Fast Fourier Transform and 2D Convolutions

Stephen Huan

October 16, 2020

1 Introduction

The Fast Fourier Transform (FFT) is a common technique for signal processing and has many engineering applications. It also has a fairly deep mathematical basis, but we will ignore both those angles in favor of accessibility. Instead, we will approach the FFT from the most intuitive angle, ***polynomial multiplication***. First, we represent polynomials by a list of coefficients, where the number at index 0 represents the coefficient of x^0, the number at index 1 represents the coefficient of x^1, and so on. For example, the polynomial $3 + 2x + 4x^2$ becomes [3, 2, 4].

The multiplication of two polynomials f and g is then simply each term of f multiplied with each term of g and then added up. We can also assume that f and g are the same length N, where the polynomial of lesser degree is padded with zeros. If we say the product is p, we can give an formula for an index in p in the following way:

$$p[n] = (f * g)[n] = \sum_{i=0}^{n} f[i]g[n-i]$$

$p[n]$ is the coefficient of x^n in the product, and it is formed by adding up all the possible ways to get to x^n, i.e. $f[0]x^0$ times $g[n]x^n$, $f[1]x^1$ times $g[n-1]x^{n-1}$, etc. Intuitively, this "flips" g, and then the resulting product is computed by "sliding" g over f and then computing the dot product between the two lists, or a weighted average.

Sticking with the example from before, we have the polynomial [3, 2, 4] and the polynomial $1 + 3x + 2x^2 = [1, 3, 2]$. To compute their product, we first flip the second list to get [2, 3, 1]. We then slide [2, 3, 1] over [3, 2, 4], imagining there are zeros such that the parts of [2, 3, 1] that don't overlap with [3, 2, 4] aren't counted. For the first value, 1 overlaps with 3 so we get 3. Then, [3, 1] overlaps with [3, 2] so we get $3 \cdot 3 + 1 \cdot 2 = 11$. [2, 3, 1] overlaps with [3, 2, 4] = 16, [2, 3] overlaps [2, 4] = 16, and finally [2] overlaps with [4] to give 8. Our final answer is then [3, 11, 16, 16, 8] = $3 + 11x + 16x^2 + 16x^3 + 8x^4 = (3 + 2x + 4x^2)(1 + 3x + 2x^2)$.

What if we computed $g * f$? It should be the same since polynomial multiplication should be commutative, but we can prove it.

Theorem 1.1. *$f * g = g * f$, i.e. polynomial multiplication is commutative.*

Proof. We have $(f * g)(n) = \sum_{i=0}^{n} f[i]g[n-i]$ by definition. Perform the variable substitution $k = n - i$, so $i = n - k$. Summing from $\sum_{i=0}^{n}$ will sum from $k = n$ to $k = 0$ in descending order, so $\sum_{i=0}^{n} = \sum_{k=0}^{n}$ (from the commutativity of addition).

$$\begin{aligned}(f * g)[n] &= \sum_{i=0}^{n} f[i]g[n-i] && \text{Definition} \\ &= \sum_{k=0}^{n} f[n-k]g[k] && \text{Substitution} \\ &= (g * f)\end{aligned}$$

□

This operation is known as a ***convolution***, which is equivalent to polynomial multiplication in the discrete case and is denoted $f * g$. Its relevance to image processing will be expounded on later (for now, this puts the "convolutional" in "Convolutional Neural Networks"). Today's lecture is about the ***Fast Fourier Transform***, an efficient algorithm to perform convolutions.

2 Algorithms

A naive approach to the convolution of two lists of length N, M will have runtime $O(NM)$ using the standard polynomial multiplication algorithm (each term of the first list multiplied with each term of the second list). However, the length of the convolution will be $N + M - 1$. The first list is a polynomial of degree $N - 1$, the second of degree $M - 1$, so the resulting polynomial has degree $(N - 1) + (M - 1) = N + M - 2$. A polynomial of degree D has $D + 1$ coefficients, so the length of the product is $N + M - 1$. Thus, the lower bound for a convolution is linear, so a better runtime than quadratic could exist.

2.1 FFT

2.1.1 Point-Value Representation

The key observation is that we can represent polynomials in a different form than a coefficient list. In particular, we can use a ***point-value*** representation, or a list of (x, y) pairs that give an input and the corresponding output of a polynomial. We call the process of going from a coefficient representation to a point-value representation ***evaluation***, since we evaluate the polynomial at multiple points to get the point-value representation. Likewise, we call the process of going from a point-value representation to a coefficient representation ***interpolation***, since we are finding a polynomial which "fits" the data. Suppose we have a polynomial of degree n. We then need a certain number of points for evaluation and interpolation to be well-defined. Evaluation is always well-defined, because we can always evaluate a polynomial of any degree or coefficient

representation. However, if we don't have enough points, interpolation is not necessarily possible. For example, consider the point-value representation $[(0,0),(1,1)]$ and a degree of 2. This could be the polynomial x^2 or $2x^2 - x$. So for a polynomial of degree n, we need at least $n+1$ distinct points (since each point gives another linear equation constraining the $n+1$ coefficients of the polynomial). We can in fact prove that if we have $n+1$ points, that uniquely determines a polynomial of degree n.

