Accelerating High-Resolution Tumor Detection in MRI Scans using Fast Fourier Transform-based Convolutional Neural Network

Manuel Thimoty Silalahi
NIM : 13524102
Program Studi : Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, JL. Ganesha 10 Bandung
13524102.std.stei.itb.ac.id, manuelsilalahi06@gmail.com

***Abstract*—Brain tumor detection utilizing Magnetic Resonance Imaging (MRI) is a of the most critical task in modern medical diagnostics that relies heavily on the performance of Machine Learning, like CNNs. While CNNs demonstrate superior accuracy, standard spatial convolution operations impose a quadratic computational complexity of $O(N^2K^2)$, creating a significant bottleneck, particularly with large kernel sizes. This paper analyzes the mathematical optimization of convolutional layers by transforming operations from the spatial domain to the frequency domain, also called as Discrete Fourier Transform. However, the time complexity taken to compute DFT in conventional way is still approximately $O(N^2)$. Therefore, Fast Fourier Transform came and do its magic, so that the transformation can be done in only $O(N \log N)$. This study also implements a CNN architecture from scratch using NumPy on a subset of the MRI Tumor dataset resized at a resolution of 128×128. Theoretical and experimental analyses demonstrate that the FFT-based approach reduces complexity to $O(N^2 \log N)$, offering significant computational efficiency improvements over naive convolution methods, thereby validating the vital role of vector basis transformation in Deep Learning optimization.**

I. Introduction

A. Background

Magnetic resonance imaging, or MRI, is a noninvasive medical imaging test that produces detailed images of almost every internal structure in the human body, including the organs, bones, muscles and blood. vessels [1]. The MRI machine is a large machine that creates strong radiowaves around the patient and sends pulses of radio waves from a scanner.

Tumor detection is a critical task in modern medical diagnostics. Among various approaches to achieve high detection accuracy, Deep Learning,specifically the Convolutional Neural Network (CNN) has emerged as the gold standard. Accurate tumor segmentation requires two contradictory features. There are a large receptive field to capture the global context of the organ, and high-resolution image inputs to preserve fine pathological details. Consequently, this necessitates the use of large kernel sizes within the CNN architecture. But, before moving up to large kernel sizes, there has to be an algorithm or framework that optimize the computation speed, even when the kernel is not that big enough. Therefore, this can lead to another visioner solution.

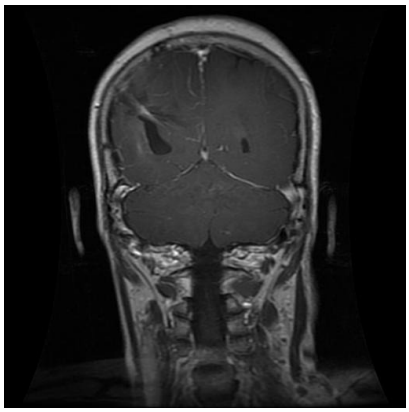

Fig. 1. Example of MRI Scans Result

B. Problem Formulation

Based on the background, this study addrresses the following mathematical and algorithm challenges :

1) **Computational Bottleneck.** Conventional way requires $O(N^2K^2)$. This becomes computationally ineffective, especially when we increase the size of the Kernel.
2) **Transformation Bottleneck.** Conventional way for computing a Discrete Fourier transform requires $O(N^2)$.
3) **Overhead.** We want to see the overhead effect on Fast Fourier Transform, especially when the size of the kernel is not big enough.

C. Objectives

1) To implement the Fast Fourier Transform (FFT) for the convolutional forward pass, leveraging the Convolution Theorem to transform the operation from a sliding window dot-product in the spatial domain to an element-wise multiplication in the frequency domain. This problem is also equivalence with analyzing time complexity of 2 polynomials.

2) To comparatively analysis the efficiency between CNNs that use spatial convolution method against FFT-based Convolution.
3) To analyse the effect of small kernel size on FFT - based method.

