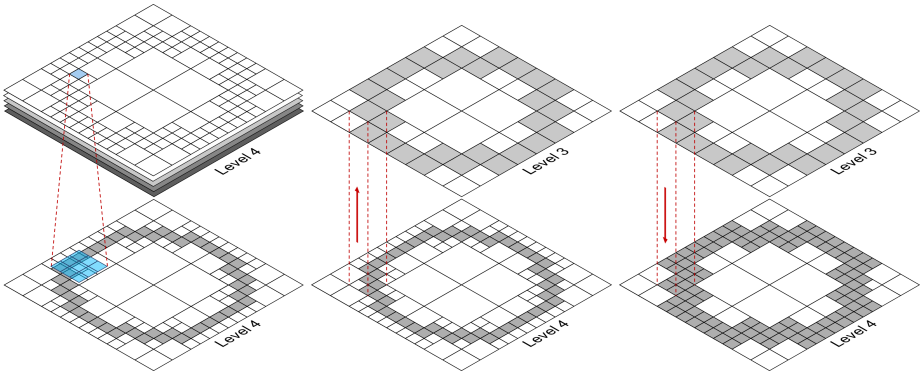
Fig. 2: Data structure for representing the image as a linear quadtree to support CNN operations. Left: quadtree generated for a U-shape image (numbers represent z-order indices), where gray nodes contain non-zero pixels in their bounds, while white nodes contain only zero pixels. Gray nodes are subdivided in the next level. Middle: Index array contains the corresponding z-order indices; offset array contains a monotonic sequence starting from 1 for gray nodes, white nodes are set to 0. Right: Data array holding the feature maps in each quadtree level.

commonly employed tensors in convolutional neural networks. This is necessary since unlike traditional grayscale images which store pixel data in a single 2D array, the linear quadtree stores quadtree node indices with non-zero pixels as a hierarchy of 1D arrays, and the pixel values themselves as a single 1D array corresponding to the deepest quadtree level. Moreover, since we only subdivide the non-empty quadtree nodes, we store an additional array, similar to O-CNN [4], to record the parent-child relationship (see Fig. 2):

Index array I: stores the z-order indices of all the nodes in the quadtree level-wise. We denote by $I_l[i]$ the index of a node i at level l. It is mainly used to lookup the indices of non-zero pixels and restrict convolutions on these nodes to extract features efficiently.

Offset array O: stores a monotonic sequence of integers starting from 1 to mark nodes that are to be subdivided (i.e., gray nodes in Fig. 2). If a node i is not subdivided (white nodes in Fig. 2), then its corresponding value is set to 0. The offset array is of same size as the index array, i.e., one value for each node, and we denote by $O_l[i]$ the offset of a node i at level l.

We use O_l for pooling features from children nodes at level $l+1$ to parent nodes at level l and upsampling features from parent nodes at level l to children nodes at level $l+1$. For example, $O_2[5] = 3$ means that the 3^{rd} set of quadruples (nodes 20–23 in I_3) in level 3, are the children of node 5 in I_2 in level 2. The index and offset arrays are generated once from the input image, and remain fixed afterwards throughout training/testing.



(a) (b) (c)

Fig. 3: Illustration of CNN operations on quadtrees. (a) Convolutions are restricted on the sparse quadtree nodes, and produce feature maps that fit in the same quadtree level. (b) Pooling downsamples the feature maps such that it fits in the previous quadtree level. (c) Upsampling resizes the feature map such that it fits in the next quadtree level.

Data array D: contains the feature maps at each level of the quadtree and is initially only available for the deepest quadtree level ℓ after performing convolution. It contains the values corresponding to the nodes indexed by I_ℓ in an array of dimensions $d \times e$ where d in the number of channels, and e is the number of stored quadtree nodes. The data array corresponding to other quadtree levels are eventually generated by the pooling operation which downsamples the feature maps.

Minibatch representation When using dense tensors, minibatches are created by concatenating a set of images along one of the axes; this operation requires all axes in the tensor which are smaller than the minibatch axis to agree in dimensions. Since quadtree structures vary among different images, we update the index arrays I_l of each of the B quadtrees in a minibatch by adding $4^l b$, where $b \in [0 \,..\, B-1]$. We then concatenate them together level-wise to form a single larger quadtree representing the entire batch. From this, individual quadtrees can be identified by computing $b = \lfloor (I_l[i] \;/\; 4^l) \rfloor$, while the local indices are given by $(I_l[i] \bmod 4^l)$. Similarly, the offset arrays O_l of each quadtree are updated to form a monotonic sequence throughout the batch and concatenated level-wise.