Theorem 2.1. *A point-value representation with n distinct points uniquely determines a polynomial of degree $n-1$.*

Proof. We have a polynomial of the form $p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$ and n points of the form (x_i, y_i) such that $p(x_i) = y_i$. Those constraints determine the following matrix equation:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

The leftmost matrix is known as the *Vandermonde matrix*, denoted $V(x_0, x_1, \ldots, x_{n-1})$ which has the determinant (left as an exercise for the reader)

$$\prod_{0 \le j < i \le n-1} (x_i - x_j)$$

A matrix is invertible if and only if its determinant is nonzero, so this matrix is invertible if each x_i is distinct. Thus, we can solve for the coefficients by multiplying by the inverse, so $\vec{a} = V^{-1}\vec{y}$, and this solution is unique since an invertible matrix is a *bijective* transformation between a vector space and itself. □

This proof directly gave an easy construction of the interpolating polynomial, by $V^{-1}\vec{y}$. Matrix inverses can be computed in $O(n^3)$ as an easy upper bound, but that can be improved with *Lagrange's interpolating formula* to yield a $O(n^2)$ time algorithm. I will not elaborate on Lagrange's formula in this lecture but a good Wikipedia page is available here.

If we have a list of N coefficients, then the polynomial is of degree $N-1$ and thus we need N distinct points. We first figure out how to evaluate polynomial at a single point, and will repeat the process for all the points. Suppose we have a polynomial of the form $p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$. If we evaluate at a particular x_0, we compute each $a_i{x_0}^i$ term which would take $O(N^2)$ time with repeated multiplication and $O(N \log N)$ time with fast exponentiation. But we can do better with *Horner's rule*. We notice that the degree in coefficient form is monotonically increasing, so we can successively factor out a multiplication by x.

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

We do N multiplications and N additions, so the algorithm runs in $O(N)$. Evaluating a polynomial at N points then takes $O(N \cdot N) = O(N^2)$ time.

So we can do both evaluation and interpolation in $O(n^2)$ and both are well-defined if we have enough points. Why did we figure this out? We can multiply two polynomials efficiently if we have the point-value representations of each! Suppose we have polynomials f, g in coefficient form. We also assume that the polynomials are evaluated at the *same* points, so we have $[(x_0, f(x_0)), (x_1, f(x_1)), \dots]$ and $[(x_0, g(x_0)), (x_1, g(x_1)), \dots]$. $f * g$ is then simply $[(x_0, f(x_0)g(x_0)), (x_1, f(x_1)g(x_1)), \dots]$, or the element-wise multiplication of the two lists. This can be easily computed in $O(n)$!

So our algorithm for polynomial multiplication is as follows:

1. Evaluate a coefficient representation into a point-value representation.
2. Multiply the two point-value representations in linear time.
3. Interpolate the resulting point-value representation back to coefficients.

The speed of this algorithm is contingent on our ability to quickly evaluate and interpolate a polynomial. Currently, with our $O(n^2)$ time evaluation and interpolation algorithms we match the $O(n^2)$ naive algorithm. However, under this framework we can improve the time if we pick our points cleverly rather than arbitrarily.

2.1.2 Complex Roots of Unity

Our special points are going to be ***complex roots of unity***, or roots of 1 that are allowed to have an imaginary component. For example, the second root of 1 can be 1 or -1 (taking "second root" to mean anything which squared is 1). The fourth root of 1 can be 1, -1, i, or $-i$. (since $i^4 = (i^2)^2 = (-1)^2 = 1$).

To easily compute these roots, we can rewrite 1 using Euler's formula $e^{ix} = \cos x + i \sin x$ (a proof of this appears in the appendix). $e^{2\pi i} = \cos 2\pi + i \sin 2\pi = 1$. So we can take a nth root by simply raising $(e^{2\pi i})^{\frac{1}{n}}$, so a root is $e^{\frac{2\pi i}{n}}$.

