QCNN: Quadrature Convolutional Neural Network with Application to Unstructured Data Compression

Kevin Doherty∗1, Cooper Simpson∗2, Stephen Becker1, and Alireza Doostan2

1Applied Mathematics, University of Colorado Boulder
2Smead Aerospace Engineering Sciences, University of Colorado Boulder

November 11, 2022

Abstract

We present a new convolution layer for deep learning architectures which we call QuadConv — an approximation to continuous convolution via quadrature. Our operator is developed explicitly for use on unstructured data, and accomplishes this by learning a continuous kernel that can be sampled at arbitrary locations. In the setting of neural compression, we show that a QuadConv-based autoencoder, resulting in a Quadrature Convolutional Neural Network (QCNN), can match the performance of standard discrete convolutions on structured uniform data, as in CNNs, and maintain this accuracy on unstructured data.

1 Introduction

Discrete convolutions are one of the canonical operations used in a plethora of deep learning applications. They have thoroughly been proven to be effective in extracting important features from data, and they possess a number of desirable properties. In particular, and most relevant for our work, is the locality of the filters due to their compact support. This facilitates the extraction of local features from the data, and provides a significant boost to computational efficiency, both in terms of time and memory. Other properties such as translation and rotation invariance are also useful in many settings. These traditional convolutions, however, rely on the assumption that the data is defined on a uniform grid. Such limitations are unfortunate, as there are a host of settings where convolutions may be effective, but the relevant data is unstructured; Fig. 1 shows a few representative examples.

In this paper we introduce a quadrature-based discrete convolution operator suitable for unstructured data, which we call QuadConv. Our construction is based on the continuous definition of convolution, which, for two functions \( f, g : \mathbb{R}^D \rightarrow \mathbb{R} \), is defined as follows:

\[
(f \ast g)(y) = \int_{\mathbb{R}^D} f(x) \cdot g(y - x) \, dx.
\] (1)

A variety of conditions on \( f \) and \( g \) can guarantee this integral is well defined, e.g., Young’s Inequality, or, of particular relevance here, if \( f \) and \( g \) are compactly supported. Note that we use the term quadrature to refer to a weighted sum approximation of an integral. Some authors make a distinction between one dimensional quadrature and higher dimensional cubature which we will not employ. Our proposed method is mathematically quite simple, but, as we will see, it is non-trivial to build an efficient implementation.

We will denote vectors using lowercase bold font (e.g., \( \mathbf{x} \)), and matrices or operators using uppercase bold font (e.g., \( \mathbf{X} \)). We will often refer to \( g \) as the kernel (or filter) and \( f \) as the data. This distinction is important in this context, as the data \( (f) \) is given and the kernel \( (g) \) is a learned map. Where appropriate, continuous functions will be referred to

∗These authors contributed equally to this work.
with an argument from their domain in parenthesis $f(\cdot)$ or $f(x)$, and its discrete counterpart will be referred as $f(x_i)$, or, if the arguments are clear from context, with an index $f_i$.

To summarize our contributions, we propose a novel convolution operator for deep learning applications that is suitable for unstructured data. In addition to this, we discuss the practical implementation of our operator, showing that it is a computationally feasible approach. Lastly, we present the application of our work to autoencoder based data compression. We show that it matches the effectiveness of standard convolutions on structured uniform data, and performs equally as well on unstructured data.

We will finish the remainder of this section by motivating our approach, and discussing the related literature. Section 2 will then introduce our mathematical formulation, with Section 2.1 describing the practical implementation. In Section 3 we will apply our method to a number of datasets on uniform grids and unstructured meshes. Section 4 will summarize our work and present possible directions for future research.

1.1 Motivation

The standard convolution operator used in deep learning is a particular discretization of Eq. (1) operating on $D$-dimensional tensors. For example, for a single output and input channel, the one dimensional form is given as follows:

$$(f * g)(y) = \sum_i f(x_i) \cdot g(y - x_i).$$

This discretization is mathematically justified as long as the points are equally spaced, and is drawn from the definition of convolution for functions defined on $\mathbb{Z}$, which is the same as Eq. (2) up to notation. The following example will demonstrate how using Eq. (2) in the mathematically justified setting can yield accurate results, the effects of using this form of convolution on non-uniformly spaced points, and how our proposed method fixes these issues. To that end, consider the following functions:

$$f(x) = \sin(\pi x) + \sin(14\pi x) \quad \text{and} \quad g(x) = \frac{8 \sin(8\pi x)}{\pi x},$$

where $f$ is a signal composed of low and high frequency sine waves, and $g$ is the ideal low-pass filter whose action under convolution will remove the higher frequency from $f$. We can see the analytic results of this convolution in Fig. 2a.
In Fig. 2b, one can observe the normal usage of the discrete convolution, with uniformly spaced grid points, and note that its result closely matches the continuous baseline. Figure 3, on the other hand, considers the results of a number of convolution type operations when the input signal is sampled in a non-uniform manner.

