Fast Training of Convolutional Networks through FFTs

Vijay Aravindh Viswanathan (1219496837)

Rajesh Sathya Kumar (1219515401)

{ vviswa19, rsathyak }@asu.edu

Abstract:

Convolutional Neural Networks (CNN) have revolutionized the field of Computer Vision and pattern recognition. As the use of CNN's gets increasingly wide, the architectures that they employ also have become extremely complex over the last decade. The complex architecture and growing data requirements makes it computationally very expensive to run the algorithms, which may take weeks, if not months in several large-scale applications. In this project, we showcase an algorithm which significantly decreases the training time for any existing state-of-the-art CNN models. The algorithm involves computing convolutions as pointwise products in the fourier domain and also reusing the same map.

The algorithm will be implemented on a GPU architecture. We will be using the CIFAR-10 [2] dataset for demonstration of the running time analysis with and without using Fast Fourier Transform(FFT) in the convolutional layer.

This project is the work of Michael Mathieu, Mikael Henaff and Yann LeCun from the Courant Institute of Mathematical Sciences of NYU, cited in [1]. In this project, we do statistical analysis of how the algorithm scales for higher dimensional datasets.

Problem Statement:

Modern computer vision architectures predominantly employ the convolutional neural network. The training time of these networks increases as the architectures become more complex. The convolution operation is performed between a kernel and a 2-D matrix (input image/ output of previous layers etc). This becomes expensive as the number of layers and number of kernels in each layer increases. Modern problems incorporate millions of images as input to these networks. Modern networks use billions of parameters to train on the given data.

Any significant improvement in training time can bring about reductions in cost, emissions, compute resources etc.

Proposed Dataset:

We use the CIFAR-10 dataset. Below are the dataset specifications:

Training Data: 50000, *Test Data:* 10000, No Validation dataset

Number of classes: 10

Classes: 'plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'

Citation:

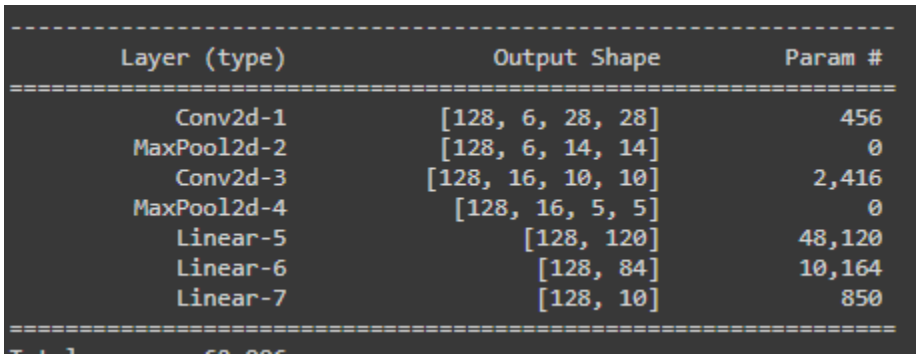
Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The

CIFAR-10 dataset. www.cs.toronto.edu/~kriz/cifar.html.

Source: https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz

Proposal Baselines and Evaluation Metrics:

Model Summary:



Layer (type)	Output Shape	Param #
Conv2d-1	[128, 6, 28, 28]	456
MaxPool2d-2	[128, 6, 14, 14]	0
Conv2d-3	[128, 16, 10, 10]	2,416
MaxPool2d-4	[128, 16, 5, 5]	0
Linear-5	[128, 120]	48,120
Linear-6	[128, 84]	10,164
Linear-7	[128, 10]	850

Batch Size - 128

We use the above model as a baseline and get an accuracy of 61%. The baseline training time stats in milliseconds without FFT algorithm are as below for each epoch:

Mini Batch Training Time (in ms)

	Num of minibatches	Min(in ms)	Max(in ms)	Mean(in ms)	Median(in ms)	Std Dev(in ms)
Epoch 1	391.000	19.775	96.575	44.199	40.729	7.567
Epoch 2	391.000	20.708	99.861	44.424	40.718	8.248

Implementation of FFT Convolutional Layer

The Goal of this project is to improve on the baseline training time with the same model using the FFT Convolution algorithm [1] in the CIFAR-10 dataset. In this project, we try to reduce the training time of the network while maintaining accuracy.