However, note that we can rewrite 1 in many different ways since sine and cosine are periodic. Since adding 2π doesn't change the value of sine and cosine, 1 is also equal to $e^{4\pi i}$, $e^{6\pi i}$, and so on. In general, $e^{2\pi k i}$ is equal to 1 for any integer k, so if we take the nth root, $e^{\frac{2\pi k i}{n}}$ is also a valid root of unity. However, not every k gives a distinct root of unity. $k = n + 1$ is equivalent to $k = 1$ since $\cos\left(\frac{2\pi(n+1)}{n}\right) = \cos\left(2\pi + \frac{2\pi}{n}\right) = \cos\left(\frac{2\pi}{n}\right)$. This generalizes such that an power k equivalent to $j \bmod n$ will have the same root.

We can easily keep track of the n distinct nth roots of unity by writing them as powers of the ***principle root of unity***, or the root of unity when $k = 1$. We will denote this principle root as ω_n, where $\omega_n = e^{\frac{2\pi i}{n}}$, the first of unity we calculated above. Since we picked $k = 1$, we can represent every nth root of unity as a power of this root of unity since $e^{\frac{2\pi k i}{n}} = (e^{\frac{2\pi i}{n}})^k = \omega^k$. Also, every power of the principle root of unity is itself a nth root of unity, because $(\omega_n^k)^n = (\omega_n^n)^k = 1^k = 1$.

We now come to an observation that will be instrumental in developing the FFT - that the square of a nth principle root of unity is a $\frac{n}{2}$th principle root of unity. This follows nearly from definition: $\omega_n^2 = (e^{\frac{2\pi i}{n}})^2 = e^{\frac{4\pi i}{n}} = e^{\frac{2\pi i}{n/2}} = \omega_{\frac{n}{2}}$.

We now show that evaluating a polynomial at n distinct nth roots of unity can be written as a recurrence relation. Our observation is a clever rewrite of a polynomial into two parts. Suppose we have the polynomial $p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^{n-1}$. We divide the coefficient list of p into two parts, one with even powers and the other with odd powers, the left and right halves respectively. We assume that n is a power of 2 so that p can always be divided in such a manner (if n isn't, we can always pad with 0's).

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^{n-1} \tag{1}$$

$$L(x) = a_0 + a_2x + a_4x^2 + \dots \tag{2}$$

$$R(x) = a_1 + a_3x + a_5x^2 + \dots \tag{3}$$

It follows that p can be written in terms of L and R:

$$p(x) = L(x^2) + xR(x^2) \tag{4}$$

Recall that we are trying to evaluate p at n roots of unity. Suppose we have a function that takes as input a list of coefficients and returns the evaluation at n roots of unity. We can define this function in terms of itself, because we have a recurrence relation - divide the list in two, giving us L evaluated at $\frac{n}{2}$th roots of unity and the same for R (from the fact that a nth root of unity squared is a $\frac{n}{2}$th root of unity). Finally, we can reconstruct p from L and R according to (4). This works directly for ω_n^0 to $\omega_n^{\frac{n}{2}-1}$, however for a power greater than $\frac{n}{2} - 1$ we need to put it in terms of a power less than $\frac{n}{2}$ (since L and R are only $\frac{n}{2}$ long). Luckily,

$$\begin{aligned}
\omega_n^{k+\frac{n}{2}} &= \cos\left(2\pi\frac{k+\frac{n}{2}}{n}\right) + i\sin\left(2\pi\frac{k+\frac{n}{2}}{n}\right) \\
&= \cos\left(\frac{2\pi k}{n} + \pi\right) + i\sin\left(\frac{2\pi k}{n} + \pi\right) \\
&= -\omega_n^k
\end{aligned}$$

So, for some power k of the base root of unity we can compute

$$p(\omega_n^k) = L(\omega_n^{2k}) + \omega_n^k R(\omega_n^{2k})$$

and, using the negative property derived above,

$$p(\omega_n^{k+\frac{n}{2}}) = L(\omega_n^{2k}) - \omega_n^k R(\omega_n^{2k})$$

We compute L and R recursively, and we're done! The base case is also trivial - at any point, say, for $n = 1$, we can stop dividing the list in half and then just evaluate the polynomial at a single point. The 1st principle root of unity is just 1, and evaluating a polynomial at $x = 1$ is the sum of its coefficients, which for a polynomial with one

coefficient is just the single coefficient. Thus, we can just return the input to the function. The running time of this algorithm will be $O(n \log n)$, since at every level of the recursion we divide the list in half, making the depth of the recursive tree $\log n$, and at every level of the tree we do n total work (at a particular level k, we have 2^k nodes and each node has a list of length $\frac{n}{2^k}$, so the total cost of merging the lists together on that level is $2^k \frac{n}{2^k} = n$. Thus, $O(\log n \cdot n) = O(n \log n)$ (this algorithm has the same recursive pattern as merge sort).