II. Theoretical Frameowrk

A. Convolution

In deep learning, especially computer vision, convolution is one of the most fundamental linear operations aiming to extract local features of input images. Consider kernel is a 'machine' to extract features from image pixel matrix. Therefore the result of convolution is arithmetic operations beetween this kernel and the input image matrix. For example, consider a 5×5 image and kernel size of 3×3, [4]

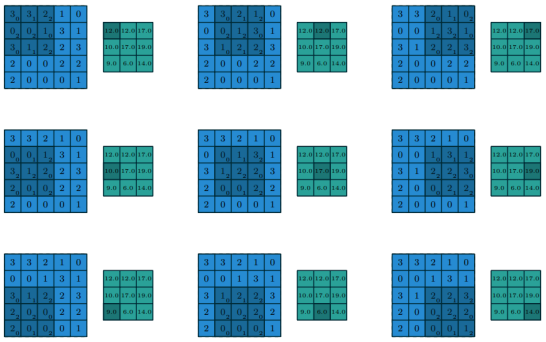

Fig. 2. Example of Discrete Convolution

Formally, discrete convolution operation in Deep Learning, especially CNN, where the kernel is not flipped also called as Cross Corelation. If the input images size I and the kernel size is $k \times k$, then the convolution $I * K$ is defined by

$$I * K(x, y) = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} I(x + i, y + i) \cdot K(i, j)$$

There are some properties that is used in Convolution Arithmetics.

1) *stride*. Stride value s_j is defined as distance between 2 consecutive kernel along axis j.
2) *zero padding*. Zero padding with value p_j is defined as number of zeros concatenated at the beginning and at the end of an axis j.
3) *filters*. Number of kernels.

Universal theorem that we'll be use is : for input size that have been padded i, kernel size k, and stride s, then the output size o is defined by

$$o = \left\lfloor \frac{i - k}{s} \right\rfloor + 1$$

We'll use this ultimate relationship to set the output size value.

B. Convolutional Neural Network

Artificial Neural Networks (ANNs) are computational processing systems of which are heavily inspired by way biological nervous systems (such as the human brain) operate. ANNs are mainly compromised with high amount of connecting nodes aiming to learn from input in order to optimise the output.

Meanwhile, Convolutional Neural Networks are analogous to the Traditional ANNs. Input of raw images will be converted to vector, then the neural networks will learn from the input images to adjusting the weights and biases in order to optimise the output and get the minimal error. final output. One of the example of CNNs Architecture can be seen below

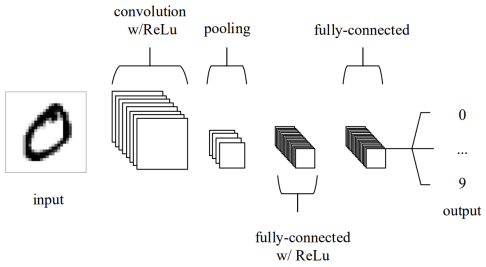

Fig. 3. Example of CNNs Architecture

From the above figure, we can see some example of the layers :

1) Input Layer : Layer that will hold the image pixels.
2) Convolutional : Layer that will do the Convolution Operation, between the weights (kernel) and the input.
3) Pooling Layer : Layer that will do downsampling along the spatial dimensionality of given input. This will reduce number of parameter.
4) Fully-connected Layer : A layer that will perform the same function as ANNs and attempt to produce class scores from the activation.

Therefore, CNNs activities are simply transforming the image more and more so that the neural network will 'learn' from them. This learning process will be divided into Forward Propagation and Backward Propagation.

Forward Propagation is a process through which input data will be transformed by a layer. For example, the convolutional layer will perform its forward propagation by receiveng input images in pixel matrix form, then will use the convolution operation to transform the values. Every type of layer will apply different type of transformation for each input images.

