FAST FOURIER TRANSFORM

Yash Gutgutia **Pritesh Palod**

180001064 **180001038**

Computer Science and Engineering

Algorithm Design and Analysis Lab (CS 254)

Second Year

Under the guidance of

Dr. Kapil Ahuja

Department of Computer Science and Engineering

Indian Institute of Technology Indore

Spring 2020

Table of Contents

Introduction **3**

Fourier Transform **4**

Discrete Fourier Transform **5**

Fast Fourier Transform **8**

Need and History 8

Algorithm Analysis and Design 9

Algorithm Implementation 14

Complexity Analysis **15**

Applications in various Fields **16**

References **17**

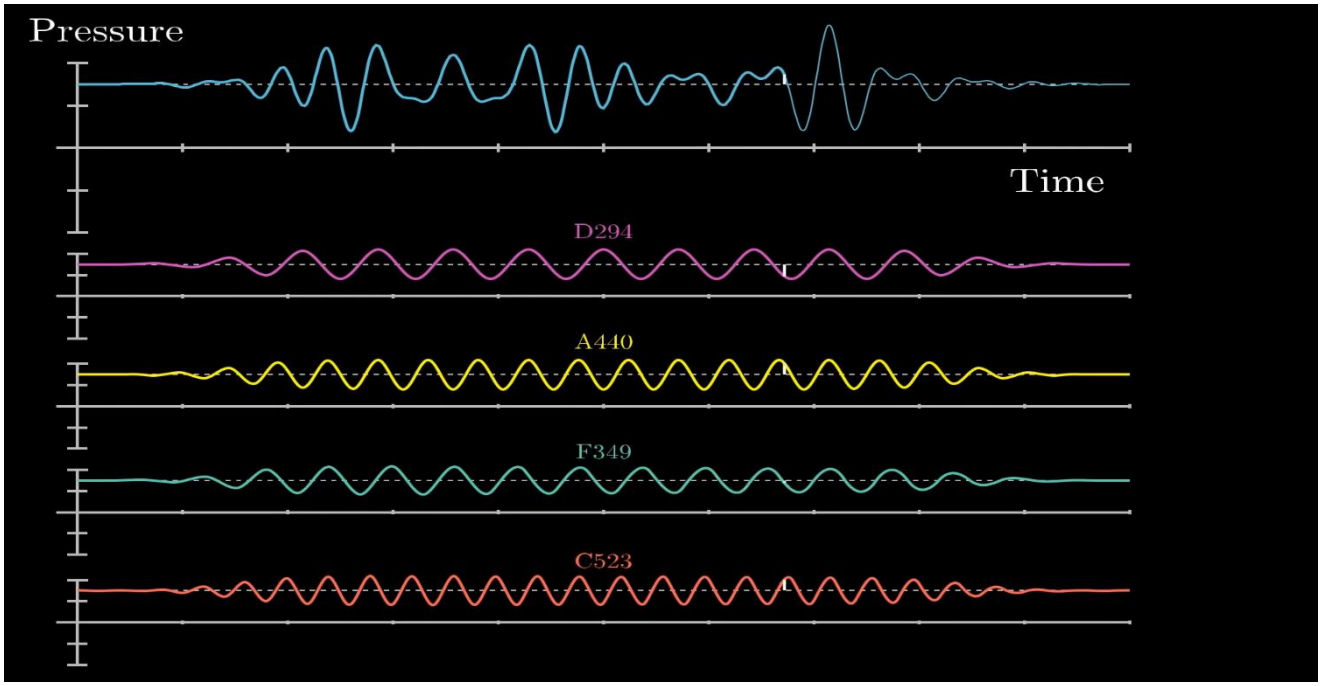
Introduction

The Fourier Transform is a mathematical technique that transforms a function of time, $g(x)$, to a function of frequency, $F(\omega)$. The Fourier transform of a function of time is itself a complex-valued function of frequency, whose magnitude represents the amount of that frequency present in the original function, and whose argument is the phase offset of the basic sinusoid in that frequency. The Fourier transform is not limited to functions of time, but the domain of the original function is commonly referred to as the time domain.

When a signal is discrete and periodic, we don't need the continuous Fourier transform. Instead we use the discrete Fourier transform, or DFT.

The FFT is a fast algorithm for computing the DFT. Let's discuss a bit about the Fourier Transformation and the Discrete Fourier Transformation, before diving into the Fast Fourier Transformation.