So we can evaluate a polynomial at n roots of unity in $O(n \log n)$ with the above algorithm, called the FFT. Thus, if we want to multiply two polynomials f, g, we can compute $\text{FFT}(f) \circ \text{FFT}(g)$, where $\circ$ is the element-wise multiplication of the outputs in the point-value representations. How do we interpolate coefficients from this point-value representation to complete our convolution? We need the inverse FFT, which luckily can be written in terms of the FFT. Recall that the FFT essentially computes the multiplication of the Vandermonde matrix with the coefficients to get to the outputs, e.g. $V\vec{a} = \vec{y}$. To go from the outputs to the coefficients, we can simply multiply by V^{-1}, i.e. $\vec{a} = V^{-1}\vec{y}$. Computing V^{-1} is tedious and I don't have much insight (read *Introduction to Algorithms* for a proper proof), but it essentially involves just the definition of matrix inverse and more properties of roots of unity. It turns out that V^{-1} is essentially V but evaluated at x^{-1} instead of x. Also, divide by n. So we can just use the FFT but take the inverse of the root of unity, and divide each element by n at the end. Finally, we arrive at the FFT formulation of convolutions.

Theorem 2.2. *$f * g = FFT^{-1}(FFT(f) \circ FFT(g))$, i.e. convolutions can be done with FFTs in time $O(n \log n)$.*

Proof. Follows from the last 5 pages. A concrete implementation can be found here. □

2.1.3 Iterative variant

The recursive algorithm can be made iterative surprisingly elegantly from a pattern in binary form of the indexes when recursively subdividing. I omit the details here, although it makes the algorithm $O(n)$ in memory instead of $O(n \log n)$ and will likely run faster than the recursive algorithm. An implementation is above.

2.2 Number Theoretic Transform (NTT)

Another improvement on the FFT comes from the observation that complex roots of unity were an arbitrary pick, any ***field*** with sufficient properties will do. In particular, we can pick a large prime number p and find an equivalent to a root of unity under the field modulo p. The details are incredibly tedious and number theory heavy, but they yield the ***number theoretic transform***, a variant of the FFT which operates on integers (useful for polynomials of integer coefficients or certain types of data, e.g. music or images, which have integer pixel values). One downside is that negative numbers do not exist under modulo, which can be accounted for by assuming large numbers are in fact negative, changing the range from $[0, p)$ to $[-\frac{p}{2}, \frac{p}{2})$.

3 Applications of Convolutions

3.1 Audio Processing

Before we apply 2D convolutions to images, we elucidate the 1D convolution and its usefulness through an illustrative example.

> *The anime music quiz problem.* We have a song that is 1 minute and 30 seconds long, and a 10 second clip from that song. We wish to compute:
>
> 1. Out of a list of songs, which song the clip came from.
> 2. From a known song, the timestamp where the clip occurred.

```
(amq) stephenhuan@MacBook-Pro ~/P/p/p/amq (master o)> python db.py clip
Picking a clip from NeonGenesisEvangelion at -4.04dB
```

(a) Generating a clip from an anime intro.

```
NekomonogatariKuro-OP1        : 4590137    loss, occurs at 3.4   seconds in
NeonGenesisEvangelion         : 3931757    loss, occurs at 35.2  seconds in
Nichijou_op2                  : 8316782    loss, occurs at 42.4  seconds in
Nisemonogatari_op1            : 4663729    loss, occurs at 77.3  seconds in
```

(b) Comparison between songs; finds it occurs exactly 35.2 seconds in.

```
----------
Final answer: Neon Genesis Evangelion
123 songs in 8.61 seconds = 14.29 songs per second
```

(c) Song with the lowest loss.

Figure 1: An example run of the system.

First, some basics about the representation of audio data. We will use the mp3 file format at a sample rate of 48kHz. Audio is fundamentally just a list of numbers, where each number represents the amplitude of the sound wave at that time. A 48kHz sample rate means there are 48,000 of these measurements per second. Each number is a 16-bit number in the range $[0, 1)$, which we will transform to an integer in the range $[0, 2^{16})$ for the NTT. So we have two lists of integers, and now wish to find where the smaller list "fits" into the larger list the best. One way to do this is to compute the ℓ^2 ***norm***, or the vector difference between the two lists. So we slide the smaller list over the larger list, computing the sum of squares error as we go. Note that this is very similar to the convolution, except we to calculate the sum of squares instead of the element-wise product. We also need to flip one of the lists because the convolution flips a list.