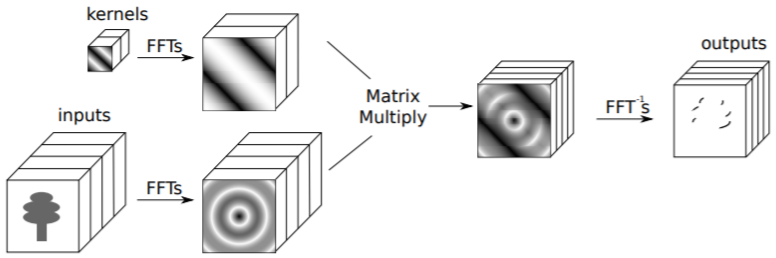


Figure 1: Illustration of the algorithm. Note that the matrix-multiplication involves multiplying all input feature maps by all corresponding kernels.

Illustration of the algorithm [1]

The need to compute FFT on each kernel and input increases the compute overhead. But the advantages become more pronounced as the input size, batch size and kernel size increase. We expect to see a decrease in the training time as the input and kernel size increase compared to the non-FFT convolutional neural network.

Fig 1. Illustrates the convolutional layer with the FFT implementation. The function accepts the input Tensor and kernel and transforms them in the fourier domain. Then, it performs a matrix multiplication on the two Tensors and then transforms back to the original domain by computing the Inverse Fourier transform. The prototype of the ***FFTConv*** class is given below. We have designed the parameters and implemented them in such a way that it can directly be used as a convolutional layer within the pytorch network.

This is a Higher order class which can be extended to any number of dimensions. But, we have only defined FFTConv1d, FFTConv2d, FFTConv3d in our project.

```python
class _FFTConv(nn.Module):
    """Base class for PyTorch FFT convolution layers."""

    def __init__(
        self,
        in_channels: int,
        out_channels: int,
        kernel_size: Union[int, Iterable[int]],
        padding: Union[int, Iterable[int]] = 0,
        stride: Union[int, Iterable[int]] = 1,
        groups: int = 1,
        bias: bool = True,
        ndim: int = 1,
    ):
```

Fig 2. Class Prototype of Convolution using FFT (can be used as a Convolutional Layer in Pytorch)

Proposal vs Experiments

We acknowledge that we have performed our training on a different dataset different from our proposal. This is due to the experimental requirement, we need to analyze data of different input sizes to test our algorithm, we used *'torch.dataset.FakeData'*. This provided the flexibility to run our experiments on different input sizes and the amount of dataset used

Experiments and Results

We broadly classify the experiments that we conducted on the convolutional algorithms into two major sections:

1. Functional Analysis
2. Neural Net training Analysis

1. Functional Analysis

Firstly, we have defined 3 custom convolutional functions conv1d, conv2d, conv3d which are not based on FFT. But, based on a naive windowing technique to convolve with 1D, 2D, and 3D tensors respectively. Then, we conducted 3 experiments to compare our naive convolution method and fft convolution method.

I. Input Size vs Execution Time
II. Kernel Size vs Execution Time
III. Dimension Size vs Execution Time

I. Input Size vs Execution Time

We tested the two methods by varying different input sizes and measured the execution times. We have used the following parameters for testing.

Input size:	3-300 in steps of 5
Kernel Size:	2
Dimension:	2 (Conv2D)
Total Runs:	20

Results:

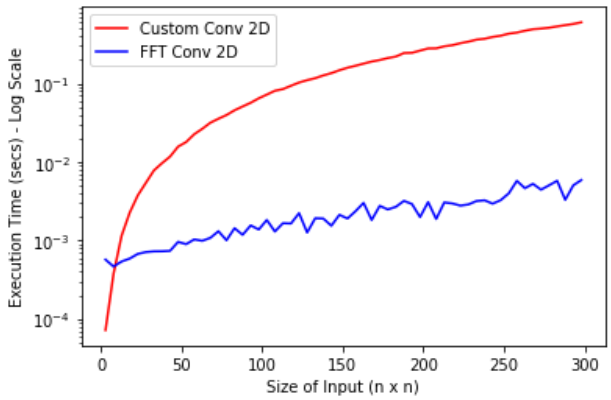


Fig 3. Functional Analysis: Input Size vs Execution Time - Results

We see that the custom conv2d implemented is faster only for a very small input size. But the fft_conv2d is easily faster for larger inputs and does not scale exponentially as with the windowing method.

II. Kernel Size vs Execution Time

We analyzed the two methods for different kernel sizes with the following parameters.