In the other hand, Backpropagation is process of learning through the error of the model. Then, it will calculate the gradient error and

C. Roots of Unity

In complex number, we know that

$$e^{i\theta} = \cos(\theta) + i \sin(\theta)$$

Substituting $\theta = 2\pi$ gives us

$$e^{2i\pi} = 1$$

Notice that we can find all of the root of $x^n = 1$ using this. Let k be any positive integer. Then

$$e^{2k\pi i} = 1^k = 1$$

It is given that $x^n = 1$, then

$$x^n = 1 = e^{2\pi i} = e^{4\pi i} = e^{6\pi i} = \cdots = e^{2k\pi i}$$

Raising each power $\frac{1}{n}$ gives us

$$x = e^{\frac{2\pi i}{n}} = e^{\frac{4\pi i}{n}} = e^{\frac{6\pi i}{n}} = \cdots = e^{\frac{2k\pi i}{n}}$$

Therefore, we can conclude that solutions of $x^n = 1$ is $x = e^{\frac{2k\pi i}{n}}$ with $k = 1, 2, \ldots, n$. This is also called as *Roots of Unity*.

D. Discrete Fourier Transform of Polynomials

Let $A(x)$ be a polynomial with degree $n-1$,

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

Consider a vector that represents the coefficients of this polynomial, $V = (a_0, a_1, \ldots, a_n)$. We know that the solution of $x^n = 1$ are on the form $w_{n,k} = e^{\frac{2k\pi i}{n}}$ with $k = 0, 1, 2, n-1$. Additionally these complex numbers have some very interesting properties: e.g. the principal n-th root $w_n = w_{n,1} = e^{\frac{2\pi i}{n}}$ can be used to describe all other n-th roots: $w_{n,k} = (w_n)^k$. Therefore, Discrete Fourier Transform of V is defined by the value of $A(x)$ evaluated at the points $x = w_{n,k} = (w_n)^k$,

$$(A(w_n^0), A(w_n^1), A(w_n^2), A(w_n^2), \ldots, A(w_n^{n-1})$$

Let's call this term as $DFT(V)$ or $DFT(A(x))$, the Discrete Fourier Transform of Vector V.

Now, Let there is an inverse of DFT, such that

$$IDFT(DFT(V)) = V$$

From those properties, we can see a polynomial multiplication from different perspective. Let 2 Vectors P, and Q, represents coefficient of Polynomial $A(x)$ and $B(x)$, respectively. Let C = $A(x)B(x)$ with the vector R represents the coefficient of polynomial C. Therefore, we can conclude that

$$DFT(R) = DFT(P \cdot Q) = DFT(P) \cdot DFT(Q)$$

And Also

$$P \cdot Q = IDFT(DFT(P) \cdot DFT(Q))$$

E. Fast Fourier Transform

Conservative way to compute polynomial multiplication is doing brute force. This time complexity is $O(n^2)$. But, Discrete Fourier Transform allows us to compute this in $O(n \log n)$. This method is called as **Fast Fourier Transform**. Suppose polynomials $A(x)$ with degree of $n-1$, where n is the power of 2, i.e. there are integer k such that $2^k = n$. If n is not the power of 2, we simply add missing term $a_i x_i^i$ and set the coefficient to 0.

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

Then, we divide $A(x)$, into 2 components. The first one only contains the coefficient of even positions and the other one on the odd positions.

$$A(x) = A_0(x^2) + xA_1(x^2)$$

Note that the size of the polynomial A_0 and A_1 is now $\frac{n}{2}$.Assume that we have computed the $DFT(A_0)$ and $DFT(A_1)$

$$DFT(A_0) = \begin{bmatrix} y_0^0 \\ y_1^0 \\ y_2^0 \\ \vdots \\ y_{\frac{n}{2}-1}^0 \end{bmatrix} \qquad DFT(A_1) = \begin{bmatrix} y_0^1 \\ y_1^1 \\ y_2^1 \\ \vdots \\ y_{\frac{n}{2}-1}^1 \end{bmatrix}$$

where y_i^0 is the value of A_0 evaluated at $w_{\frac{n}{2},i}$ and y_i^1 is the value of A_1 evaluated at $w_{\frac{n}{2},i}$. Now, see that, for the first $\frac{n}{2}$ values, since $A(x) = A_0(x^2) + xA(x^2)$, then