How do we reduce sum of squares to an element-wise product? We notice that $(a_i - b_j)^2 = a_i^2 - 2a_i b_j + b_j^2$ for elements of the lists a, b. When we sum over the length of a (assuming a is the smaller list), we get $\|a\|^2 - 2a \cdot b' + \|b'\|^2$, where b' is the slice that a overlaps. $\|a\|$ is a constant, so it can be ignored. Thus, we only need to compute $a \cdot b'$ and $\|b'\|$. $a \cdot b'$ directly follows from a convolution and can be read from $a * b^r$, where b^r is the reverse of b. Lastly, if we make sure to scan from left to right, then we can compute $\|b'\|$ by storing an intermediate value, and updating it when we slide a an additional index by subtracting out the front of b', where a left from, and adding the new value that a covers.

Algorithm 1 minimum ℓ^2 between two lists

```
def min_offset(a: list, b: list) -> tuple:
    N, M = len(a), len(b)
    p = fft(a[::-1], b)[N - 1:]
    x2, xy, y2 = sum(x*x for x in a), p[0], \
                 sum(b[i]*b[i] for i in range(N))
    best, l2 = 0, -2*xy + y2
    for i in range(1, M - N + 1):
        y2 += b[N - 1 + i]*b[N - 1 + i] - b[i - 1]*b[i - 1]
        xy = p[i]
        d = -2*xy + y2
        if d < l2:
            best, l2 = i, d
    return best, x2 + l2
```

We need to be careful about a few things. If we don't pick p for the NTT large enough, then it won't work. If m is the largest number in a list and n is the length of the list, then we need p to be bigger than $m^2 n$ (the largest a single element can become). n is $90 \cdot 48,000 \approx 4 \cdot 10^6$ and m is 2^{16}. $m^2 n = 2^{32} \cdot 4 \cdot 10^6 \approx 2^{54}$. This seems fine since 2^{54} will fit in a long, but this won't work since we need to compute x^2 as part of the FFT, and $(2^{54})^2$ will definitely overflow. We could get around this overflow by doing modulo multiplication instead of standard multiplication, but that would introduce a log factor, making the algorithm 64x slower, an unacceptable slowdown. One trick is to reduce the bitrate of the mp3 at the sacrifice of audio quality, and go from 16-bit audio to 8-bit audio. A naive way to do it would be to multiply the real number by 2^8 and round, but a better way is the μ-law algorithm, a trick that preserves frequencies closer to the human voice. A comparison between naive scaling and the μ law is presented here. With 8-bit audio, $m^2 n = 2^{16} \cdot 4 \cdot 10^6 \approx 2^{38}$. This goes over the limit of 2^{32} for x^2 to fit in a long, but it works in practice since mp3 audio rarely hits maximum volume and our clip is 10 seconds long, and we computed for 90 seconds.

As mentioned in the NTT, we also need to avoid negative numbers. If we have a value greater than $\frac{p}{2}$, we assume it is negative and subtract p from it to get its proper value, otherwise we keep the value the same.

accounting for negative values

```
def ntt_sign(l: list, p: int) -> list:
    return [x if x < (p >> 1) else x - p for x in l]
```

An implementation is given here and a video walkthrough here.

4 2D Convolutions

2D convolutions, a convolution generalized to matrices, are useful in computer vision for a variety of reasons, including edge detection and convolutional neural networks. Their exact usage will not be discussed here, and instead we will discuss an efficient way to calculate a 2D convolution with the FFT we have already developed. We have an "data" matrix, representing an image, and we have a ***kernel*** matrix, which is the matrix we imagine sliding over the image. This is also known as a ***filter***.

For 2D convolutions, the result is slightly ambiguous depending on how one defines it. We will use scipy's definition, where to calculate the value of the convolution at a particular point, we imagine the *bottom right* corner of the kernel placed over that point.

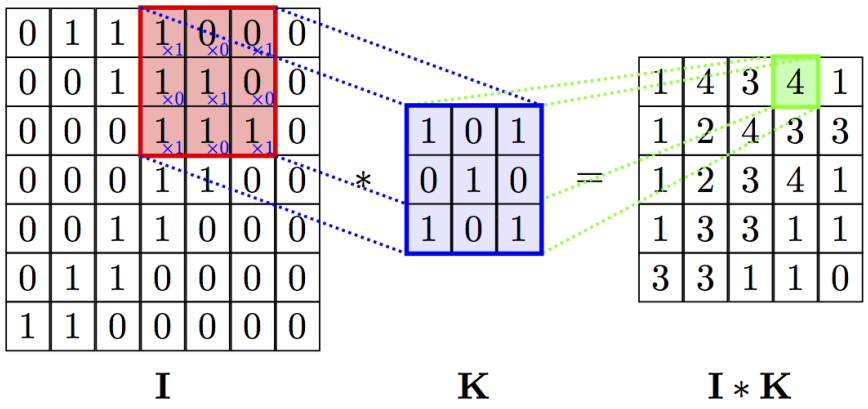

Figure 2: A convolution taken from here.