Input size:	150
Kernel Size:	2-50 in steps of 2
Dimension:	2 (Conv2D)
Total Runs:	20

Results:

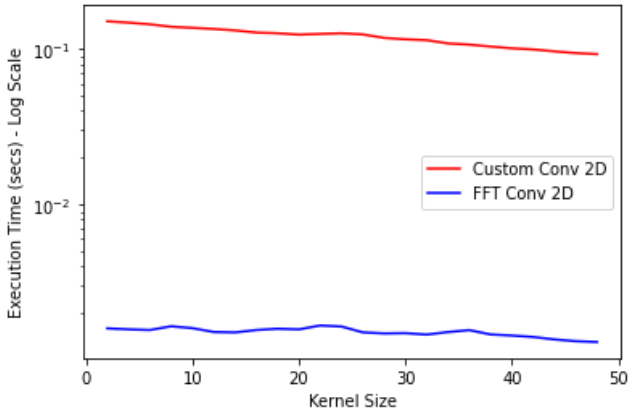


Fig 4. Functional Analysis: Kernel Size vs Execution Time - Results

This result was very interesting because we expected a similar rise for both the algorithms from the input size graph. But, here FFT algorithm is flat and low running time as expected, but the Naive approach is also flat because of the usage of numpy matrix multiplication methods which makes use of hardware acceleration to perform parallel linear algebra operations.

III. Dimension Size vs Execution Time

Dimension size of the inputs is tested against the 2 algorithms and running time is measured. Below are the parameters used for testing.

Input size:	50
Kernel Size:	2
Dimension:	1 (Conv1D), 2 (Conv2D), 3 (Conv3D)
Total Runs:	20

Results:

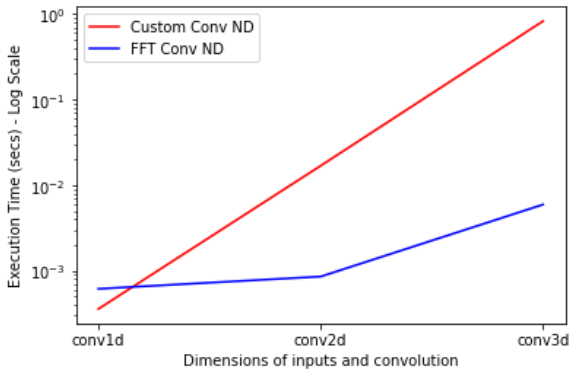


Fig 5. Functional Analysis: Kernel Size vs Execution Time - Results

This is an expected result in which naive conv2d performs better with low input dimensions and fft conv2d performs better for higher dimensions.