Having defined the data structure, we now discuss the adaptation of various CNN operations from images to quadtrees.

Convolution The convolution is the most important and expensive operation in convolutional neural networks. We implement convolutions on the quadtree as a single matrix multiplication[16] which can be carried out by highly optimized

matrix multiplication libraries on the CPU and GPU. The coefficients of the filters are arranged row-wise in a $c \times df^2$ matrix, and the pixels in the receptive field around each of the e quadtree nodes $D_l^{\text{in}}[\cdot,\cdot]$ in each input channel are arranged column-wise in a $df^2 \times e$ matrix, where d is the number of input channels, c is the number of output channels, f is the filter size, and e is the number of quadtree nodes in level l:

$$D_l^{\text{out}} := \begin{bmatrix} w_0^0 & w_1^0 & w_2^0 & \cdots \\ w_0^1 & w_1^1 & w_2^1 & \cdots \\ \vdots & \vdots & \vdots & \vdots \\ w_0^{c-1} & w_1^{c-1} & w_2^{c-1} & \cdots \end{bmatrix} \begin{bmatrix} \vdots & \vdots & \cdots & \vdots \\ D_l^{\text{in}}[0,0] & D_l^{\text{in}}[0,1] & \cdots & D_l^{\text{in}}[d-1,0] \\ \vdots & \vdots & \vdots & \vdots \\ D_l^{\text{in}}[1,0] & D_l^{\text{in}}[1,1] & \cdots & D_l^{\text{in}}[d-1,1] \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Here, the superscript in w_*^* runs from $[0 \,..\, c-1]$ indexing the number of filters, while the subscript runs from $[0 \,..\, df^2 - 1]$. This is different from traditional CNNs in two ways: first, only the non-zero quadtree nodes participate in convolutions, and second, the neighborhood lookup is different since the quadtree nodes are linearized into a 1D array in z-order. In detail, a node at index z in 1D can be deinterleaved once to obtain the 2D index (x, y). From this, it is straightforward to compute the neighbors using the lookup table $t(\cdot)$ proposed in Section 3.2 in constant time. If the neighbor index is present in the index array I_l, then its pixel value from D_l is assigned to the corresponding coefficient in the matrix above; otherwise it is set to 0 assuming a black background. Additionally, since the quadtree structure is fixed for each input sample, we can precompute the neighbors once and reuse them for convolutional operations throughout the network for even better efficiency.

We begin applying the convolution operation on the data D_ℓ stored in the finest quadtree nodes at level ℓ to obtain c output feature maps (see Fig. 3a) which fit in the same quadtree level if a unit stride is used. It is not possible to use arbitrary strides in QCNN since the output would then not conform to the quadtree structure. However, strides which are powers of 2 can be supported—for example, convolving the input data which resides at level l with a stride 2^s, where $s \in [0 \,..\, l)$, will result in an output that will fit in level $l - s$. We only use unit stride in all our experiments and leave downsampling to the pooling layers.

Pooling A common and important operation in convolutional neural networks is pooling, used to downsample the feature maps and aggregate information as we go deeper into the network. As demonstrated by Stringenberg et al. [17], pooling can be achieved by convolving with non-unit strides without any loss of accuracy. However, since the pooling operation is generally more efficient than convolutions, we implement it as follows. Pooling in QCNNs are particularly simple—we only need to assign the maximum (or average) of the 4 children nodes in level $l + 1$ to their corresponding parent node at level l, see Fig. 3b. This is easy to implement since the quadtree nodes are linearized in z-order, and

all 4 children nodes corresponding to a parent are stored in succession:

$$D_l[i] := \text{pool}(\{D_{l+1}[4(O_l[i]-1)+j] \mid j \in [0 \,..\, 3]\})$$

where the pool$(\cdot)$ function computes the maximum or average of the set.