$$y_k = y_k^0 + w_n^k y_k^1$$

for $k = 0, 1, 2, \ldots, \frac{n}{2} - 1$. But, for the second $\frac{n}{2}$ value,

$$y_{k+\frac{n}{2}} = A\left(w_n^{k+\frac{n}{2}}\right) \tag{1}$$

$$= A_0\left(w_n^{2k+n}\right) + w_n^{k+\frac{n}{2}} A_1\left(w_n^{2k+n}\right) \tag{2}$$

$$\tag{3}$$

Because $w_n^n = w_{n,n} = e^{\frac{2n\pi i}{n}} = e^{2\pi i} = 1$. Use same method we can also find that $w_n^{\frac{n}{2}} = -1$. Therefore, the last expression is equivalence to

$$y_k = A_0(w_n^{2k}) - w_n^k A_1(w_n^{2k}) \tag{4}$$

Notice that $w_n^{2k} = w_{\frac{n}{2}}^k$, therefore $A_0(w_n^{2k}) = y_k^0$ and $A_1(w_n^{2k}) = y_k^1$. Therefore, the final expression of$y_{k+\frac{n}{2}}$ yields

$$y_{k+\frac{n}{2}} = y_k^0 - w_n^k y_k^1$$

. Thus, we can compute the whole vector $y_k = DFT(A)$,

$$y_k = y_k^0 + w_n^k y_k^1 \quad k = 0, 1, \ldots, \frac{n}{2} - 1$$

$$y_{k+\frac{n}{2}} = y_k^0 - w_n^k y_k^1 \quad k = 0, 1, \ldots, \frac{n}{2} - 1$$

How about the time complexity? Note that, if we can compute DFT(A) in $O(n)$ time using DFT(A_0) and DFT(B_0), then we have $T_{DFT}(N) = 2T_{DFT}(\frac{n}{2}) + O(n)$. This equivalence to

Algorithm 1 Fast Fourier Transform

Require: $A = [a_0, a_1, \ldots, a_{n-1}]$
  $N \leftarrow n$
  if $N \leq 1$ **then**
    return A
  end if
  $Even \leftarrow \text{FFT}(A\left[0, 1, \ldots, \frac{n}{2} - 1\right])$
  $Odd \leftarrow \text{FFT}(A\left[\frac{n}{2}, \frac{n}{2} + 1, \ldots, n - 1\right])$
  $k \leftarrow \frac{N}{2}$
  $t \leftarrow e^{\frac{2k\pi i}{N}} \cdot Odd$
  return concat$([Even + t, Even - t])$

$T_{DFT}(N) = O(N \log N)$. Thus, the complexity of computing y_k is $O(n \log n)$, faster than computing the DFT on naive brute force .

We also can compute the imverse of FFT. Let $y = [y_0, y_1, \ldots, y_{n-1}]$ be the DFT of $A(x)$. Using Matrix Inverses, we get that

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j w_n^{-kj}$$

. Now, let's see the formula of y_k,

$$y_k = \sum_{i=0}^{n-1} a_i w_n^{ki}$$

Notice that these formulas are almost the same. So, one of the way to found coefficients a_k is to do the divide and conquer algorithms, as well as the direct FFT. The only difference is that instead w_n^k, we use w_n^{-k}. . Thus, the time complexity is almost the same as the time complexity of direct FFT, $O(n \log n)$.

F. The Correlation between Fast Fourier Transform and Convolution Theorem

Let $M \in \mathbb{R}^{N \times N}$ denote the input image matrix representing pixel intensities, and let $K \in \mathbb{R}^{k \times k}$ denote the convolutional kernel matrix. In the spatial domain, the convolution operation $Y = M * K$ requires sliding the kernel K over every pixel of M, resulting in a computational complexity of $O(N^2 k^2)$.