Fourier Transform



Picture taken for illustration purposes from the YouTube Channel 3Blue1Brown

In the example above, we have Pressure vs Time curve of a few Sound waves. The 2nd, 3rd, 4th and 5th graphs represent the Pressure vs Time curves of 4 different pure sine waves. The topmost curve represents the resultant pressure vs Time graph when the 4 pure sine waves are combined. The pressure at various points sometimes add up to create a larger pressure than any of its individual components, and sometimes they cancel each other out, to create a point with no pressure in space at a particular time.

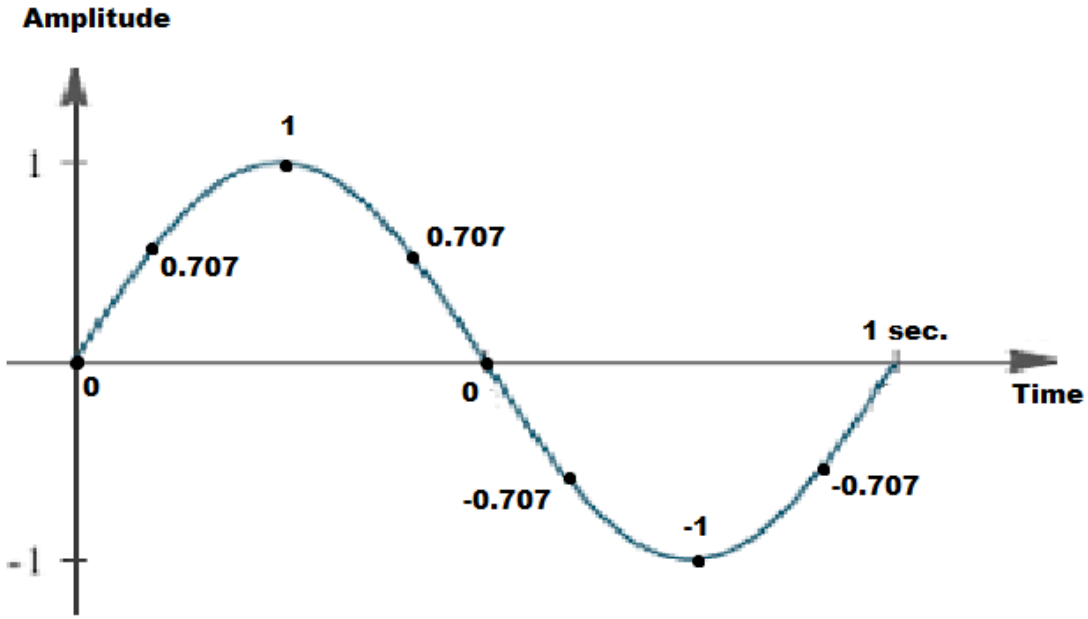
As the number of combining waves increases, the resultant function gets more complex, which makes it difficult to obtain the individual components from the resultant function. The Fourier Transformation and Inverse Fourier Transform together helps us to analyse the resultant signal as a function of frequency and identify the original constituent waves producing the complex function.

The Fourier transform (FT) of the function f(x) is the function F(ω), where: $F(\omega) = \int_{-\infty}^{+\infty} f(x) e^{-i\omega x}\ dx$

Discrete Fourier Transformation (DFT)

When we are given a discrete set of points instead of a continuous function, we take the help of the Discrete Fourier Transformation to calculate the same. The Discrete Fourier Transform (DTF) can be written as follows:

$$x[k] = \sum_{n=0}^{N-1} x[n] e^{-\frac{i2\pi kn}{N}}$$



Curve drawn using paint for illustration

Let's take an example of a sine wave with unit amplitude and a unit Hertz Frequency, where we are given 8 discrete points as marked in the above example.

In order to calculate the Fourier Transformation of the function, we need to calculate the frequency bins, x[k] for k varying from 0 to 7.

$$x[k] = \sum_{n=0}^{N-1} x[n] e^{-\frac{i2\pi kn}{N}}$$

1 sample: $F_0 = x_0 * \text{exponential}$

2 samples: $F_0 = x_0 * \text{exponential} + x_1 * \text{exponential}$

$F_1 = x_0 * \text{exponential} + x_1 * \text{exponential}$

3 samples: $F_0 = x_0 * \text{exponential} + x_1 * \text{exponential} + x_2 * \text{exponential}$

$F_1 = x_0 * \text{exponential} + x_1 * \text{exponential} + x_2 * \text{exponential}$

$F_2 = x_0 * \text{exponential} + x_1 * \text{exponential} + x_2 * \text{exponential}$

Computationally there are n multiplications for each of the 'n' terms, assuming that the exponential terms are already calculated. So, the algorithmic complexity for the DFT via classical method turns out to be $O(N^2)$.