We define the 2D convolution between an image x of size MxN and a kernel h of size HxW as follows (similar to the 1D case, we assume both matrices are padded with 0's):

$$(x * h)[i, j] = \sum_{k=0}^{i} \sum_{l=0}^{j} x[k][l]h[i-k][j-l]$$

This operation is also symmetric, so what we call the image and the kernel is essentially arbitrary (by convention, the kernel is the smaller matrix). The resulting matrix is going to be of size $(M + H - 1)$x$(N + W - 1)$ from the same logic as the 1D case. Thus, the time it takes to compute the convolution is $O(MNHW)$. We can, however, take advantage of a trick if the kernel has a certain property.

4.1 Separable Kernels

A matrix M is ***separable*** if it can be written as $\vec{u}\vec{v}^T$ for some vectors $\vec{u}, \vec{v}$. For example, the famous Sobel matrix for edge detection is separable:

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$$

If a matrix is separable, we can convolute with $\vec{u}$ and then with $\vec{v}$.

Theorem 4.1. *If $h = \vec{u}\vec{v}^T$, then $(x * h) = ((x * \vec{u}) * \vec{v})$, i.e. we can separate a convolution into two parts.*

Proof.

$$(x * u)[i, j] = \sum_{k=0}^{i} \sum_{l=0}^{j} x[k][l]u[i-k][j-l] \qquad \text{Definition}$$

Since u is a column vector, it only has values when $l = j$, removing the inner sum.

$$= \sum_{k=0}^{i} x[k][j]u[i-k][0]$$

Convoluting with v,

$$((x * u) * v)[i, j] = \sum_{k=0}^{i} \sum_{l=0}^{j} (\sum_{m=0}^{k} x[m][l]u[k-m][0])v[i-k][j-l]$$

Since v is a row vector, it only has values when $k = i$, removing the outermost sum.

$$= \sum_{l=0}^{j} (\sum_{m=0}^{i} x[m][l]u[i-m][0])v[0][j-l]$$

Swapping the order of summations and renaming m to k,

$$= \sum_{k=0}^{i} \sum_{l=0}^{j} x[k][l]u[i-k][0]v[0][j-l]$$

From the fact that $h[x][y] = u[x][0]v[0][y]$,

$$= \sum_{k=0}^{i} \sum_{l=0}^{j} x[k][l]h[i-k][j-l]$$
$$= (x * h)$$

□

How does this help us? Well, recall the running time of $O(MNHW)$. If we do two convolutions of a kernel of Hx1 and another of 1xW, the running time will be $O(MNH + MNW) = O(MN(H+W))$, a significant improvement as HW grows quadratically while $H + W$ grows linearly. We can also use repeated 1D convolution to compute the 2D convolution for the specific case of a vector, yielding a $O(MN \log MN)$ time algorithm.

However, not every matrix is separable. The conditions are quite strict, a matrix is separable if and only if every pair of rows is a multiple of each other, i.e. the matrix is made up of multiples of a particular row vector. As a consequence, the matrix is also made up of multiples of a particular column vector. These matrices are relatively rare, so there is utility in deriving a more general algorithm.

4.2 FFT Algorithm

We can reduce 2D convolutions to 1D convolutions if we're clever. The observation is that if we ***flatten*** both matrices into a 1D list by reading from top to bottom, left to right, we can just convolute in 1D and reconstruct the matrix afterwards. We need to make sure both matrices are sufficiently padded with zeros, such that the zeros force values in the kernel to their proper rows in the image. It turns out that we can just pad both matrices to the final column size of the convolution, $N + W - 1$, flatten both, convolute with the FFT, and then reshape the resulting list to a matrix of proper size. In most computer vision applications, the kernel is a square matrix of size KxK, where

Algorithm 2 2D Convolution Algorithm

```
def flatten(m: list, pad=0) -> list:
    """ Flattens a matrix into a list. """
    return [x for row in m for x in row + [0]*pad]

def reshape(l: list, m: int, n: int) -> list:
    """ Shapes a list into a MxN matrix."""
    return [[l[r*n + c] for c in range(n)] for r in range(m)]

def conv(h: list, x: list):
    """ Computes the 2D convolution. """
    M, N, H, W = len(x), len(x[0]), len(h), len(h[0])
    # need to pad the columns to the final size
    h, x = flatten(h, N - 1), flatten(x, W - 1)
    return reshape(fft(h, x), M + H - 1, N + W - 1)
```

K is an odd number. The *middle value* of the kernel is then placed over each pixel of the image, yielding a transformed image of the same dimensionality as the original. We can simulate this by simply cutting off the first and last $\frac{K-1}{2}$ rows and the same for the columns. This transforms the resulting size from $N + K - 1$ to $N + K - 1 - 2\frac{K-1}{2} = N$.