The Convolution Theorem provides a fundamental bridge between the spatial domain and the frequency domain. It states that the convolution of two signals in the spatial domain is mathematically equivalent to the element-wise multiplication of their Fourier Transforms in the frequency domain. Formally, this relationship is expressed as:

$$\text{DFT}(M * K) = \text{DFT}(M) \odot \text{DFT}(K) \tag{5}$$

Where:

- $*$ denotes the convolution operation.
- $\odot$ denotes the element-wise product.

Notice that when we apply the Inverse Fourier Transform (IDFT), then

$$M * K = \text{IDFT}\left(\text{DFT(M)} \odot \text{DFT}(K)\right) \tag{6}$$

Remember how to compute the DFT in the efficient way? While a naive DFT computation takes $O(N^2)$, the FFT reduces this to $O(N \log N)$. Consequently, by utilizing the Convolution Theorem via FFT, the operation is transformed from a computationally expensive sliding window process into a sequence of three efficient steps: And since this task is element wise multiplication, the problem is similar to polynomial multiplication. This correlation forms the theoretical basis for the speedup observed in large-scale kernel operations.

III. Analytical Steps

A. Preparing Dataset

To evaluate the comparative performance of the spatial domain convolution versus the frequency domain (FFT) optimization, this study utilizes the "MRI Scans of Tumor Detection" dataset, which is publicly available on the Kaggle repository [2].

The dataset consists of high-resolution Magnetic Resonance Imaging (MRI) scans categorized into four distinct classes based on the tumor pathology:

1) Glioma Tumor: Tumors that occur in the brain and spinal cord.
2) Meningioma Tumor: Tumors that form on the membranes that cover the brain and spinal cord.
3) Pituitary Tumor: Tumors that form in the pituitary gland.
4) No Tumor: Control samples containing healthy brain scans

While the original dataset contains thousands of images, this study enforces a strict subset constraint due to the nature of the implementation. Since the convolutional layers are implemented from scratch using fundamental linear algebra operations (Python/NumPy) without using of optimized low-level backend libraries (e.g., CUDA/C++), the computational cost for standard spatial convolution is extremely high.Therefore, to maintain training time on a standard CPU while sufficient for demonstrating the algorithmic optimization of FFT, we utilize a balanced random subset of 80 images from the total dataset. This subset is distributed equally across the four classes (20 images per class) and preprocessed to a uniform resolution of 128×128 pixels to ensure consistent matrix dimensions during linear transformations.

B. Polynomial Multiplication Benchmark

To empirically validate the application of Fast Fourier Transform, especially polynomial multiplication, we conduct a test for on 1D polynomial multiplication. The experimental steps is as follows :

1) Generate 2 random polynomials $A(x)$ and $B(x)$ with size of N each. We'll do the test with some different N values.
2) We'll do spatial computation. This method iterates through every pair of a_i, b_j to compute the resulting coefficient c_{i+j}.This represents the spatial domain convolution with a theoretical complexity of $O(N^2)$.
3) Then, we'll use the FFT Divide and Conquer Algorithm to compute the result of the multiplication. First, we'll

apply $A(x)$ and $B(x)$ with Discrete Fourier Transform using FFT. This can be done in $O(n \log n)$ time. Then, We'll multiply each corresponding value. This can be done in $O(n)$ time. Therefore, the theoretical time complexity for this method is $O(n \log n)$.

```
def multiplypolinoms(p1,p2):
    # p1 and p2 contains array of
        coefficients
    target_length = len(p1) + len(p2) -1

    n = 1
    while(n < target_length) :
        n*=2
    new_p1 = np.zeros(n)
    new_p2 = np.zeros(n)

    new_p1[:len(p1)] = p1
    new_p2[:len(p2)] = p2

    # Convert to list of value DFT
    fft_p1 = fft_recursive(new_p1)
    fft_p2 = fft_recursive(new_p2)
    multiple = fft_p1 * fft_p2
    coeff = inverse_fft(multiple)
    coeff = coeff[:n-1]
    return coeff
```

4) We measure the execution time for both methods.