2. Neural Net training Analysis

We have used the below architecture for our analysis of the algorithm in a neural network. While this architecture was not the one which was initially proposed, this simple architecture enabled us to experiment with varied parameters to test the algorithm for the convolutional layer, which is the main purpose of the project. In this analysis, we tested the running time of training the CNN model with 3 different convolutional layers. They are 2D convolutional layer implementation using FFT, Mem Strided Im2Col Implementation and Pytorch version of Conv2d. Then, we perform similar experiments in this neural network architecture using these algorithms.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
          _FFTConv-1          [8, 16, 65, 65]         196,624
         MaxPool2d-2          [8, 16, 32, 32]               0
            Linear-3                [8, 120]       1,966,200
            Linear-4                 [8, 84]          10,164
            Linear-5                 [8, 10]             850
================================================================
Total params: 2,173,838
Trainable params: 2,173,838
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 1.50
Forward/backward pass size (MB): 5.14
Params size (MB): 8.29
Estimated Total Size (MB): 14.93
----------------------------------------------------------------
```

Fig 6. Architecture used for Neural Net training Analysis of our convolutional algorithms

Experimental dataset:

We test these algorithms using the *'torchvision.datasets.FakeData'* method. This allowed us to generate different sizes of image data for one of our experiments. This API was also very convenient in generating varied dataset sizes to test our algorithms.

I. **Input Size vs Running Times:**
We perform a similar experiment by applying the convolutional algorithms within a neural network architecture. Below are the parameters.

Dataset size	100
Batch size	10
Kernel size	64
Input sizes	70, 80, 90, 100, 110,128
Optimizer	SGD (with learning rate 0.001 and momentum 0.9)

Results:

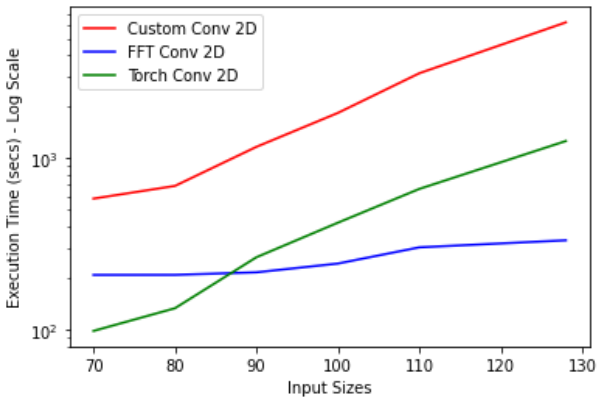


Fig 7. Neural Net Training Analysis - Input sizes vs Running time: Result

This is expected as the FFT convolutional running times are largely faster when the input image sizes grow. This is faster than the Pytorch and custom Conv 2d implementations. The overall growth of FFT is logarithmically constant while the other algorithms are logarithmically linear.

II. Kernel Size vs Running times:

We modify different kernel sizes and then train all of our models. Below are the parameters used:

Dataset size	128
Batch size	4
Kernel sizes	8,16,24, 32, 40, 48, 56, 64
Input size	128
Optimizer	SGD (with learning rate 0.001 and momentum 0.9)

Results:

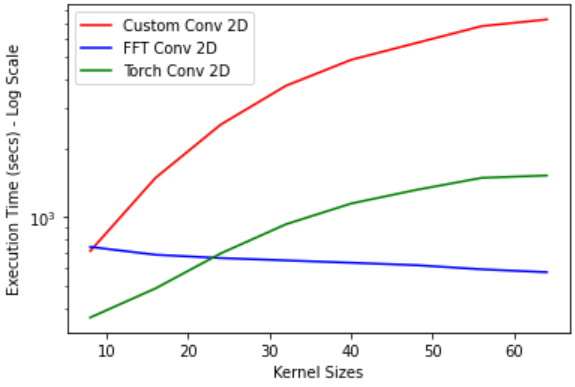


Fig 8. Neural Net Training Analysis - Kernel sizes vs Running time: Result

We can see that here also, the FFT version of Conv 2d works very well on larger kernel sizes, while the pytorch version performs better on lower kernel sizes

III. Batch Size vs Running Times:

We again experiment our architecture with different batch sizes. Below are the parameters used:

Dataset size	200
Batch sizes	16, 32,64,128,150, 180, 200
Kernel size	32
Input size	64
Optimizer	SGD (with learning rate 0.001 and momentum 0.9)

Result:

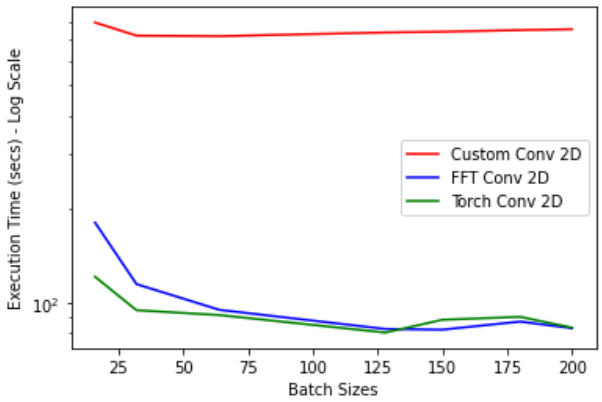


Fig 9. Neural Net Training Analysis - Batch size vs Running time: Result

This is the only experiment where we saw FFT performing on par with the Pytorch version of convolution algorithm for larger batch sizes. The Pytorch version even performs better on lower batch sizes.

Citations:

1. Mathieu, M., Henaff, M., & LeCun, Y. (2014). Fast training of convolutional networks through FFTS: International Conference on Learning Representations (ICLR2014), CBLS, April 2014. Paper presented at 2nd International Conference on Learning Representations, ICLR 2014, Banff, Canada.
2. Krizhevsky, A. (2009). Learning Multiple Layers of Features from Tiny Images. , , 32--33.

Other References:

1. Fast Training of Convolutional Neural Networks through FFTs, https://arxiv.org/pdf/1312.5851.pdf
2. How Are Convolutions Actually Performed Under the Hood? https://towardsdatascience.com/how-are-convolutions-actually-performed-under-the-hood-226523ce7fbf
3. FFT - Convolution operations using pytorch, https://github.com/fkodom/fft-conv-pytorch