To determine the DTF of a discrete signal x[n] (where N is the size of its domain), we multiply each of its value by e raised to some function of n. We then sum the results obtained for a given n. If we used a computer to calculate the Discrete Fourier Transform of a signal, it would need to perform $O(N^2)$ operations.

Here is a snippet of code that shows the calculation of the Fourier Transformation via the DFT expression.

```
/*--------------------------------------------------------------------------*/


void dft(complex<double> x[],ll n,complex<double> f[])
{
    complex<double> w(cos((-2*PI)/n),sin((-2*PI)/n));
    complex<double> w2(1,0);
    for(int i=0;i<n;i++)
    {
        f[i]=x[0];
        complex<double> w3=w2;
        for(int j=1;j<n;j++)
        {
            f[i]+=x[j]*w3;
            w3*=w2;
        }
        w2*=w;
    }
}


/*--------------------------------------------------------------------------*/
```

From the code snippet also, it is very clear that the brute force implementation is of order $O(N^2)$.

Fast Fourier Transformation (FFT)

Need and History

Since the brute-force computation of the DFT of length N requires $O(N^2)$ operations, and we need to use the DFT of long lengths for many practical applications. The computational requirement in that case becomes very high. Due to such high computational requirement of brute-force computation of DFT, it was not possible to use that for real-time and online Digital Signal Processing applications in the early 1900s.

In 1965, when Cooley and Tukey developed the famous Fast Fourier transform (FFT) algorithm, it became possible to reduce the operation count of DFT from $O(N^2)$ to $O(N\log_2 N)$, for a DFT of length N. They presented an efficient algorithm based on Divide and Conquer approach in order to compute the DFT. Divide and conquer approach was applied to the DFT recursively, such that a DFT of any size $N = N_1N_2$ was computed in terms of smaller DFTs of sizes N_1 and N_2.

Assuming that each operation takes 1 minute to compute (just for comparison purpose), if we have a signal of frequency 20000 Hz with 1 Hz spacing, the DFT performs $(20000)^2$=400 million operations, which would take about 760 years, while the FFT performs only $20000*\log_2 20000$= 28600 operations substantially reducing the days to compute the transformation to a mere 198 days.

Algorithm Analysis and Design

The computational complexity of DFT is substantially reduced by using the following trigonometric symmetry and periodicity of the twiddle factor $e^{-\frac{i2\pi km}{N/2}}$.

Symmetric Identities:

$$\cos\left(-\frac{2\pi km}{N/2}\right) = \cos\left(-\frac{2\pi(N/2+k)m}{N/2}\right)$$

$$\sin\left(-\frac{2\pi km}{N/2}\right) = \sin\left(-\frac{2\pi(N/2+k)m}{N/2}\right)$$

$$e^{-\frac{i2\pi km}{N/2}} = e^{-\frac{i2\pi(N/2+k)m}{N/2}}$$

For k: 0, 1, ……., N

Here we can notice that $e^{-\frac{i2\pi km}{N/2}}$ is periodic for k with a period of N/2.

This periodicity property later helps us to reduce the number of operations in each iteration from N to N/2 which apparently plays a crucial role in reducing down the complexity.

Suppose, we separated the Fourier Transform into even and odd indexed sub-sequences as such:

$$\begin{cases} n = 2r & \text{if } even \\ n = 2r+1 & \text{if } odd \end{cases}$$

$$\text{where} \quad r = 1, 2, ..., \frac{N}{2} - 1$$

After performing a bit of algebra, we end up with the summation of two terms. The advantage of this approach lies in the fact that the even and odd indexed sub-sequences can be computed concurrently.