C. Constructing the Layers

As previously discussed, the proposed CNN architecture comprises several distinct computational layers.

For the Convolutional Layer, we define a class initialized with the following parameters: number of filters, input channel size, kernel size, padding, and stride. This class implements four key methods: spatial forward propagation, spatial backward propagation, FFT-based forward propagation, and FFT-based backward propagation.

```
class ConvLayer():
    def __init__(self, num_filters,
        input_channel, kernel_size,
        padding=0, stride=1):
        self.num_filters = num_filters
        self.input_channel = input_channel
        self.kernel_size = kernel_size
        self.padding = padding
        self.stride = stride
        self.weights =
            np.random.randn(num_filters,
            input_channel, kernel_size,
            kernel_size) # random init for
            weights
        self.bias = np.zeros(num_filters)
        self.input_padded = None
```

For downsampling, we utilize a Max Pooling layer. The class accepts two parameters: pool size and stride, both defaulting to 2. Similar to the convolutional layer, this class includes methods for both forward and backward propagation.

```
class MaxPoolingLayer:
    def __init__(self, pool_size=2,
        stride=2):
        self.pool_size = pool_size
        self.stride = stride
        self.input = None
        self.max_indices = None # Simpan
            posisi max value
```

Finally, the Dense (Fully Connected) Layer is defined by two arguments: the size of the input vector and the size of the output vector.

```
class DenseLayer:
    def __init__(self, input_size,
        output_size):
        self.input_size = input_size
        self.output_size = output_size
        self.weights =
            np.random.randn(input_size,
            output_size) * 0.01
        self.bias = np.zeros((1, output_size))
```

D. Constructing the CNN Architecture

Having defined the individual component classes, we now assemble the final neural network structure. The proposed model architecture is organized sequentially as follows:

1) **Input Layer:** Accepts input images with dimensions of $128 \times 128 \times 3$.
2) **Convolutional Layer 1:** Configured with 8 filters, a kernel size of 3×3, and padding of 1.
3) **Activation:** ReLU (Rectified Linear Unit).
4) **Convolutional Layer 2:** Configured with 16 filters, a kernel size of 3×3, and padding of 1.
5) **Activation:** ReLU (Rectified Linear Unit).
6) **Pooling Layer:** Max Pooling with a pool size of 2 and a stride of 2.
7) **Dense (Output) Layer:** A fully connected layer mapping the flattened feature maps to the 4 target classes.

```
class TumorDetectionCNN:
    def __init__(self, num_classes=4):
        self.num_classes = num_classes
        self.conv1 = ConvLayer(num_filters=8,
            input_channel=3, kernel_size=3,
            padding=1)
        self.relu1 = ReLU()
        self.pool1 =
            MaxPoolingLayer(pool_size=2,
            stride=2)
        self.conv2 =
            ConvLayer(num_filters=16,
            input_channel=8, kernel_size=3,
            padding=1)
        self.relu2 = ReLU()
        self.pool2 =
            MaxPoolingLayer(pool_size=2,
            stride=2)
        self.flatten_size = 16 * 32 * 32 #
            16384
        self.dense =
            DenseLayer(self.flatten_size,
            num_classes)
```

E. Hyperparameters and Experimental Setup

We'll run the experiment to see the difference of time taken between spatial and FFT method. To evaluate the proposed architecture, we configure the training process with specific hyperparameters designed for stability on the limited dataset.

- **Loss Function:** We utilize Categorical Cross-Entropy Loss to measure the discrepancy between the predicted probability distribution and the true labels.
- **Optimizer:** We implement standard Stochastic Gradient Descent (SGD) manually, with a learning rate set to $\alpha = 0.01$.
- **Training Config:** The model is trained for 10 epochs with a mini-batch size of 16. The small batch size is chosen to manage memory overhead during matrix operations on the CPU.
- **Initialization** : Don't forget to ensure that the initiated weights and biases for the spatial and fft method are the same. This can be done by copying the weights and biases from spatial to fft before running the experiment.