Upsampling Another common operation in convolutional neural networks is upsampling the feature maps, which can be used for visualization [18] or as part of autoencoder-like, or U-shaped network architectures [19] where features are concisely encoded using pooling and decoded using deconvolution, unpooling or upsampling. In this work, we implement upsampling by resizing the feature maps to a higher resolution, which is followed by a learnable convolution layer. Upsampling in QCNN is performed by traversing the quadtree in the opposite direction compared to pooling, i.e., we assign the value of the parent node in level $l-1$ to all 4 children in level l:

$$D_l[4(O_{l-1}[i]-1)+j] := D_{l-1}[i],\ j \in [0 \,..\, 3]$$

which roughly corresponds to nearest neighbor interpolation, see Fig. 3c.

With these fundamental operations defined on the quadtree, it is straightforward to compose them to design commonly used CNN architectures.

4 Experiments

We demonstrate the efficiency and versatility of our QCNN by conducting experiments on sparse image datasets for supervised classification and sketch simplification. We study the performance as well as training behavior of QCNNs compared to CNNs and show that QCNNs are well-behaved and converge to results achieved with traditional CNNs with much less computation time and memory. Our implementation is in C++ and CUDA (built upon the Caffe [20] framework), and runs on an NVIDIA GTX 1080 GPU with 8GB memory.

For brevity, in the following we denote a convolution unit by $C_l(c)$ which is composed of: (1) a quadtree convolutional layer with 3×3 filters that accepts data D_l^{in} from a quadtree of level l as input and outputs c feature maps D_l^{out} fitting in the same level, (2) a batch normalization layer that normalizes the minibatch using the mean and standard deviation computed from elements in D_l^{out}, and (3) a rectified linear unit activation function [21]: $\text{ReLU}(x) = \max(0, x)$. We denote by P_l a quadtree max pooling layer that downsamples the feature maps from level l to $l-1$ and by U_l a quadtree upsampling layer that resizes feature maps from level l to $l+1$.

After completing all the quadtree convolutional and pooling operations in the network, we apply a quadtree padding operation denoted as "pad", to convert the sparse quadtree based feature maps into dense ones by zero padding the empty quadtree nodes and reshaping the quadtree minibatch of size B with d channels into a 4D dense tensor of dimensions $B \times d \times 2^l \times 2^l$. This is necessary to align the features computed for different images before feeding them to the fully-connected layers, since their quadtree structures are different.

4.1 Classification

We train traditional CNNs and our quadtree CNNs with similar network architectures on four sparse image datasets. Note that this experiment is mainly to study the behaviour of QCNN compared to traditional CNN and is not tuned to obtain the best accuracy.

MNIST is a popular dataset of numeric digits [0–9] consisting of grayscale images of size 28×28 that are split into $60,000$ training and $10,000$ test images. We zero pad each image to 32×32, and decompose them into quadtrees of level 5. The network structure is defined as:

$$\begin{aligned}\text{input} \rightarrow C_5(32) \rightarrow C_5(32) \rightarrow P_5 \rightarrow C_4(64) \rightarrow C_4(64) \rightarrow P_4 \rightarrow \text{pad} \rightarrow \text{FC}(1024) \\ \rightarrow \text{ReLU} \rightarrow \text{Dropout}(0.5) \rightarrow \text{FC}(K) \rightarrow \sigma,\end{aligned}$$

where C_*, P_* are the quadtree based convolutional units and pooling operations, respectively, defined earlier, pad is the quadtree padding operation to align features from different inputs, Dropout [22] with a rate of 0.5 is used to prevent overfitting, FC(n) is a fully-connected layer with an n-element vector as output, and σ is the softmax function that normalizes each element in its input into the range $[0, 1]$ such that the result is a discrete probability distribution, defined as $\sigma(x_j) = \frac{\exp x_j}{\sum_{k=1}^{n} \exp x_k}, j \in [1 \,..\, n]$, and K corresponds to the total number of classes in the dataset (10 in this case).

EMNIST Balanced [23] extends the classic MNIST dataset with more samples including alphabets, and contains $112,800$ training and $18,800$ test images of size 28×28. The 26 upper and lower case alphabets ([A–Z], [a–z]), and 10 digits ([0–9]) are combined into a total of $K = 47$ balanced classes. We zero pad the images as before and decompose them into quadtrees of level 5. We train them on a network that is defined exactly as in the previous case.