$$x[k] = \sum_{n=0}^{N-1} x[n] e^{\frac{-j2\pi kn}{N}}$$

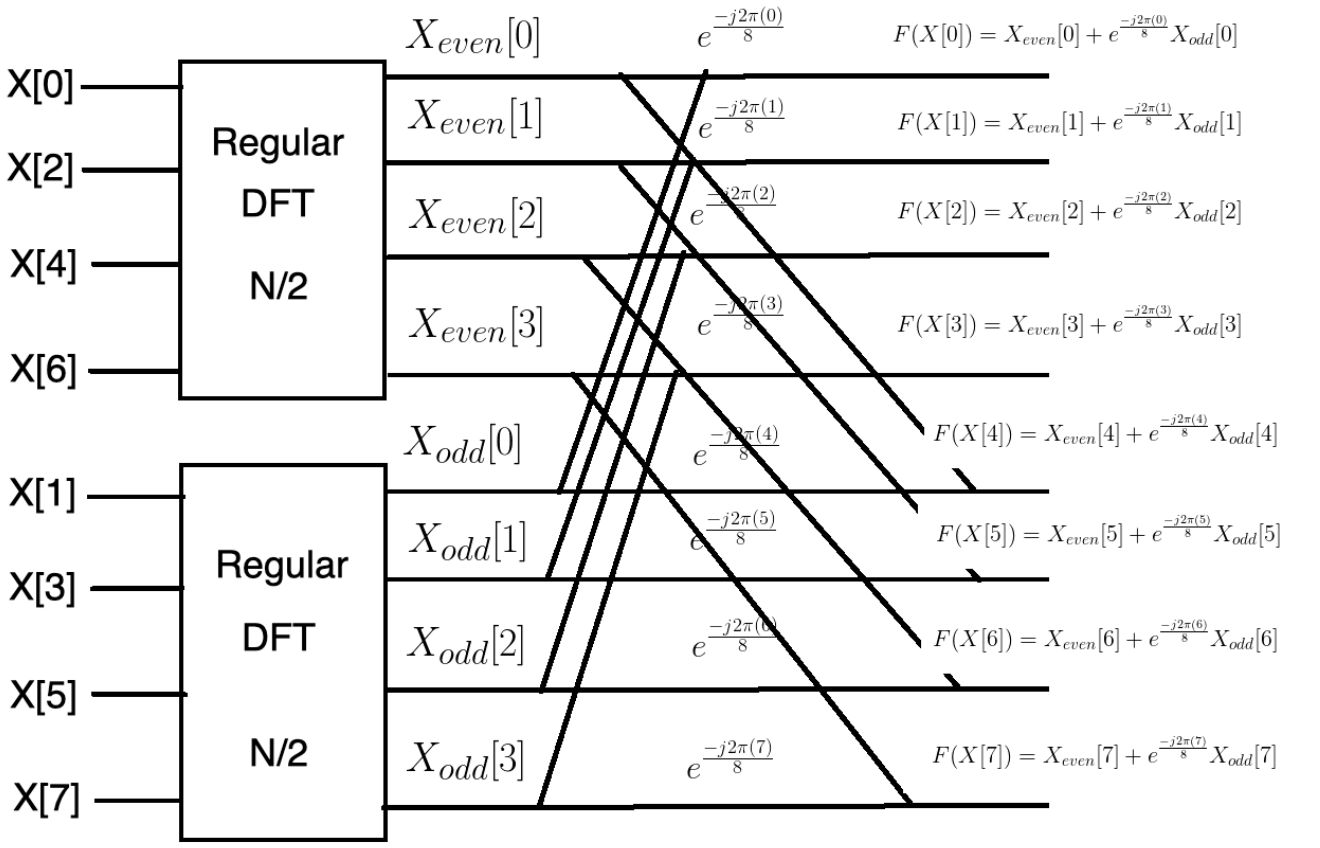
$$x[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r] e^{\frac{-j2\pi k(2r)}{N}} \quad + \quad x[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r+1] e^{\frac{-j2\pi k(2r+1)}{N}}$$

$$x[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r] e^{\frac{-j2\pi k(2r)}{N}} \quad + \quad x[k] = e^{\frac{-j2\pi k}{N}} \sum_{r=0}^{\frac{N}{2}-1} x[2r+1] e^{\frac{-j2\pi k(2r)}{N}}$$

$$x[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r] e^{\frac{-j2\pi k(r)}{N/2}} \quad + \quad x[k] = e^{\frac{-j2\pi k}{N}} \sum_{r=0}^{\frac{N}{2}-1} x[2r+1] e^{\frac{-j2\pi k(r)}{N/2}}$$

$$x[k] = x_{even}[k] \; + \; e^{\frac{-j2\pi k}{N}} x_{odd}[k]$$

Suppose, N = 8, to visualize the flow of data with time, we can make use of a butterfly diagram. We compute the Discrete Fourier Transform for the even and odd terms simultaneously. Then, we calculate x[k] using the formula from above.