Since this study focuses on algorithmic efficiency (Big-O analysis) rather than hardware acceleration, all experiments are conducted on a standard environment without GPU support:

- **Processor:** 13th Gen Intel(R) Core(TM) i5-13450HX (16CPus) @ 2.4GHz
- **RAM:** 12 GB
- **Software Stack:** Python 3.13.0 and NumPy 2.3.4.

This setup ensures that the recorded execution times purely reflect the computational complexity of the Spatial vs. FFT convolution algorithms.

IV. Result and Analysis

A. Polynomial Multiplication

We'll take some sizes : 100,500,1000,2000,3000,5000. To validate our theoretical assumption, we'll do the polynomial multiplication with differen polynomial sizes given above. We can see the result below. As we have seen, a small

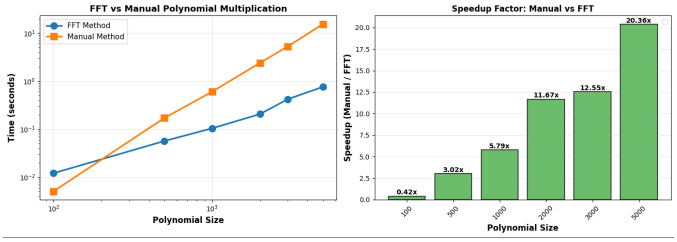

Fig. 4. Polynomial Multiplication Result Chart

input of N can make the spatial/manual method to do faster then FFT-based. The spatial method only required **0.005s**, while FFT-based took **0.012**. This phenomenon is attributed to the computational overhead inherent in the FFT algorithm, especially when N is so small.

However, the advantage of using FFT-based method can be seen with the sizes after $N = 100$. At $N = 500$, the FFT method overtakes the spatial method with a speedup of 3.02x. As N scales to 1000, the gap widens significantly, yielding a 5.79x speedup. As the largest tested size of $N = 5000$, the spatial method degrades drastically, while FFT-based method achieved massive speed up of 20.36x.

These experimental results align perfectly with our theoretical analysis about time complexity of both spatial and FFT-based method. We know that the spatial method has time complexity of $O(N^2)$, while the FFT-based method has time complexity of $O(n \log n)$. And since $\log n$ grows slower than n, especially when n is big, we can conclude when N is big, substansial, the speed up can also be massive.

B. Spatial VS FFT-based CNN

The result of the test can be seen from chart below.

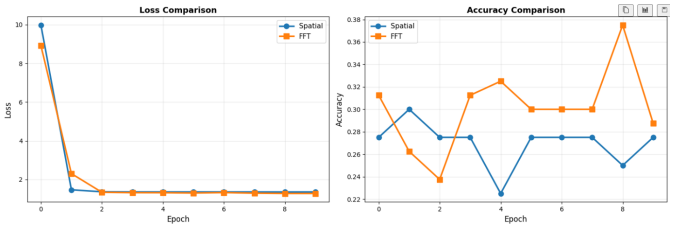

Fig. 5. Loss and Accuracy Chart

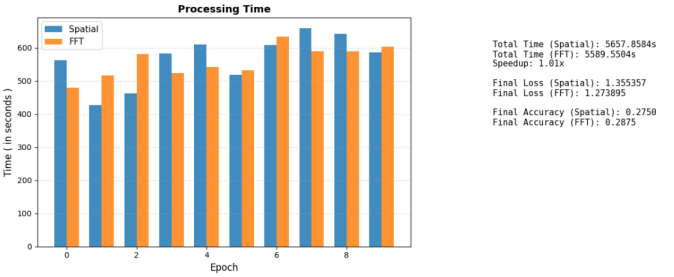

Fig. 6. Preprocessing Time for Both Spatial and FFT-based method

It is notable that the elapsed execution time for both the spatial and FFT-based methods does not differ significantly. In fact, the spatial method nearly matches the performance of the FFT-based method. Specifically, the total execution time for the spatial method was 5657.8584 s, whereas the FFT-based method required 5589.5504 s. Consequently, the FFT-based method outperformed the spatial approach by a marginal speedup factor of $1.01\times$.