Figure 3a visualizes the results of discrete convolution if we sample the data at non-uniform locations, but continue to use the uniformly sampled kernel. These results only vaguely resemble the analytic output. The discrete convolution of Fig. 3b employs a continuous kernel that can be sampled wherever necessary and performs markedly better. However, the use of a continuous kernel alone is not sufficient to effectively approximate the accuracy of the analytic convolution. Instead, in this work, we propose modeling the kernel, \( g \), as a continuous function and performing the calculation of discrete convolution as a quadrature approximation of Eq. (1). This yields the results we see in Fig. 3c. The method we propose includes quadrature weights in Eq. (2), so that our 1D equivalent would be the following:

\[
(f * g)(y) = \sum_i \rho_i \cdot f(x_i) \cdot g(y - x_i). \tag{3}
\]

The combination of the continuous filter and the reduction of integration error gives us the best approximation of the operation we wish to perform. Although this example presumes a fixed continuous kernel, as opposed to a learned kernel, it demonstrates the fundamental problem of naively applying convolutional kernels to discrete data on non-uniform points.
1.2 Related Work

Graph convolutions Graph convolutions are perhaps the most widely used convolution method for unstructured data, applicable when the data has a readily available adjacency structure. For a full review of the relevant methods see [31]. In general, these methods are either spectral or spatial. Spectral graph convolutions operate performing convolution in the Fourier domain, computing the convolution as a point-wise product of two signals, whereas spatial graph convolutions work in the spatial domain directly. Graph convolutions do not always have clear connections to traditional convolution. Operations such as down-sampling over an arbitrary graph are complicated and require extra operations in order to transform the spatial domain of the data. This makes them unsuitable for spatially embedded data, where the coordinates of the points can be used instead of the graph structure.

Fundamentally, since graph convolutions use only adjacency structure, they do not exploit the full knowledge of unstructured data. For example, recall the 1D example of the previous section. In 1D, all grids (uniform or not) of the same number of nodes will have the same adjacency structure, so graph based methods could not distinguish uniform from non-uniform samples. As an extreme example, suppose $f$ has domain $[-1, 1]$ but is supported only in $[0, 1]$. A graph based method could not distinguish non-uniform samples of $f$ (within $[-1, 0)$) from uniform samples of the zero function.
Quadrature Convolutions

Kevin Doherty, Cooper Simpson, Stephen Becker, and Alireza Doostan

Point cloud convolutions There are numerous approaches to convolution for spatially embedded data, owing to the availability of point cloud data from LiDAR measurements [12]. Some of these methods voxelize the input data and perform 3D convolutions on the resulting voxels [30]. This often results in significant sparsity which can be taken advantage of for reasonable computational complexity. However, it eliminates the application to many unstructured problems, if, for example, maintaining variable levels of density is desirable. Other methods may perform convolution directly on the points by various methods such as utilizing Multi-Layer Perceptrons (MLPs) to define the convolutional kernels over all the necessary locations [1] or changing the definition of convolution to adapt to the irregular domain [32, 16]. We believe the most similar of these approaches to our work is PointConv [29], which attempts to approximate the integral via Monte Carlo integration. However, we approximate the integral via quadrature and since typical meshes are in three dimensions or less, there is little advantage to Monte Carlo over quadrature.

Continuous convolutions Other approaches have also leveraged the continuous setting as a way of improving convolution on structured signals. By treating data (e.g., image and video) as sampled versions of inherently continuous signals, [22] introduces a continuous convolution kernel (S4ND). This generalizes the state space model of S4 [11] to arbitrary dimension. Their kernel is constructed (globally) as a Kronecker product of sampled one-dimensional convolution kernels created as a linear combination of a hand-picked set of basis functions. The authors do not consider the application to unstructured data, although it is feasible that their global kernel could be constructed to match the underlying mesh.

During the preparation of this paper, the work of [4] was made available, which introduces their “continuous filter” – a continuous convolutional operator. Similarly to the approach we take, they represent the kernel as an MLP and learn it from data, which allows them to operate on arbitrary points. The authors seek to develop an operation which resembles traditional convolutions as much as possible, while still being applicable to unstructured data. This differs from our motivation via quadrature, and leads to several significant differences in construction. Most obvious is our inclusion of quadrature weights, whereas they approximate the convolution integral simply as an iterated sum.

General applications There are many applications where the density of points is directly related to the underlying data and can be used to achieve more accurate convolution integrals than the traditional convolution discretization. Non-uniform unstructured meshes are prevalent in PDE simulations, where nodes may be concentrated in areas of large variation in either the boundary or the PDE solution. Since PDE solvers use quadrature themselves, utilizing these meshes should also make convolution more accurate when appropriately leveraged.

Compression of scientific data The simulation of PDEs can create immense amounts of data, which then requires compression in order to store on disk for later scientific usage. While more classical methods exist for both lossless [18] and lossy compression [17, 5] for these types of datasets, they are largely limited to structured uniform grids or lose efficiency when applied to unstructured data. Neural compression has also proven to be very effective with structured uniform grids achieving very high compression ratios (in excess of 100×) with sufficiently low distortion to enable the usage of these methods with scientific data [9].

