Abstract.

Operational Neural Networks (ONNs) have recently been proposed as a special class of artificial neural networks for grid structured data. They enable heterogenous non-linear operations to generalize the widely adopted convolution-based neuron model. This work introduces a fast GPU-enabled library for training operational neural networks, FastONN, which is based on a novel vectorized formulation of the operational neurons. Leveraging on automatic reverse-mode differentiation for backpropagation, FastONN enables increased flexibility with the incorporation of new operator sets and customized gradient flows. Additionally, bundled auxiliary modules offer interfaces for performance tracking and checkpointing across different data partitions and customized metrics.

Keywords: Operational Neural Networks; ONN; FastONN;

1 Introduction

CNNs have advanced the state-of-the-art in challenging computer vision problems including object recognition, detection and segmentation [1]–[3]. Each of the stacked layers of convolutional neurons inside a CNN, applies filtering to local regions (receptive fields) of the input using the discrete convolution operation with learnable filter coefficients. Despite its wide-scale adoption, the convolutional neuron model, being an extension of a simple perceptron, is inherently linear, where the lone source of non-linearity is the point-wise non-linear activation. Consequently, a significantly large number of layers with interlaced non-linear activation functions are required to yield a hypothesis space strong enough to navigate complex non-linear spaces. Recently, the idea of Operational Neural Networks (ONN) [4] has been proposed which is a diversification of the convolutional neuron model that involves embedding non-linear transformations at the level of individual receptive fields guided by learnable kernels.

The primary building block of the ONN framework is the operational neuron model which extends the principles of GOPs [5]–[7] to convolutional realm. While retaining favorable characteristics of sparse-connectivity and weight-sharing, an operational neuron provides the flexibility to incorporate non-linear transformations within local receptive fields without the overhead of additional trainable parameters. Driven by rich non-linear operators (operator sets), ONNs were shown to outperform CNN across a variety of challenging learning problems such as image-to-image translation, denoising, synthesis and segmentation [4]. However, the primary bottleneck hindering large scale adoption of ONNs is the lack of an open-source fast GPU enabled implementation which can facilitate reproducibility and fuel independent research efforts.

In this study, we propose FastONN, the first open-source Python-based and GPU-enabled ONN implementation, which aims at facilitating training and configuration of ONNs. FastONN leverages a novel formulation of an operational neuron which allows efficient utilization of GPU for increased computational efficiency. The library comprises of standalone modules, which allow training of end-to-end operational networks while preserving the ability to embed an operational module (operational layer/neuron) within any given network architecture. In addition, by automating gradient calculation, FastONN eases incorporation of novel operators to further diversify the operator set library.

2 Problems and Background

The primary building block of a convolutional neuron is the 2D discrete convolution operation. The convolution of a 2D image \( y \in \mathbb{R}^{M \times N} \) with a filter \( w \in \mathbb{R}^{m \times n} \) is given as:

\[
    x(i, j) = \sum_{u=0}^{m-1} \sum_{v=0}^{n-1} w(u, v) y(i - u, j - v)
\]

Figure 1. Operational neuron incorporates patch-wise non-linear operations in the linear formulation of the convolutional neuron using generic nodal \( \psi \) and pool \( f \) functions.

Given any operator set \( \theta = (\psi, \phi, f) \), an operational neuron equipped with \( \theta \) modifies the operation of (1) as depicted in Figure 1 and formulated as:

\[
    x(i, j) = \phi^{n-1}_{u=0} \phi^{m-1}_{v=0} \left( \psi(w(u, v), y(i - u, j - v)) \right)
\]
where $\psi, \phi$ and $f$ are termed as nodal, pool and activation operator functions respectively.

FastONN uses an alternate formulation of the above operation. Firstly, $y$ is reshuffled such that values inside each $m \times n$ sliding block of $y$ are vectorized and concatenated as rows to form a matrix $Y \in \mathbb{R}^{M \times N}$ where $M = mn$ and $N = \tilde{N} = \tilde{N}_L \times \tilde{N}_W$. We can denote each element of $Y$ as,

$$Y(i, j) = y(\neg(i, j))$$  \hspace{1cm} (3)

where $\neg$ is a matrix that stores the index locations of $y$ in $Y$. This operation is commonly referred to as “im2col” and is critical in conventional GEMM-based convolution implementations. Secondly, we construct a matrix $W \in \mathbb{R}^{\tilde{M} \times \tilde{N}}$ whose rows are repeated copies of $\vec{w} = vec(W) \in \mathbb{R}^{mn}$, where $vec(\cdot)$ is the vectorization operator. Each element of $W$ is given by the following equation:

$$W(i, j) = \vec{w}(i)$$  \hspace{1cm} (4)