From these results, we can conclude that our theoretical analysis regarding the algorithmic overhead of the FFT-based method is empirically valid. The kernel size selected for this project (3×3) is likely too small for the FFT-based approach to fully demonstrate its computational advantage. However, even with this small kernel size, the FFT method still achieved slightly superior performance compared to the spatial method. This suggests that the Fast Fourier Transform is particularly

well-suited for MRI analysis tasks involving high-resolution images that necessitate larger kernel sizes.

V. Conclusion

This study successfully demonstrates the critical role of Linear Algebra, specifically the Convolution Theorem and Basis Transformation, in optimizing deep learning architectures. By implementing a Convolutional Neural Networks (CNN) from scratch without reliance on optimized backend libraries, we explicitly isolated the mathematical operations governing feature extraction.

Our theoretical analysis and empirical benchmarks lead to several key conclusions:

1) **Mathematical Verification:** The Fast Fourier Transform (FFT) successfully reduces the complexity of the convolution operation from quadratic $O(N^2K^2)$ to log-linear $O(N^2 \log N)$. This was empirically validated through the 1D polynomial multiplication benchmark, where the FFT method achieved a massive **20.36x speedup** at $N = 5000$.
2) **Domain Transformation Efficiency:** Transforming input matrices from the spatial basis to the frequency basis DFT is proved to be efficient for high-dimensional data. While there is a computational overhead for small inputs ($N < 100$), the benefits become exponential as image resolution increases, making it a viable solution for high-resolution MRI processing. Also, notice that there is some overhead in FFT - based method, especially when the kernel is not big enough. This can be optimized by increasing the size of the kernel.

VI. Future Work

This research highlights the implementation and using of FFT-based Convolution in CNN. But, there are several limitations suggest directions for future investigation :

1) **Memory Complexity Optimization:** Although FFT reduces time complexity, it increases space complexity. The 'storage' of complex numbers and the padding requirements for linear convolution significantly increase memory usage. Future studies should explore **Overlap-Save** methods or real-valued FFT algorithms (rFFT) to mitigate this memory overhead.
2) **Dealling with Overhead :** It is observed that for small kernel sizes, specifically 3×3, the FFT-based approach does not yield a significant speedup and may theoretically match or slightly lag behind the spatial method. This is due to the fixed algorithmic overhead of the Fourier Transform (computation of roots of unity and complex number arithmetic). The advantage of FFT, described by $O(N^2 \log N)$ versus $O(N^2K^2)$, only becomes dominant when the quadratic term K^2 is sufficiently large enough to 'win' the logarithmic constant. This can be done by increasing the kernel size.
3) **Hardware Acceleration:** This study was constrained to CPU-based execution to demonstrate algorithmic efficiency. Implementing these linear algebra stacks required more computational power and the power of paralel computing. This can be done with useful libraries/engine, for example **CUDA**, one of the features from **Pythorch** or **Tensorflow** libraries. Those libraries probably the two most well-known library for Machine Learning.

Acknowledgment

Links

The project repository can be viewed here
Short explanation can be viewed here

References

[1] https://www.hopkinsmedicine.org/health/treatment-tests-and-therapies/magnetic-resonance-imaging-mri
[2] https://www.kaggle.com/datasets/masoudnickparvar/brain-tumor-mri-dataset
[3] https://cp-algorithms.com/algebra/fft.html.
[4] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv preprint arXiv:1603.07285*, 2018.
[5] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.
[6] 3Blue1Brown Youtube Channel : https://www.youtube.com/@3blue1brown
[7] Reducible Youtube Chnnael : https://www.youtube.com/@reducible
[8] https://www.geeksforgeeks.org/data-science/how-to-perform-faster-convolutions-using-fast-fourier-transformfft-in-python/

VII. Pernyataan

Dengan ini saya mengatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung. 24 Desember 2025

Manuel Thimoty Silalahi