CASIA-HWDB1.1 [24] is a huge database of more than a million handwritten Chinese character sample images representing $3,866$ classes. We experiment with a 200-class subset of the dataset comprising $48,020$ training and $11,947$ test images. Since the images are of varying dimensions, we rescale them into 64×64, and decompose them into quadtrees of level 6. The network that we use for training is:

$$\begin{aligned}\text{input} \rightarrow C_6(64) \rightarrow P_6 \rightarrow C_5(128) \rightarrow P_5 \rightarrow C_4(256) \rightarrow P_4 \rightarrow C_3(512) \rightarrow P_3 \\ \rightarrow \text{pad} \rightarrow \text{FC}(1024) \rightarrow \text{Dropout}(0.5) \rightarrow \text{FC}(200) \rightarrow \sigma.\end{aligned}$$

TU-Berlin Sketch Dataset [25] contains $20,000$ images of freehand sketches drawn by non-experts, with 80 images in each of the $K = 250$ classes, in raster and vector formats. We split the dataset into $18,000$ training and 2000 test images, and resize each image into dimension 128×128. We then decompose the

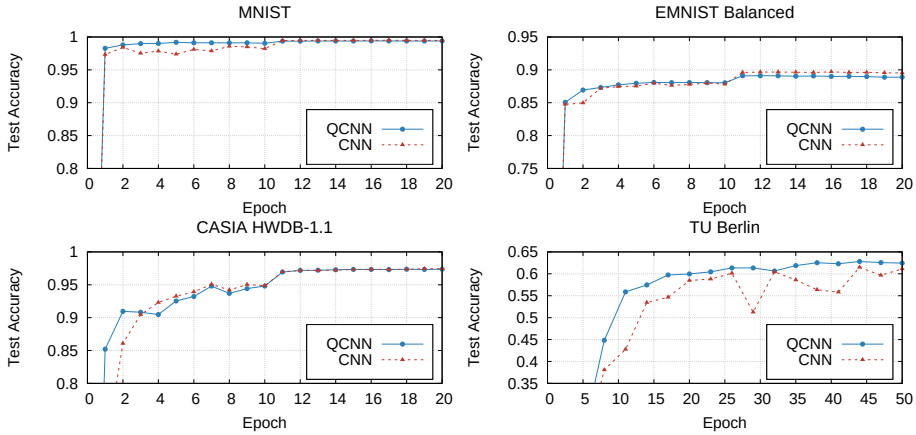


Fig. 4: Comparison of mean test accuracy (computed from 5 runs) of QCNN and traditional CNN classifiers progressively during training on various datasets.

images into quadtrees of level 7. The network for training is defined as:

$$\begin{aligned}\text{input} \to C_7(32) \to C_7(32) \to C_7(32) \to P_7 \to C_6(64) \to C_6(64) \to P_6 \\ \to C_5(128) \to \ P_5 \to C_4(256) \to P_4 \to C_3(512) \to P_3 \to C_2(1024) \\ \to \text{pad} \to \text{Dropout}(0.5) \to \text{FC}(250) \to \sigma.\end{aligned}$$

All these datasets are particularly suitable for our QCNN since the images are sparse in nature. We use the standard cross entropy loss for classification and use stochastic gradient descent for optimizing the first three networks with a learning rate 0.05, decayed by a factor of 10 after every 10 epochs, for 20 epochs, and ADADELTA optimization method [26] for the last network for 50 epochs (which provided better results compared to SGD). Weights are initialized using Glorot et al.'s technique [27], and regularized by a decay factor of 5×10^{-4}, while biases are initialized to zero. We do not perform any data augmentation for simplicity.

The computational statistics of the classification experiments are summarized in Table 1. We compare our QCNN results with those obtained by a traditional CNN with the same network architecture, initial weights, and hyperparameters. It can be seen that QCNNs are highly efficient in terms of computational effort (represented in column 3 as the number of multiply-accumulate operations in the convolutional layers) while yielding similar test accuracies (column 2). In practice, we observed that deeper networks train faster, for example, the TU-Berlin QCNN took one-third the training time compared to traditional CNN. To study the behaviour of QCNN throughout the training phase, we plot the learning curves in Fig. 4 comparing the test accuracy after each epoch of training. It is apparent that QCNNs closely follow CNNs in terms of accuracy on all the datasets throughout the training phase, while being computationally efficient.

Table 1: Computational statistics for classification comparing QCNNs and CNNs based on mean accuracy and standard deviation at the end of training from 5 runs, and average multiply-accumulate operations per sample (during forward pass) in the convolutional layers.