The convolution operation of (1) can then be expressed as,

$$x = vec_{M \times N}^{-1} \left( \sum_i (Y \otimes W) \right)$$  \hspace{1cm} (5)

where $\otimes$ represents the Hadamard product, $\sum_i$ is the summation across the dimension $i$. In (5), $vec_{M \times N}^{-1}$ is the inverse vectorization operation that reshapes the $MN$-sized vector back to $M \times N$. The alternate formulation of operational neuron in (3) can now be generically reformulated as follows:

$$x = vec_{M \times N}^{-1}(\phi(\psi(Y, W)))$$  \hspace{1cm} (6)

where $\psi(\cdot) : \mathbb{R}^{M \times N} \to \mathbb{R}^{M \times N}$ and $\phi(\cdot) : \mathbb{R}^{\tilde{M} \times \tilde{N}} \to \mathbb{R}^{\tilde{N}}$ are the nodal and pool functions respectively. It can be clearly seen that the convolution operation of (5) is now a special form of (6) with nodal function $\psi(\alpha, \beta) = \alpha \ast \beta$ and pooling function $\phi(\cdot) = \sum_i$. To complete the forward-propagation, a non-linear activation operation $f$ follows the convolution. So, the final output of a convolutional neuron can be expressed as follows:

$$y = vec_{M \times N}^{-1}f(x)$$  \hspace{1cm} (7)

where $f(\cdot)$ represents the point-wise activation operation. With the aforementioned formulation, each forward pass inside an operational neuron involves a single Hadamard product between two large matrices, $Y$ and $W$. Such a formulation is highly desirable for efficient utilization of GPU resources as it prevents intermittent memory copying and allows advanced optimization techniques such as tiling for enhanced computational performance [8]. Furthermore, as discussed in detail in Section 4.1, the backpropagation, and in turn the gradient calculation process, is handled by the automatic reverse-mode backpropagation engine, thus enabling differentiation through any choice of $\psi, \phi$ and $f$, as long as they follow the semantics provided in Table 2.
3 Software Framework

3.1 Software Architecture
The core of the library consists of three main classes, as depicted in the entity-relationship diagram in Figure 2 and described below:

- **Operational Block**: class implementing the atomic operational neuron. Each OpBlock object has an associated operator set that is used for accomplishing the required non-linear operation.

- **Operational Tier**: has one or more operational blocks and implements the unfolding operation, propagation through operational blocks and subsampling operations. An OpTier object represents the abstraction of a neural network layer and can be used as an out-of-the-box replacement for a convolutional layer in any given architecture.

- **Operational Network**: has one or more operational tiers and implements propagation through the layers. An OpNetwork object is an end-to-end ONN that can be trained in the same way as any other ANN model.

In addition to the main classes listed above, a variety of auxiliary modules are also provided that facilitate training:

- **OSL**: operator set library module which houses all nodal, pool and activation function definitions. Table 2 shows details of a sample operator set library along with generic formulations for nodal, pool and activation functions that can be used to add custom functions.

- **Trainer**: helper class which supports a wide array of functionalities used for training a model for the given number of epochs and randomly initialized runs, evaluation on validation and test data, plotting of metrics, performance tracking and model checkpointing across all metrics and data partitions.

3.2 Software Functionalities
Following are the major highlights of the features of FastONN:

- Configuration of a fully operational end-to-end network with customized number of layers, neurons per layer, kernel sizes and subsampling factors. Each layer is capable of full heterogeneity with the ability of specifying a different operator set for individual neurons.

- A comprehensive operator set library with multiple predefined nodal, pool and activation functions. New operators can be added following simple semantics as illustrated in Table 2.

- Back-Propagation (BP) training an ONN network on GPU using custom defined objective functions and any choice of major optimization techniques such as SGD with momentum, Adam etc.

- Defining custom performance metrics and dynamically tracking them across the training process for all data partitions. Automatic checkpointing is enabled for each metric to allow rolling back of the model to any desired state. Any state of the model is not limited to the values of trainable parameters, but also includes the state of the optimizer and other training statistics in order to allow seamless resumption of the training.

4 Implementation and Empirical Results

4.1 Implementation details
FastONN is based on PyTorch [9], a popular Python-based deep learning framework, which enables imperative Pythonic programming styles for designing high-performance GPU-accelerated neural networks. In addition, PyTorch leverages the AutoGrad [10] package to provide an intuitive interface for reverse-mode automatic differentiation, thus automating the gradient computation process [11] for many choices of nodal, pool and activation functions. Nevertheless, the ability to define custom backpropagation formulation is still preserved and can be used to define custom derivatives and perform gradient modification for complex operators. This is essential for a flexible ONN implementation as the gradient flows within operational neurons are highly customized [4], owing to patch-wise non-linearities. A detailed comparison of analytical and automatic backpropagation through an operational neuron is provided in the supplementary material. PyTorch also supports broadcasting semantics which allow making virtual copies of a tensor along certain dimension(s) without explicitly copying the data, which results in improved memory utilization. This is essential for an operational neuron as it provides a memory-efficient way to accomplish the formulation of (3). All the standalone modules of FastONN shown in Figure 2 inherit from the `nn.Module` base class of Pytorch, thus ensuring compatibility and reusability. The source-code for FastONN is available online as a public Github repository and uses Git versioning system as well as supporting continuous integration with automated build and tests.