Other scientific applications In addition to data compression, neural networks have been applied to PDE super-resolution methods [28] and reduced-order modeling [20, 19]. All of these techniques could benefit from a generalization of traditional convolution that is applicable to the non-uniform unstructured meshes commonly found in PDE simulation data.

2 Methods

We will begin by discussing the construction of our discrete convolution operator, and then describe the details associated with a practical implementation. Our approach will be to approximate Eq. (1) via quadrature, so denote the nodes as $x_i$ and the weights as $\rho_i$, for $i = 1, \ldots, N$, and then consider the following:

$$ (f * g)(y) \approx \sum_{i=1}^{N} \rho_i \cdot f(x_i) \cdot g(y - x_i). \quad (4) $$
We employ the following form for our kernel function \( g \):

\[
g(z) = \text{bump}(z) \cdot h(z; \theta),
\]

where \( \theta \) are learnable parameters for some function \( h : \mathbb{R}^D \rightarrow \mathbb{R} \). The definition of the bump function is then given below for some \( \alpha > 0 \):

\[
\text{bump}(z) = \begin{cases} 
\exp \left( 1 - \frac{1}{1 - \alpha \|z\|^4} \right) & \|z\|^4 < \frac{1}{\alpha} \\
0 & \text{else}
\end{cases}.
\]

We can observe that \( \alpha \) allows us to control the support of \( g \), and therefore the effective size of the kernel. As discussed earlier, this compact support is desirable because we want to extract local features from the input data. It also has the other important consequence of significantly reducing the computational overhead, which we will see in more detail later on.

We have written equation Eq. (4) as a single sum based on the direct quadrature approximation of Eq. (1), but this can also be viewed as an iterated integral and a corresponding iterated sum. For example, in two dimensions we could let \( x = (x_1, x_2) \) and \( y = (y_1, y_2) \) to rewrite Eq. (4) as follows:

\[
(f * g)(y_1, y_2) \approx \sum_{i=1}^{N_1} \sum_{j=1}^{N_2} \rho_{ij} \cdot f(x_1^i, x_2^j) \cdot g(y_1 - x_1^i, y_2 - x_2^j),
\]

where we have quadrature weights \( \rho_{ij} \) for \( i = 1, \ldots, N_1 \) and \( j = 1, \ldots, N_2 \). In fact, if the points lie on a grid, then using the two-point composite Newton-Cotes quadrature recovers traditional convolution. Such a choice of quadrature yields weights \( \rho_{ij} = 1/4 \), and equates to using the trapezoidal rule along each dimension. This constant weight can then be absorbed into the learned kernel, producing the simple finite sum we see with traditional convolutions.

The specifics of the quadrature (i.e. the nodes and weights) is a question so far unaddressed. In general, we consider the nodes to be fixed as part of the input data. In other words the quadrature nodes are fixed by the locations at which the input signal \( f \) is evaluated at. In many scenarios, such as some PDE simulations similar to those we consider in Section 3 using finite-element methods, a set of points may come with associated weights. If this is the case, then the convolution applied to these points may appropriate those weights. In other cases, the weights may not already be provided, but can be readily computed on the fly. For example, if the input points are on a grid, then the operator may use the Newton-Cotes weights. Because we already expect to learn the kernel (i.e, \( \theta \)) from data, perhaps the easiest option is to learn the quadrature weights as well. This approach is also particularly useful if the input points were not explicitly chosen according to some quadrature. When we learn the quadrature weights we ensure they are strictly positive in order to avoid catastrophic cancellation and resulting loss of precision [21].

Figure 4 visualizes, in two dimensions, the difference between the standard discrete convolution and our quadrature convolution. The standard method uses a grid based kernel (\( 3 \times 3 \) in this case) that slides across the spatial dimensions of the domain to compute the output values. However, when one moves to an unstructured mesh, this operation no longer applies. On the other hand, the quadrature method computes the output values using mesh nodes that lie within the compact support (opaque red circle) of the kernel, and so it is agnostic to the underlying mesh structure.
So far, we have only considered scalar valued inputs to the convolution operator, but it is necessary to extend this to a multi-channel (i.e. vector) setting. Traditional deep learning convolutions accomplish this by stacking multiple kernels into a filter, and then applying multiple filters to the input data. We will adopt a similar approach. Define \( G: \mathbb{R}^D \rightarrow \mathbb{R}^{C \times C} \) as a map from a point to a matrix, which we interpret as a stack of filters. Despite the incongruence in vocabulary, we will refer to \( G \) as a filter. Now, we consider \( f: \mathbb{R}^D \rightarrow \mathbb{R}^C \) as our vector valued input, and the quadrature convolution is given as follows:

\[
(f \ast G)(y) = \sum_{i=1}^{N} \rho_i \cdot G(y - x_i) \cdot f(x_i),
\]

where we can easily observe that the output lies in \( \mathbb{R}^{C} \). The mapping \( G \) is parameterized by \( \theta \) according to the multi-channel extension of Eq. (5):

\[
G(z) = \text{bump}(z) \cdot H(z; \theta),
\]

where \( H(\cdot; \theta): \mathbb{R}^D \rightarrow \mathbb{R}^{C \times C} \). The form of the bump function remains unchanged, and thus we see that \( G \) may be close to \( 0 \) depending on the location it is evaluated at. We may also note that the dimension, \( D \), of the data was arbitrary, so our operator can easily be applied to data in any dimension.