pruning

```
def prune(h: list, x: list) -> list:
    """ Prunes a convolution for the specific KxK filter case. """
    m, k = conv(h, x), min(len(h), len(x))
    pad = (k - 1)>>1
    return [row[pad:-pad] for row in m[pad:-pad]]
```

The running time of the algorithm is going to be $O(MN \log MN) = O(MN(\log N + \log M) = O(MN \log N)$ since we convolute a list of length $M(N+W-1)$, and we assume $N \geq M > W$. This is not necessarily faster than the brute-force algorithm; it depends on the kernel size. For simplicity, suppose we have a NxN image and a KxK kernel where $N > K$. Brute force yields $O(N^2K^2)$ while the FFT algorithm yields $O(N^2 \log N)$. Thus, if $\log N < K^2$ then the FFT algorithm is going to be faster. For $K > 5$ that is a fair assumption since $K^2 = 25$, 2^{25} is several million. Obviously the FFT algorithm has a much larger constant factor, but for a sufficiently large kernel the time savings become greater and greater.

5 Conclusion

The convolution, an operator very useful for signal, audio, and image processing, can be efficiently computed with the Fast Fourier Transform, or FFT. If the data is integer, then floating-point arithmetic can be avoided with the Number Theoretic Transform (NTT), a variant of the FFT which uses modulo instead of complex numbers, and calculates entirely in integers.

This lecture skips over the continuous case (what I've been calling the Fast Fourier Transform is more mathematically called the ***Discrete Fourier Transform***, or DFT) but the idea is essentially the same, any summation turns into an integral. It also skips over the mathematical interpretation of the FFT, involving decomposing a function into a series of sine and cosine waves. This is useful for signal processing and audio analysis, but requires a stronger mathematical background and to be honest, I haven't studied it at all myself. Fourier analysis goes deeper than we need here.

Introduction to Algorithms is definitely the most helpful source on the FFT (from a computer science perspective), and more thorough treatments of the FFT from an engineering or mathematical standpoint are not hard to find.

6 Sample Problems

1. SPOJ POLYMUL: Direct application of the FFT.

2. SPOJ MUL: Given 1000 pairs of numbers, compute the product of each pair; each number can have up to 10,000 digits.

   Solution: Think of numbers as polynomials, where the digits are coefficients and x is 10. Then, you can multiply two numbers by multiplying the polynomials. However, there is no guarantee that the coefficients of the resulting polynomial are less than 10, so it is not a valid number. As a last post-processing step, start from the smallest place value and move your way to the largest, moving the digit overflow from one place value to the next. Since you iterate over the number of digits in the number, it takes $O(\log n)$ which is dominated by the FFT.

   An extension of this idea is the Schönhage–Strassen algorithm, which disregards the requirement that the intermediate numbers fit in a long, at the cost of being $O(n \log n \log \log n)$. A more recent algorithm, by Harvey and van der Hoeven, achieves $O(n \log n)$.

3. SPOJ MAXMATCH: Given a string S of length N made up of the characters "a", "b", and "c", compute the maximum self-matching, where a self-matching is defined as the number of characters which match between S and S shifted some nonzero number of characters.

   Solution: For an offset i, the size of the overlap will be $N - i$. So we just need to find the number of differences, and subtract that from $N - i$ to obtain the number of matches. The easiest thing to do is to keep track of each character separately, so to compute the differences for each character. Suppose our character is "a". We encode "a" as a 0, and the other characters as a 1. We then find the ℓ^2 norm between this new list and this list with N 1's added to it (so that when we overlap, the non "a" characters aren't counted). This has the complication of counting "a"'s which are off the edge of the string, which we can account for by simply keeping track of the number of "a"'s we have seen.

   Given $a[i]$ as the number of mismatches with the character "a" at a shift of i, and $b[i], c[i]$, the number of matches is $N - i - \frac{a[i]+b[i]+c[i]}{2}$. We divide by 2 because we count each mismatch twice (once for each character in the pair).

   A much conceptually simpler algorithm is to encode "a", "b", and "c" cleverly and then compute the matches in one shot. If we encode "a" as (1, 0, 0), "b" as (0, 1, 0), and "c" as (0, 0, 1), the FFT of the resulting list with its reverse will give us the number of matches at each index because the character representations dot each other will be 1 if they are equal, and 0 if they are unequal. Thus, the FFT will give us exactly the number of matches, but we need to only look at every 3rd index since the other 2 are byproducts of our transformation.

   In practice, running one big FFT is faster than running 3 smaller FFTs.

4. Codechef FARASA: Given an array, find the number of distinct sums of a contiguous subarray.

   Solution: editorial.