4.2 Empirical Results
We provide empirical results of training an ONN for the image transformation as introduced in [4], using the FastONN library. The task involves configuring and training an ONN for learning 10 distinct folds of 4-to-4 face image translations. Table 1 presents the results for each of the 10 folds along with the computational efficiency in terms of time taken per image to train the network on GPU, including performance tracking, plotting and checkpointing.

5 Illustrative Examples
In this section, we provide the code illustrations for the experiments conducted to produce the results presented in
The Trainer class, which comes bundled with FastONN, can be used to train the ONN model with any choice of objective functions, optimization technique, learning rate, performance metrics and device. The metrics attribute takes a Python dictionary as input with metric name and calculation rule as key value pairs. Each metric’s calculation rule is a Python tuple which contains the function used to calculate the metric’s value, along with the criteria used for tracking the best state of the metric. In the example of Figure 4, the Trainer object would track the SNR value (‘snr’) and save the state of the model which maximizes the SNR for each data partition. Tracking of the loss function is performed by default. Furthermore, the save_all method is used to save all the statistics and model checkpoints in a single .pth file.

Figure 3 Code snippet illustrating the configuration of an ONN model using FastONN library.
6 Conclusions

In this paper, we introduce FastONN, the first open-source and GPU enabled Python-based implementation of Operational Neural Networks (ONNs). Based on a novel reformulation of the operational neuron, FastONN provides an easy-to-use interface to configure and train ONNs as end-to-end models, embed operational layers in contemporary network architectures and integrate novel application specific non-linear operator sets. This paves the way for independent research efforts and makes it easy to implement off-the-shelf ONN solutions for industrial and academic use. Future software updates will comprise of additional modules for optimal operator search and hyperparameter optimization.

References
[1] K. He, G. Gkioxari, P. Dollár, and R. Girshick, “Mask R-CNN,” IEEE Trans. Pattern Anal. Mach. Intell., 2020.
[2] K. He, X. Zhang, S. Ren, and J. Sun, “ResNet,” Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit., 2016.
[3] E. Shelhamer, J. Long, and T. Darrell, “Fully Convolutional Networks for Semantic Segmentation,” IEEE Trans. Pattern Anal. Mach. Intell., 2017.
[4] S. Kiranyaz, T. Ince, A. Iosifidis, and M. Gabbouj, “Operational neural networks,” Neural Comput. Appl., 2020.
[5] S. Kiranyaz, T. Ince, A. Iosifidis, and M. Gabbouj, “Generalized Model of Biological Neural Networks: Progressive Operational Perceptrons,” vol. 1, pp. 2477–2485, 2017.
[6] D. T. Tran, S. Kiranyaz, M. Gabbouj, and A. Iosifidis, “PyGOP: A Python library for Generalized Operational Perceptron algorithms,” Knowledge-Based Syst., 2019.
[7] D. T. Tran, S. Kiranyaz, M. Gabbouj, and A. Iosifidis, “Heterogeneous Multilayer Generalized Operational Perceptron,” IEEE Trans. Neural Networks Learn. Syst., 2020.
[8] S. Chetlur et al., “cuDNN: Efficient Primitives for Deep Learning,” Oct. 2014.
[9] A. Paszke et al., “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” in Advances in Neural Information Processing Systems 32, H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alchê-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8026–8037.
[10] Dougal Maclaurin, “Modeling, Inference and Optimization with Composable Differentiable Procedures,” Harvard Univ., 2016.

7 B- Required Metadata

7.1 B1 Current executable software version

Table 3 Software metadata

| Nr | (executeable) Software metadata description | Value |
|----|-------------------------------------------|-------|
| S1 | Current software version                  | 0.1.1 |
| S2 | Permanent link to executables of this version | [Link](https://github.com/junaidmalik09/fastonn/releases/tag/0.1.1) |
| S3 | Legal Software License                    | MIT   |
| S4 | Computing platform / Operating System     | Linux, OSX, Unix-like, Windows. |
| S5 | Installation requirements & dependencies  | Python, Anaconda, PyTorch |
| S6 | If available Link to user manual - if formally published include a reference to the publication in the reference list | [Link](https://github.com/junaidmalik09/onn_torched/README.md) |
| S6 | Support email for questions               | junaid.malik@tuni.fi |

7.2 B2 Current code version

Table 4 Code metadata

| Nr | Code metadata description | Please fill in this column |
|----|---------------------------|---------------------------|
| C1 | Current Code version      | 0.1.1                     |
| C2 | Permanent link to code / repository used of this code version | [Link](https://github.com/junaidmalik09/fastonn/releases/tag/0.1.1) |
| C3 | Legal Code License        | MIT                       |
| C4 | Code Versioning system used | Git                      |
|   | Software Code Language used |
|---|-----------------------------|
| C5 | Python                      |
| C6 | Compilation requirements, Operating environments & dependencies | torch, torchvision, numpy, scipy, PIL, glob, matplotlib, pathlib |
| C7 | If available Link to developer documentation / manual | https://github.com/junaidmalik09/fastmom/README.md |
| C8 | Support email for questions | junaid.malik@tuni.fi |