Dataset	Non-zero	Accuracy (%)		#MACC ($\times 10^6$)	
	pixels (%)	CNN	QCNN	CNN	QCNN
MNIST	32.82	99.48($\pm$0.03)	99.39($\pm$0.02)	18.36	6.1
EMNIST Balanced	44.15	89.53($\pm$0.09)	88.89($\pm$0.18)	18.36	9.03
CASIA-HWDB1.1	19.06	97.44($\pm$0.12)	97.36($\pm$0.08)	229.34	136.7
TU-Berlin	4.53	61.13($\pm$2.04)	62.44($\pm$1.16)	837.53	257.07

4.2 Sketch Simplification

We next study QCNN for the task of sketch simplification [1]. This application is again suitable for our method since relatively high resolution sparse images are trained on deep convolutional neural networks.

We adapt the TU-Berlin sketch dataset for this experiment by synthesizing an inverse dataset consisting of sketchy rough line drawings. We utilize the SVG version of TU-Berlin dataset where each file represents a clean sketch drawing as a collection of paths in Bézier form. We duplicate each path 3 times and apply random affine transformations where the rotation angle and translation are drawn from Gaussian distributions with zero mean and standard deviations of 1.5° and 2, respectively. We repeat this for all SVG files in the dataset and rasterize them while setting the stroke width of the paths to 1px.

Next, we decompose all the rough sketches into quadtrees and represent the corresponding clean sketches using the same quadtree structure so that both are directly comparable during training. We define an encoding-decoding QCNN similar to Simo-Serra et al. [1]:

$$\begin{aligned}
&\text{sketchy} \to C_\ell(48) \to C_\ell(48) \to P_\ell \to C_{\ell-1}(128) \to C_{\ell-1}(128) \to P_{\ell-1} \\
&\to C_{\ell-2}(256) \to C_{\ell-2}(256) \to C_{\ell-2}(256) \to P_{\ell-2} \to C_{\ell-3}(256) \\
&\to C_{\ell-3}(512) \to C_{\ell-3}(1024) \to C_{\ell-3}(1024) \to C_{\ell-3}(1024) \to C_{\ell-3}(1024) \\
&\to C_{\ell-3}(512) \to C_{\ell-3}(256) \to U_{\ell-3} \to C_{\ell-2}(256) \to C_{\ell-2}(256) \\
&\to C_{\ell-2}(128) \to U_{\ell-2} \to C_{\ell-1}(128) \to C_{\ell-1}(128) \to C_{\ell-1}(48) \\
&\to U_{\ell-1} \to C_\ell(48) \to C_\ell(24) \to \text{conv}(1) \to \text{sigmoid} \to \text{clean image}
\end{aligned}$$

We introduce skip connections between the input of each pooling layer to the output of each corresponding upsampling layer in the same level to speed up the convergence. Note that we did not tune the network architecture or dataset to obtain best results, but rather study the performance and training behaviour. We train the network for 20 epochs with the mean-squared error loss between the rough and clean sketch data stored in the quadtrees using the ADADELTA [26] optimization method. As before, we also train a traditional CNN similarly for

this task to compare the results as well as computational and memory usage. As shown in Fig. 5, QCNN can obtain comparable simplified sketches, while

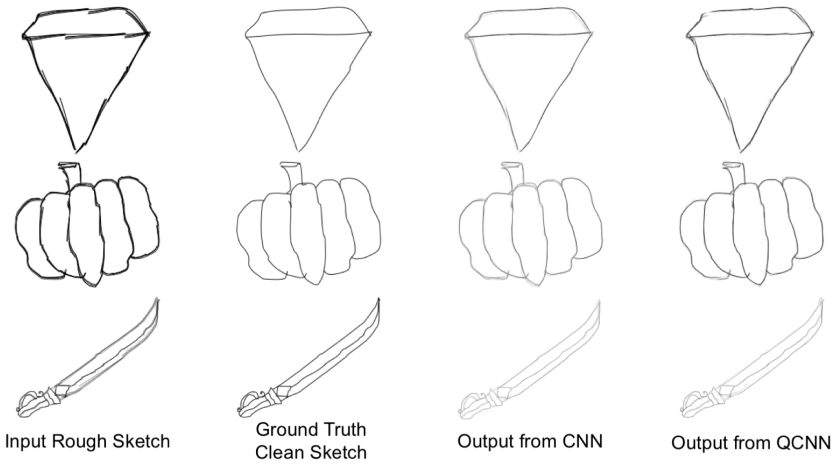


Fig. 5: Sketch simplification results obtained with traditional CNN and QCNN trained using the same network architecture.

greatly reducing computation and memory use, see Table 2. Training QCNN in our setup took only around one-fourth the time compared to CNN. We used images of dimension 256×256 ($\ell = 8$) for this experiment. However, since sketch simplification typically involves high-resolution images leading to extended training time, we also compute and provide the computation and memory usage for higher resolutions to illustrate the drastically increasing complexity, see second and third rows in Table 2.