### 2.1 Practical Computation

The methodology we have developed so far is the foundation of our proposed operator, but certain aspects of the implementation are as of yet unclear. Further, specific elements of the computation must be leveraged to achieve any reasonable level of efficiency. We will discuss the practical implementation of the quadrature-based convolution operator, and compare its complexity to other convolution types.

In general, we consider input in the form of a tuple \((X, F)\), where \( X \in \mathbb{R}^{D \times N} \) are \( N \) points in \( D \)-dimensional space, and \( F \in \mathbb{R}^{C \times N} \) are the associated data with \( C \) channels. The QuadConv operator, which we will denote as \( Q \), acts in the following manner:

\[
(X, F) \xrightarrow{Q} (Y, \tilde{F}),
\]

where \( Y \in \mathbb{R}^{D \times \tilde{N}} \) are \( \tilde{N} \) output points, and \( \tilde{F} \in \mathbb{R}^{C \times \tilde{N}} \) are the recovered features with \( \tilde{C} \) channels. The practitioner is thus left to specify the number of output points and channels, and also the output locations themselves, for any given operator. Traditional deep learning convolutions also require manually setting the number of output channels, but the output structure is usually determined as a combination of various other hyperparameters such as the stride and padding. Our method easily facilitates up-sampling or down-sampling, and can maintain non-uniform density. This is due to the continuous kernel being defined at all points inside the domain, but the quadrature is only computed where the data is defined, which maximizes our re-sampling flexibility. The process by which the output points are computed is also left to the user to specify. In the simplest case, they can be placed on a grid, while a more complicated mesh agglomeration strategy (e.g. [3]) may be employed to construct them directly from the input points. The actual computation by \( Q \) via Eq. (8) is given below:

\[
Q[X, F]_j = (Y, \tilde{F})_j = (y_j, \sum_{i=1}^{N} \rho_i \cdot G(y_j - x_i) \cdot f_i),
\]

for \( j = 1, \ldots, \tilde{N} \). Note that \( y_j \) and \( f_i \) are the \( j \)th and \( i \)th columns of \( Y \) and \( F \) respectively. Recall from Eq. (9) that \( H \) provides us with the learnable map from a point to a matrix. Our method, in general, is agnostic to the form of this map, but for the sake of efficiency we opt for a single MLP, \( G: \mathbb{R}^D \rightarrow \mathbb{R}^{C \times C} \). There are many other possible choices, such as sharing the MLP amongst channels, or using multiple independent MLPs.

As noted earlier, depending on their evaluation location, the approximation \( G \approx 0 \) may hold for many of the filters. Thus, the sparsity of the sum in Eq. (8) can be taken advantage of to avoid undue computation. For any pair of indices \( i \) and \( j \), one may evaluate the bump function \( \text{bump}(y_j - x_i) \), and those indices where this result is nonzero generates a map of the form \( j \mapsto \{i_k\} \). That is to say, the index \( j \) of each output location has associated with it a sequence of the input indices \( \{i_k\} \) that contribute to its value. Figure 5 below visualizes how this map can be used to implement Eq. (10).
This entire process can be executed efficiently in a vectorized manner as the individual operations are independent, and for any given output index $j$ the sum that determines its features is over far fewer indices $i$ than would otherwise be the case. Assuming that the input and output locations are static, then the maps $j \mapsto \{i_k\}$ can be computed a single time as a pre-processing step and cached for future use. Doing so avoids the costly construction of the maps themselves, which significantly improves the execution time of the operator.

Table 1 shows the computational and memory complexity for the forward pass of our proposed method, along with two baseline convolutions. We acknowledge that, during training, the backward pass from automatic differentiation also has a computational cost associated with it, but we will not discuss that in depth here. For all methods, $\{C, N\}_{in}$ and $\{C, N\}_{out}$ are the input and output channels and points. We consider a standard convolution with kernel size $K$, and for QuadConv, we consider a sparsity factor $S$ which is dependent on the support of the bump function and the local point density. The variables $M_t$ and $M_m$ denote the time and memory complexity for the chosen filter operation (i.e., MLP). For graph convolutions, the quantity $E$ denotes the number of edges for the graph representation of the data (e.g., $E = 2^D N_{in}$ for nearest-neighbor representation on a uniform grid). When considering the peak memory consumption, we do not include the requirements for the input or output data, as that is common to all three methods. One may observe a number of important details from Table 1. With respect to QuadConv, we see that sparsity is incredibly important in terms of time complexity. As well, although it may not be immediately obvious, the time efficiency requires a trade off for more memory complexity due to caching and point manipulations. The time complexity of graph convolutions is highly dependent on the number of edges. Overall, the table is not to be used to conclude which type of convolution is best, since the three types of convolutions shown apply in different settings.