   Fair warning, time bounds are ridiculous.

5. Codeforces Round #296: Given two strings T, S and an error bound k, find all the positions where T occurs in S, where T "occurring" at some index i means that the jth character of T has a corresponding character within k of its position.

   Solution: Honestly no clue but it has the "FFT" tag.

6. String matching with wildcards: Given two binary strings T, S, T has length N and has wildcards which match any character in S, find all occurrences of T in S.

   Solution: Encode 1 as 1 and 0 as -1. The dot product between T and the slice that T overlaps with S will be be N if they match exactly and less than N if they don't match exactly. To account for wildcards, encode a wildcard as 0 and count the number of wildcards, C. Then, if they match exactly it will be $N - C$, and less then that if they don't.

   This can be generalized to non-binary strings if you apply the above algorithm to each character, setting that character as 1 and not that character as -1. Sum over all possible characters, and that will tell you whether there is a mismatch somewhere (similar to SPOJ MAXMATCH).

   This idea can also be applied to string matching without wildcards. Encode each character as its ASCII value in a polynomial, and compute the ℓ^2-norm between T and S. The ℓ^2 norm will be 0 if they match, and positive if they don't.

7. 3SUM: Given a list of integers between $-N$ and N, find 3 numbers that add up to 0 (duplicates are allowed).

   Solution: The basic idea will be to encode the list into a length $2N$ polynomial p where the degree is an integer value and the coefficient is whether that value appears in the array. Compute p^3 and read off the coefficient of x^0. However, this doesn't work if the degrees are negative. If the most negative power of x in p is x^{-N}, We can simply multiply p by x^N to make every power positive, making a new polynomial p'. Then, after computing $(p')^3$, instead of looking at the coefficient of x^0, we can look at the coefficient of x^{-3N} (accounting for the fact that $p' = x^N p$, $(p')^3 = x^{3N} p^3$, $p^3 = \frac{(p')^3}{x^{3N}}$

   Alternative solution, if duplicates aren't allowed: here (look for "color coding").

8. Anime Music Quiz: Guess which anime an intro/outro comes from.

   Solution: The Shazam algorithm.

7 Appendix

Theorem 7.1. $e^{ix} = \cos x + i \sin x$, *i.e. Euler's formula*

Proof. We have the initial value problem (IVP)

$$\frac{dy}{dx} = f(x, y), y(x_0) = y_0$$

Picard's existence and uniqueness theorem says that if f and $\frac{\partial f}{\partial y}$ are continuous functions on some rectangle R that contains (x_0, y_0), then the IVP has an unique solution on some interval I whose bounds are the regions where the hypotheses hold.

In our particular case, we have $f(x, y) = iy$ and $y(0) = 1$, so $\frac{\partial f}{\partial y} = i$. By Picard's theorem, the IVP has a unique solution on the interval I where y is continuous and $\frac{\partial f}{\partial y}$ is continuous. i is continuous everywhere, so the IVP will have an unique solution wherever y is continuous.

Because this differential equation is separable, we can directly solve for y.

$$\begin{aligned}
\frac{dy}{dx} &= iy \\
\int \frac{1}{iy} dy &= \int dx \\
\frac{1}{i} \ln |iy| &= x + C \\
\ln |iy| &= ix + C \\
iy &= Ce^{ix} \\
y &= Ce^{ix}
\end{aligned}$$

Taking into account the initial condition, $y(0) = 1 = Ce^0 = C$. So $y = e^{ix}$, which is continuous on $\mathbb{R}$. Picard's theorem therefore guarantees the uniqueness of this solution. However, note that $\cos x + i \sin x$ is also a solution to the IVP. First, it fulfills the initial condition since $y(0) = \cos 0 + i \sin 0 = 1$. Second, it fulfills the differential equation:

$$\begin{aligned}
\frac{dy}{dx} &= -\sin x + i \cos x \\
&= i^2 \sin x + i \cos x && \text{Definition of } i \\
&= i(\cos x + i \sin x) \\
&= iy
\end{aligned}$$

Since e^{ix} is the unique solution to the IVP on $\mathbb{R}$, $e^{ix} = \cos x + i \sin x$. □

8 Past Lectures

1. "Edge Detection", (Alexey Didenkov, 2018)
2. "Fast Multiplication: Karatsuba and FFT" (Haoyuan Sun, 2016)
3. "Multiplying Polynomials", (Haoyuan Sun, 2015)
4. "Fast Fourier Transform", (Sreenath Are, 2013)

9 References

1. *Introduction to Algorithms*, chapter 30 (very helpful)
2. The number theoretic transform
3. μ-law algorithm
4. Picard's Existence and Uniqueness Theorem
5. Separable convolutions