Picture taken for illustration purposes from the YouTube Channel Rich Radke

We can express the gains in terms of Big O Notation as follows. The first term comes from the fact that we compute the Discrete Fourier Transform twice. We multiply the latter by the time taken to compute the Discrete Fourier Transform on half the original input. In the final step, it takes N steps to add up the Fourier Transform for a particular k. We account for this by adding N to the final product.

The complexity is calculated by breaking the original DFT in two smaller ones with N/2 elements and an extra N addition operation:

$$T(N) = 2 * (\frac{N}{2})^2 + N$$
$$= 2 * \frac{N^2}{4} + N$$
$$= \frac{N^2}{2} + N$$
$$= O(\frac{N^2}{2} + N) \sim O(N^2)$$

Notice how we were able to cut the time taken to compute the Fourier Transform by a factor of 2. We can further improve the algorithm by applying the divide and conquer approach, halving the computational cost each time. In other words, we can continue to split the problem size until we're left with groups of two and then directly compute the Discrete Fourier Transforms for each of those pairs.

$$\frac{N}{2} \rightarrow 2 * (\frac{N}{2})^2 + N = \frac{N^2}{2} + N$$

$$\frac{N}{4} \rightarrow 2 * (2 * (\frac{N}{4})^2 + \frac{N}{2}) + N = \frac{N^2}{4} + 2*N$$

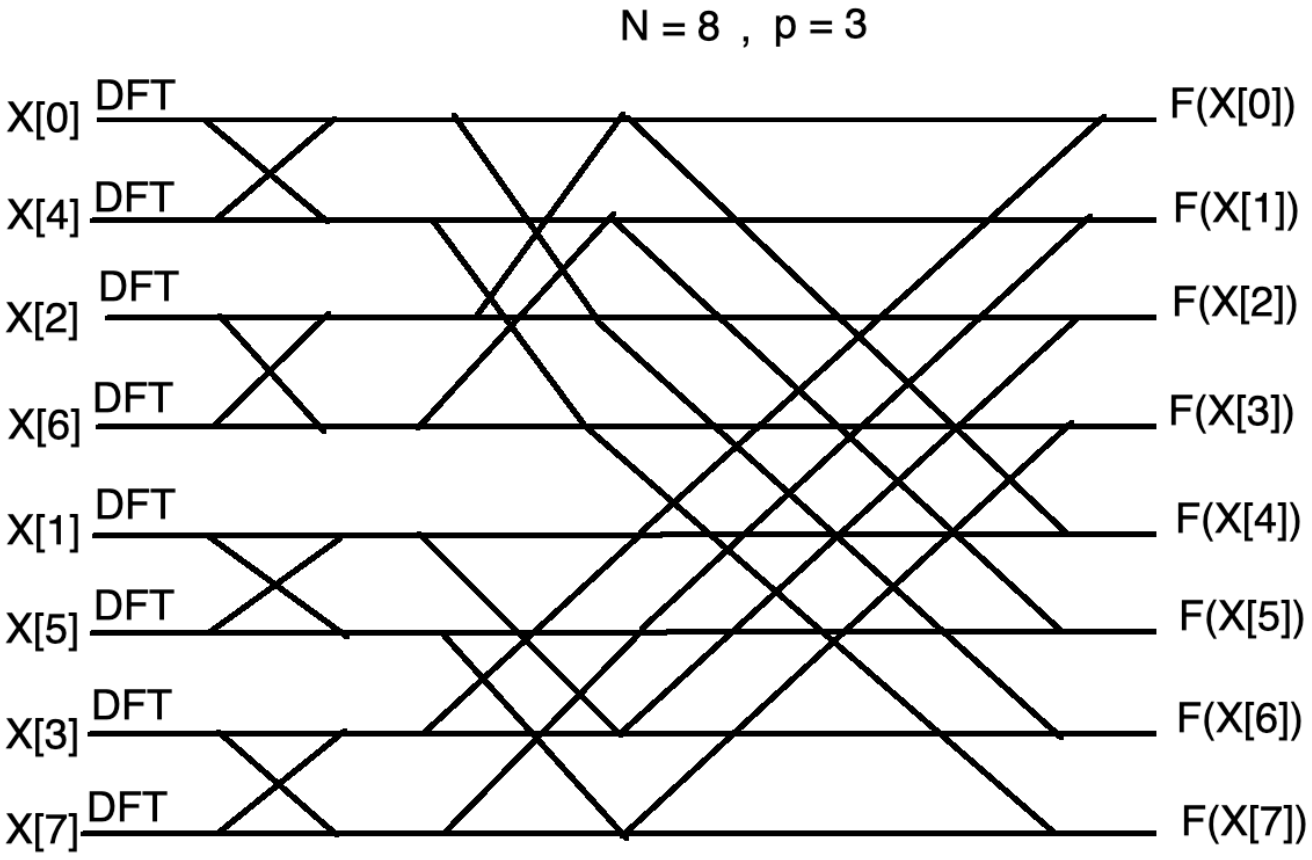
So long as N is a power of 2, the maximum number of times you split into two equal halves is given by $p = \log_2 N$.