|                | Standard          | Graph [15] | QuadConv                      |
|----------------|-------------------|------------|-------------------------------|
| **Execution Time** | $K^D N_{out} C_{in} C_{out}$ | $EC_{in} C_{out}$ | $M_t S N_{in} N_{out} C_{in} C_{out}$ |
|                | $K^D NC^2$        | $2^D NC^2$ | $M_t S N^2 C^2$               |
| **Peak Memory** | $K^D C_{in} C_{out}$ | $N_{in} + C_{in} C_{out}$ | $M_m + N_{in} + N_{out} + S N_{in} N_{out} C_{in} C_{out}$ |
|                | $K^D C^2$         | $N + C^2$  | $M_m + S N^2 C^2$             |

Table 1: Big-O comparison of convolution-type operators. The second row of the time and memory blocks take $N_{in}, N_{out} = O(N), C_{in}, C_{out} = O(C)$, and $E = 2^D N$ in order to facilitate comparisons and keeps just the leading order terms.
3 Numerical Experiments

We have implemented our quadrature convolution operator [6] using PyTorch [24] and Lightning [8]. In this section, we will conduct a number of experiments that validate our method of convolution as an effective tool in deep learning. In particular, we perform data compression of partial differential equation (PDE) simulation data using an autoencoding neural network.

First, we will compare our methods effectiveness to the traditional approach with data that lies on a grid. We then interpolate this dataset onto a non-uniform mesh and show that our new method still performs equally as well. Finally, we consider a different dataset which was simulated on an unstructured mesh, and show that our operator continues to achieve similar levels of performance. Although we examined graph convolutions in Table 1, we will not compare against this method due to their computational inefficiency and lack of obvious down-sampling procedure. This latter point is quite important, as down-sampling the mesh inside of the autoencoder is a key aspect in its effectiveness.

Figure 6 provides a high-level visualization of the architecture in question. The input is passed through the encoder, which is comprised of a Quadrature Convolutional Neural Network (QCNN) that progressively down-samples the number of points and increases the number of channels, and an MLP which outputs the latent representation $\mathbf{z}$. This latent vector is the compressed state of the data. To reconstruct, one simply performs the reverse operation. An MLP is applied to the representation $\mathbf{z}$, and then a QCNN, referred to as the decoder, up-samples the data. Our QCNN is simply a Convolutional Neural Network (CNN), a series of convolutional layers and non-linear activation functions, where the standard convolutions have been replaced with our QuadConv operators. For all experiments, after the first QuadConv layer, we re-sample output points to a grid and use the associated two-point composite Newton-Cotes weights. Doing this also allows us to use the max pooling operation and its adjoint to up-sample and down-sample respectively.

![Autoencoder Architecture](image)

Our autoencoder aims to minimize a regularized mean-squared error loss between $\mathbf{F}$ and $\tilde{\mathbf{F}}$,

$$
\mathcal{L}(\mathbf{F}, \tilde{\mathbf{F}}) = \|\mathbf{F} - \tilde{\mathbf{F}}\|_2^2 + \lambda R(\mathbf{F}, \tilde{\mathbf{F}}),
$$

averaged over all data samples $\mathbf{F}$, where each data sample $\mathbf{F}$ is one time slice from the (time-dependent) PDE simulation. Our regularization term $R(\mathbf{F}, \tilde{\mathbf{F}})$ in the structured uniform case is the mean-squared error between finite difference derivatives of $\mathbf{F}$ and $\tilde{\mathbf{F}}$ (i.e. the Sobolev norm). These finite difference terms match those that are bounded in the PDE solution. In our unstructured examples this is more difficult to define so we do not use any regularization. The final “error” we report is the relative $\ell^2$-error averaged over all data samples:

$$
\frac{\|\mathbf{F} - \tilde{\mathbf{F}}\|_2}{\|\mathbf{F}\|_2}.
$$

All experiments were conducted on 4 V100 GPUs and 1 IBM Power9 CPU. Each neural network is trained with the Adam [14] optimizer using a batch size of 8. This means that the number of steps in each of the tables that follow, and a single gradient step within each epoch, correspond to processing the same amount of data.
3.1 Structured Uniform Data