To study the learning behaviour of QCNN, we visualize the evolution of the learning process by retrieving the weights of the models throughout the training phase and visualizing the simplified results, as shown in Fig. 6. It is apparent that the learning behaviour of QCNN is stable and quite similar to traditional CNN for this task which involves a deep network, while being highly computation and memory efficient.

5 Conclusion

We have presented a quadtree convolutional neural network (QCNN) that can efficiently learn from sparse image datasets. Thanks to the quadtree-based representation that decomposes the image only in the presence of non-zero pixels, storing and computing feature maps is of linear complexity in the number of non-zero pixels and independent of image resolution. QCNNs are applicable in a wide range of applications involving sparse images. Particularly, we have

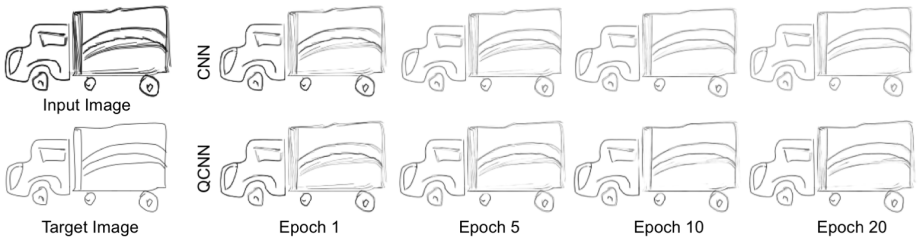


Fig. 6: Visualization of learning process for sketch simplification evolving throughout training.

Table 2: Computational statistics for sketch simplification comparing QCNNs and CNNs based on average multiply-accumulate operations and memory usage per sample (forward pass) in convolutional layers.

Resolution	#MACC($\times 10^8$)		Memory (MB)	
	CNN	QCNN	CNN	QCNN
256×256	724.76	137.54	127.14	13.5
512×512	2899.05	349.28	508.56	35.35
1024×1024	11596.21	917.14	2034.24	97.9

demonstrated the use of QCNNs on sparse image classification and sketch simplification, where similar classification and simplification results are obtained but with much lower computational complexity and memory cost, compared to traditional convolutional neural networks. This feature makes QCNN very suitable for applications on mobile devices whose computing power is limited.

In future, we wish to study QCNNs with other network architectures in more detail such as residual networks [28], to learn from extremely large datasets such as Google Quickdraw [29]. We are also interested in extending our approach to recurrent architectures to learn from sparse image sequences and improve the learning speed of adversarial networks for training on sketch-like datasets [30].

Acknowledgements

We thank the anonymous reviewers for their constructive comments. This research is supported by the National Research Foundation under Virtual Singapore Award No. NRF2015VSG-AA3DCM001-018, and the BeingTogether Centre, a collaboration between Nanyang Technological University (NTU) Singapore and University of North Carolina (UNC) at Chapel Hill. The BeingTogether Centre is supported by the National Research Foundation, Prime Ministers Office, Singapore under its International Research Centres in Singapore Funding Initiative. This research is also supported in part by Singapore MoE Tier-2 Grant (MOE2016-T2-2-065).