Thus, the computational complexity would ultimately reduce to:

$$\frac{N}{2^p} \rightarrow \frac{N^2}{2^p} + p*N = \frac{N^2}{N} + (\log_2 N) * N = N + (\log_2 N) * N$$

$$\sim O(N\log_2 N)$$

Here's what it would look like if we were to use the Fast Fourier Transform algorithm with a problem size of N = 8. Notice how we have $p = \log_2(8) = 3$ stages.



Picture taken for illustration purposes from the YouTube Channel Rich Radke

Algorithm Implementation

Here is the working computational code using the divide and conquer approach to calculate the FFT of a function.

```
/*-------------------------------------------------------------------------*/


void fft(complex<double> x[],ll n)
{
    if(n==1)return;

    complex<double> x1[n/2],x2[n/2];
    for(ll i=0;i<n/2;i++)
    {
        x1[i]=x[2*i];
        x2[i]=x[2*i+1];
    }
    fft(x1,n/2);
    fft(x2,n/2);

    complex<double> w(cos((-2*PI)/n),sin((-2*PI)/n));
    complex<double> w2(1,0);
    for(ll i=0;i<n/2;i++)
    {
        x[i]=x1[i]+x2[i]*w2;
        x[i+n/2]=x1[i]-x2[i]*w2;
        w2*=w;
    }
}


/*-------------------------------------------------------------------------*/
```

Complexity Analysis

Although it is clear through the previous explanation that the time complexity for the FFT algorithm is $O(N\log_2 N)$, we will once confirm the complexity mathematically.

We can observe from Divide and Conquer strategy that,

$$T(N) = 2*T(N/2) + N$$

By Master's Theorem, where a=2, b=2, f(n)=n;

- ⇨ $\text{Log}_b a = \log_2 2 = 1$;
- ⇨ $n^{log_b a} = n^1 = n$;
- ⇨ $f(n) = O(n^{log_b a})$;

Therefore,

$T(n) = O(f(n) * \log(n))$;

⇨ **$\underline{T(n) = O(n\log n)}$**

Application of FFT in various fields

In layman's terms, the Fourier Transform is a mathematical operation that changes the domain (x-axis) of a signal from time to frequency. The latter is particularly useful for decomposing a signal consisting of multiple pure frequencies. The discrete Fourier transform (DFT) is the most widely used tool in digital signal processing (DSP) systems. It has indispensable role in many applications such as speech, audio and image processing, signal analysis, communication systems, and many others.

The application of the Fourier Transform isn’t limited to digital signal processing. The Fourier Transform can, in fact, speed up the training process of convolutional neural networks. A convolutional layer overlays a kernel on a section of an image and performs bit-wise multiplication with all of the values at that location. The kernel is then shifted to another section of the image and the process is repeated until it has traversed the entire image. The Fourier Transform can speed up convolutions by taking advantage of its convolution property. The above equation states that the convolution of two signals is equivalent to the multiplication of their Fourier transforms. Therefore, by transforming the input into frequency space, a convolution becomes a single element-wise multiplication. In other words, the input to a convolutional layer and kernel can be converted into frequencies using the Fourier Transform, multiplied once and then converted back using the inverse Fourier Transform.

Related research papers and References

Fourier Transforms and the Fast Fourier Transform Algorithm by Paul Heckbert
(Link to corresponding Reference)

50 Years of FFT Algorithms and Applications G. Ganesh Kumar, Subhendu K. Sahoo & Pramod Kumar Meher
(Link to corresponding Reference)

On the Partial Differential Equations of Mathematical Physics by R. Courant, K. Friedrachs, H. Lewy
(Link to corresponding Reference)

An Algorithm for the machine Calculation of Complex Fourier Series by James W. Cooley and John W. Tukey
(Link to corresponding Reference)

What is Fast Fourier Transform?
(Link to corresponding Reference)