The structured dataset which we use in this section consists of 450 uniformly sampled time steps of a jet ignition simulation that lie on a uniform spatial grid of size $50 \times 50$ and is represented with 32-bit floating point numbers (single precision) [13]. The wave front is fully resolved in time. This dataset is transport dominated for the initial portion of the simulation until the jet flame reaches steady state. Each of these spatial samples is embedded into the latent space of our autoencoder, where it is represented by $z \in \mathbb{R}^{50}$, hence resulting in $50 \times$ compression. The second half of our autoencoder maps $z$ back to $\mathbb{R}^{50 \times 50}$ where this result is compared to the original data to develop our error metrics. Because the input points are already on a grid, we also use two-point composite Newton-Cotes quadrature weights for the input QuadConv layers. This matches the quadrature interpretation of the CNN which gives a very direct comparison of the QuadConv layer to the traditional CNN layer. Ultimately, we hope that the performance of the traditional CNN matches that of our QCNN on a grid. We can see in Table 2 that they indeed have very similar performance, although the QCNN does take longer to train due to the extra cost of each iteration. Even at these high compression ratios, we see few readily visible errors in the data in Fig. 7. The error visualization at the bottom compresses the color scale by an order of magnitude in order to better expose the error that does is present.

| Model Type | Error | Training Time (h) | # of Steps | # of Trainable Parameters |
|------------|-------|-------------------|------------|--------------------------|
| CNN        | 0.40% | 1.46              | 186,000    | 1,024,718                |
| QCNN       | 0.63% | 3.91              | 186,000    | 1,035,310                |

Table 2: Comparison of CNN and QCNN for ignition data compression, at $50 \times$ compression.

(a) CNN compression loss.  
(b) QCNN compression loss.

Figure 7: Comparison of CNN and QCNN ignition data reconstruction. Note that the bottom row has a rescaled color bar.
We conduct a further error analysis to understand relative performance in Figs. 8 and 9, which shows a more full extent of the similarity of performance between the two networks. This shows the distribution of the absolute errors are similar, as well as the relative error histogram, which helps to identify at what absolute pixel values large relative errors occur.

3.2 Unstructured Data

When the data is instead on an unstructured domain, the application of a traditional convolutional neural network becomes poorly defined. In the following experiments, we learn the quadrature weights associated with the input mesh in the first QuadConv layer and then re-sample the data to a structured uniform grid. After this, the autoencoder’s architecture is the same as the above experiments on a grid, until the final layer of the network which re-samples the data from a structured uniform grid back to the non-uniform mesh.
3.2.1 Unstructured ignition data

To maintain a benchmark of performance we interpolated the ignition data with 2D splines and evaluated the interpolant on the non-uniform unstructured mesh shown below. This mesh contains 2189 points, more concentrated in the middle and the right-side of the domain, and was generated with dmsh [25]. In order to evaluate the error introduced during the interpolation and subsequent re-sampling on the non-uniform mesh, we re-interpolated the data from the non-uniform mesh and compared it to the original data from the structured uniform grid. Using the same relative $\ell^2$-error averaged over all data samples as we report in our tables, we found the interpolation process introduced 0.122% error into the data. The latent space of the autoencoder is once again $z \in \mathbb{R}^{50}$ and the data is still represented as 32-bit floating point numbers.

| Model Type | Error   | Training Time (h) | # of Steps | # of Trainable Parameters |
|------------|---------|-------------------|------------|--------------------------|
| QCNN       | 0.410%  | 3.78              | 169,000    | 1,042,499                |

Table 3: QCNN results for unstructured ignition data compression.

As we can see in Table 3, Fig. 10a and Fig. 10b the results continue to be excellent. This suggests that our approximation is performing well over the non-uniform data found in the center of the domain, since we do not see an increase in error due to the change in the underlying structure of the data. Error histograms are similar to those in the uniform grid case and can be found in Appendix A.
3.2.2 Unstructured flow data

While the ignition examples are informative for verifying the performance of the QuadConv layer in comparison to traditional convolution, the unstructured mesh is not native to the data. We consider here an example of fluid flow around a cylinder in a narrow channel. This PDE simulation was conducted on the non-uniform mesh shown in Fig. 11 and exhibits regions of high and low density throughout the channel, as well as a region with no nodes that is occupied by the cylinder. This dataset has 300 time points and 7613 nodes in the domain.

This data is not as transport dominated as the previous example, and, as a result, it is less challenging to compress than the ignition data, which exhibits a higher Kolmogorov $N$-Width [10]. We continue to use the same general autoencoder structure; however, while our results thus far have focused on a $50 \times$ reduction in the data, this next example embeds the unstructured flow data in $\mathbb{R}^{15}$, which equates to a $500 \times$ reduction. This extra factor comes from the MLP, but it still requires effective feature extraction inside the QCNN. The flow data above also compresses well, with low error and converges considerably faster than the ignition data. This example shows that practical meshes are easily used, and high accuracy can be achieved with a QuadConv based network. The error histograms can be found in Appendix A.

| Model Type | Error | Training Time (h) | # of Steps | # of Trainable Parameters |
|------------|-------|-------------------|------------|--------------------------|
| QCNN       | 0.360%| 3.09              | 50,000     | 834,668                  |

Table 4: QCNN results for unstructured flow data compression.

We observe very little degradation in the quality of the solution and no visible changes in the error. In fact, since the linear layers in the network dominate the total number of trainable parameters, our embedding of the data into a smaller space reduces the overall number of parameters in the network and slightly reduces training time.