References

1. Simo-Serra, E., Iizuka, S., Sasaki, K., Ishikawa, H.: Learning to simplify: Fully convolutional networks for rough sketch cleanup. ACM Transactions on Graphics **35**(4) (2016) 121:1–121:11
2. Wu, Z., Song, S., Khosla, A., Yu, F., Zhang, L., Tang, X., Xiao, J.: 3d shapenets: A deep representation for volumetric shapes. In: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2015) 1912–1920
3. Riegler, G., Ulusoy, A.O., Geiger, A.: Octnet: Learning deep 3d representations at high resolutions. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2017) 6620–6629
4. Wang, P.S., Liu, Y., Guo, Y.X., Sun, C.Y., Tong, X.: O-cnn: Octree-based convolutional neural networks for 3d shape analysis. ACM Transactions on Graphics **36**(4) (2017) 72:1–72:11
5. Tatarchenko, M., Dosovitskiy, A., Brox, T.: Octree generating networks: Efficient convolutional architectures for high-resolution 3d outputs. CoRR **abs/1703.09438** (2017)
6. Hunter, G.M., Steiglitz, K.: Operations on images using quad trees. IEEE Transactions on Pattern Analysis and Machine Intelligence **1**(2) (1979) 145–153
7. Jackins, C.L., Tanimoto, S.L.: Oct-trees and their use in representing three-dimensional objects. Computer Graphics and Image Processing **14**(3) (1980) 249 – 270
8. Gervautz, M., Purgathofer, W.: A simple method for color quantization: Octree quantization. In Magnenat-Thalmann, N., Thalmann, D., eds.: New Trends in Computer Graphics, Berlin, Heidelberg, Springer Berlin Heidelberg (1988) 219–231
9. Sullivan, G.J., Baker, R.L.: Efficient quadtree coding of images and video. IEEE Transactions on Image Processing **3**(3) (1994) 327–331
10. Agarwala, A.: Efficient gradient-domain compositing using quadtrees. ACM Transactions on Graphics **26**(3) (2007)
11. Gargantini, I.: An effective way to represent quadtrees. Communications of the ACM **25**(12) (1982) 905–910
12. Graham, B.: Spatially-sparse convolutional neural networks. CoRR **abs/1409.6070** (2014)
13. Graham, B.: Sparse 3d convolutional neural networks. In Xianghua Xie, M.W.J., Tam, G.K.L., eds.: Proceedings of the British Machine Vision Conference (BMVC), BMVA Press (2015) 150.1–150.9
14. Graham, B., van der Maaten, L.: Submanifold sparse convolutional networks. CoRR **abs/1706.01307** (2017)
15. Riegler, G., Ulusoy, A.O., Geiger, A.: Octnet: Learning deep 3d representations at high resolutions. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2017) 6620–6629
16. Chellapilla, K., Puri, S., Simard, P.: High Performance Convolutional Neural Networks for Document Processing. In Lorette, G., ed.: Proceedings - Tenth International Workshop on Frontiers in Handwriting Recognition, Université de Rennes 1, Suvisoft (2006)
17. Springenberg, J.T., Dosovitskiy, A., Brox, T., Riedmiller, M.A.: Striving for simplicity: The all convolutional net. CoRR **abs/1412.6806** (2014)
18. Zeiler, M.D., Fergus, R.: Visualizing and understanding convolutional networks. In Fleet, D., Pajdla, T., Schiele, B., Tuytelaars, T., eds.: Computer Vision – ECCV 2014, Cham, Springer International Publishing (2014) 818–833

19. Ronneberger, O., Fischer, P., Brox, T.: U-net: Convolutional networks for biomedical image segmentation. In Navab, N., Hornegger, J., Wells, W.M., Frangi, A.F., eds.: Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015, Cham, Springer International Publishing (2015) 234–241
20. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093 (2014)
21. Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. In: Proceedings of the 27th International Conference on International Conference on Machine Learning. ICML'10, USA, Omnipress (2010) 807–814
22. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research **15** (2014) 1929–1958
23. Cohen, G., Afshar, S., Tapson, J., van Schaik, A.: EMNIST: an extension of MNIST to handwritten letters. CoRR **abs/1702.05373** (2017)
24. Liu, C.L., Yin, F., Wang, D.H., Wang, Q.F.: Online and offline handwritten chinese character recognition: Benchmarking on new databases. Pattern Recognition **46**(1) (2013) 155 – 162
25. Eitz, M., Hays, J., Alexa, M.: How do humans sketch objects? ACM Transactions on Graphics **31**(4) (2012) 44:1–44:10
26. Zeiler, M.D.: ADADELTA: an adaptive learning rate method. CoRR **abs/1212.5701** (2012)
27. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics. Volume 9 of Proceedings of Machine Learning Research., PMLR (2010) 249–256
28. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2016) 770–778
29. Ha, D., Eck, D.: A neural representation of sketch drawings. CoRR **abs/1704.03477** (2017)
30. Simo-Serra, E., Iizuka, S., Ishikawa, H.: Mastering sketching: Adversarial augmentation for structured prediction. CoRR **abs/1703.08966** (2017)