Figure 11: Unstructured flow mesh.

Figure 12: QCNN unstructured flow reconstruction.
4 Discussion

In this work we have presented a new convolution layer for deep learning applications involving unstructured data. Our operator approximates continuous convolution via quadrature, and employs a continuous learned kernel to allow for arbitrary discretizations of the input data. In addition to this, we discussed an implementation of this operator that yields sufficiently reasonable time complexity for use in modern deep learning settings. Our experiments showed that, in practice, this quadrature convolution can match the performance of traditional convolutions on grid based data, and performs equally as well on unstructured inputs.

4.1 Future Work

The results we have presented here demonstrate the exceptional performance of the QuadConv layer and QCNN in application, but there are still a number of interesting questions to address as to its performance and sensitivity to hyper-parameters. As discussed in Section 2.1, there are a number of ways to parameterize the map from point to filter, and we have only considered one such approach here. A more thorough investigation into the available approaches is warranted. Also discussed in Section 2, and put to use in Section 3, is the learning of the quadrature weights for a given set of input points. The accuracy of this learned quadrature itself is something that we can measure separately using known data and kernel function. Understanding this may inspire regularization of the quadrature weights that could enable learning higher accuracy approximations. It is also unclear if learning the quadrature may be advantageous even in situations where the weights are given. Mentioned briefly in Section 2.1 was the idea of constructing the output points of a QuadConv layer via an agglomeration of the input mesh. This would allow one to maintain properties of the original mesh while still down-sampling. We would like to incorporate such a strategy into our overall framework. Moreover, our continuous filter allows us to learn representations of data on a continuous space which enables one to perform multi-fidelity training. This would involve training on different resolutions of data using the same network but with variable quadrature weights in order to continue to approximate the convolution operator well while using the same kernels. Last but not least, developing online, parallel learning schemes to enable pass-efficient, ideally in situ, data compression, see, e.g. [7, 23], is an important future research direction.

Acknowledgments

This material is based upon work supported by the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003968 as well as Department of Energy Advanced Scientific Computing Research Award DE-SC0022283. We would also like to thank Kenneth Jansen, John Evans, and Jeff Hadley for their helpful discussions surrounding this work.

References

[1] Alexandre Boulch. “ConvPoint: Continuous convolutions for point cloud processing”. In: Computers & Graphics 88 (2020), pp. 24–34. ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2020.02.005. URL: https://www.sciencedirect.com/science/article/pii/S0097849320300224.

[2] John Burkardt. Airplane. Generated from 3D model. 2012. URL: https://people.sc.fsu.edu/~jburkardt/data/ply/ply.html (visited on 11/08/2022).

[3] Tony F Chan, Jinchao Xu, and Ludmil Zikatanov. “An agglomeration multigrid method for unstructured grids”. In: Contemporary Mathematics 218 (1998), pp. 67–81.

[4] Dario Coscia et al. “A Continuous Convolutional Trainable Filter for Modelling Unstructured Data”. In: (2022). DOI: 10.48550/ARXIV.2210.13416. URL: https://arxiv.org/abs/2210.13416.

[5] Sheng Di and Franck Cappello. “Fast error-bounded lossy HPC data compression with SZ”. In: 2016 ieee international parallel and distributed processing symposium (ipdps). IEEE. 2016, pp. 730–739.

[6] Kevin Doherty and Cooper Simpson. QuadConv. https://github.com/kvndhrty/QuadConv. A PyTorch implementation of quadrature convolutions. 2022.
[7] Alec M Dunton et al. “Pass-efficient methods for compression of high-dimensional turbulent flow data”. In: Journal of Computational Physics 423 (2020), p. 109704.

[8] William Falcon and The PyTorch Lightning team. PyTorch Lightning. Version 1.7. 2019. DOI: 10.5281/zenodo.3826935. URL: https://github.com/Lightning-AI/lightning.

[9] Andrew Glaws, Ryan King, and Michael Sprague. “Deep learning for in situ data compression of large turbulent flow simulations”. In: Phys. Rev. Fluids 5 (11 Nov. 2020), p. 114602. DOI: 10.1103/PhysRevFluids.5.114602. URL: https://link.aps.org/doi/10.1103/PhysRevFluids.5.114602.

[10] Constantin Greif and Karsten Urban. “Decay of the Kolmogorov N-width for wave problems”. In: (2019). DOI: 10.48550/ARXIV.1903.08488. URL: https://arxiv.org/abs/1903.08488.

[11] Albert Gu, Karan Goel, and Christopher Ré. “Efficiently Modeling Long Sequences with Structured State Spaces”. In: 10th International Conference on Learning Representations. ICLR. 2022. DOI: 10.48550/ARXIV.2111.00396. URL: https://arxiv.org/abs/2111.00396.

[12] Yulan Guo et al. “Deep Learning for 3D Point Clouds: A Survey”. In: IEEE Transactions on Pattern Analysis and Machine Intelligence 43.12 (2021), pp. 4338–4364. DOI: 10.1109/TPAMI.2020.3005434.

[13] “IEEE Standard for Binary Floating-Point Arithmetic”. In: ANSI/IEEE Std 754-1985 (1985), pp. 1–20. DOI: 10.1109/IEEESTD.1985.82928.

[14] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[15] Thomas N Kipf and Max Welling. “Semi-supervised classification with graph convolutional networks”. In: 5th International Conference on Learning Representations. ICLR. 2017. DOI: 10.48550/ARXIV.1609.02907. URL: https://arxiv.org/abs/1609.02907.

[16] Artem Komarichev, Zichun Zhong, and Jing Hua. A-CNN: Annularly Convolutional Neural Networks on Point Clouds. 2019. DOI: 10.48550/ARXIV.1904.08017. URL: https://arxiv.org/abs/1904.08017.

[17] Peter Lindstrom. “Fixed-Rate Compressed Floating-Point Arrays”. In: IEEE Transactions on Visualization and Computer Graphics 20 (Aug. 2014). DOI: 10.1109/TVCG.2014.2346458.

[18] Peter Lindstrom and Martin Isenburg. “Fast and Efficient Compression of Floating-Point Data”. In: IEEE transactions on visualization and computer graphics 12 (Sept. 2006), pp. 1245–50. DOI: 10.1109/TVCG.2006.143.

[19] Mohammadreza Momenifar et al. “A Physics-Informed Vector Quantized Autoencoder for Data Compression of Turbulent Flow”. In: 2022 Data Compression Conference (DCC). IEEE, Mar. 2022. DOI: 10.1109/dcc52660.2022.00039. URL: https://doi.org/10.1109%2Fdcc52660.2022.00039.

[20] Mohammadreza Momenifar et al. “Emulating Spatio-Temporal Realizations of Three-Dimensional Isotropic Turbulence via Deep Sequence Learning Models”. In: (2021). DOI: 10.48550/ARXIV.2112.03469. URL: https://arxiv.org/abs/2112.03469.

[21] Jean-Michel Muller et al. Handbook of Floating-Point Arithmetic. 2nd. Birkhäuser Basel, 2018. ISBN: 3319765256.

[22] Eric Nguyen et al. “S4ND: Modeling Images and Videos as Multidimensional Signals Using State Spaces”. In: (2022). DOI: 10.48550/ARXIV.2210.06583. URL: https://arxiv.org/abs/2210.06583.

[23] Heather Pacella et al. “Task-parallel in situ temporal compression of large-scale computational fluid dynamics data”. In: The International Journal of High Performance Computing Applications 36.3 (2022), pp. 388–418.

[24] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: Advances in Neural Information Processing Systems 32. Curran Associates, Inc., 2019, pp. 8024–8035. DOI: 10.48550/ARXIV.1912.01703. URL: https://arxiv.org/abs/1912.01703.

[25] Nico Schlömer and J. Harirhan. dmsh. https://github.com/meshpro/dmsh. Inspired by distmesh. 2022.

[26] Ryan W Skinner et al. “Reduced-basis multifidelity approach for efficient parametric study of naca airfoils”. In: AIAA Journal 57.4 (2019), pp. 1481–1491.

[27] Stanford University Computer Graphics Laboratory. Stanford Bunny. Generated from 3D model. 1994. URL: http://graphics.stanford.edu/data/3Dscanrep/ (visited on 11/08/2022).

[28] Karen Stengel et al. “Adversarial super-resolution of climatological wind and solar data”. In: Proceedings of the National Academy of Sciences 29 (2020), pp. 16805–16815.
[29] Wenxuan Wu, Zhongang Qi, and Li Fuxin. “PointConv: Deep Convolutional Networks on 3D Point Clouds”. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). June 2019.

[30] Zhirong Wu et al. “3D ShapeNets: A deep representation for volumetric shapes”. In: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Los Alamitos, CA, USA: IEEE Computer Society, June 2015, pp. 1912–1920. DOI: 10.1109/CVPR.2015.7298801. URL: https://doi.ieeecomputersociety.org/10.1109/CVPR.2015.7298801.

[31] Zonghan Wu et al. “A Comprehensive Survey on Graph Neural Networks”. In: IEEE Transactions on Neural Networks and Learning Systems 32.1 (Jan. 2021), pp. 4–24. DOI: 10.1109/tnnls.2020.2978386. URL: https://doi.org/10.1109/tnnls.2020.2978386.

[32] Yifan Xu et al. “SpiderCNN: Deep Learning on Point Sets with Parameterized Convolutional Filters”. In: Computer Vision – ECCV 2018. Ed. by Vittorio Ferrari et al. Cham: Springer International Publishing, 2018, pp. 90–105. ISBN: 978-3-030-01237-3.

A Appendices

![Error Analysis](image-url)

Figure 13: Error analysis for QCNN reconstruction of unstructured ignition data.
Figure 14: Error analysis for QCNN reconstruction of unstructured flow data.