Implementation: Padding, Fast Fourier Transform

CS 450: Introduction to Digital Signal and Image Processing

Bryan Morse
BYU Computer Science

Implementing Convolution using the Fourier Transform

We saw earlier:

$$f(t) * h(t) = \mathcal{F}^{-1}(\mathcal{F}(f(t))\mathcal{F}(h(t)))$$

1. Compute the Fourier Transform of the input signal $f(t)$
2. Compute the Fourier Transform of the convolution kernel $h(t)$
   - May be done by building the kernel and applying the DFT/FFT
   - May be calculated analytically directly in the frequency domain
3. Multiply the two Fourier Transforms together (complex multiply)
4. Apply the inverse Fourier Transform (if the signal and kernel are real-valued, just use the real part of the result)

Padding the Kernel

- To multiply $F(u)$ and $H(u)$, they must be the same size
- This means the discrete $f(t)$ and $h(t)$ must be the same size
- If necessary, pad the kernel $h(t)$ with zeroes to be the same size as the signal $f(t)$ —adding zeroes to a convolution kernel doesn't change anything

Convolution–What happens at the image boundary?

What do you do when you need to use a neighbor outside the image?

Options:

- Zero (can cause edge, darkening near the image boundary)
- Average value (can still cause edge, generally avoids darkening)
- Repeated values (same as the nearest one inside)

What does the Discrete Fourier Transform implicitly do?
Treats the signal as periodic—outside the boundary it just starts over

Specifying Your Own Outside Values

Since convolution by multiplying Fourier Transforms doesn't directly access neighbors, you can't test for going outside the image and then substitute a value.

So, what do you do if you want to specify your own outside values?

Solution: Pad the image yourself with the desired value.

Padding

Rules/steps for padding:

- Decide what values you want to use to pad (zeroes, average value, repeated values, etc.)
- Pad the image by at least the size of your kernel (the non-zero part)
- Pad further to nearest power of 2 if using the FFT (may be required depending on the implementation)
- Fourier Transform, multiply, inverse Fourier Transform
- Crop back to the original size

Discrete Fourier Transform - Revisited

The Discrete Fourier Transform is

$$F(u) = \frac{1}{M} \sum_{x=0}^{M-1} f(x)\ e^{-i2\pi ux/M}$$

- This takes $O(M^2)$ to compute
- Can we do it faster?

The Fast Fourier Transform

If we let

$$W_M = e^{-i2\pi/M}$$

the Discrete Fourier Transform can be written as

$$F(u) = \frac{1}{M} \sum_{x=0}^{M-1} f(x)\ {W_M}^{ux}$$

If M is a multiple of 2, $M = 2K$ for some positive number K.

Substituting $2K$ for M gives

$$F(u) = \frac{1}{2K} \sum_{x=0}^{2K-1} f(x)\ {W_{2K}}^{ux}$$

The Fast Fourier Transform

Separating out the K even terms and the K odd terms,

$$F(u) = \frac{1}{2}\left\{\frac{1}{K}\sum_{x=0}^{K-1} f(2x)\,{W_{2K}}^{u(2x)} + \frac{1}{K}\sum_{x=0}^{K-1} f(2x+1)\,{W_{2K}}^{u(2x+1)}\right\}$$

Notice that

$$W_{2K}^{2u} = W_K^u$$

$$W_{2K}^{2(u+1)} = W_{2K}^{2u}\,W_{2K}^{u} = W_K^u W_{2K}^u$$

So,

$$F(u) = \frac{1}{2}\left\{\frac{1}{K}\sum_{x=0}^{K-1} f(2x)\,W_K^{ux} + \frac{1}{K}\sum_{x=0}^{K-1} f(2x+1)\,W_K^{ux} W_{2K}^u\right\}$$

The Fast Fourier Transform

$$F(u) = \frac{1}{2}\left\{\frac{1}{K}\sum_{x=0}^{K-1} f(2x)\,W_K^{ux} + \frac{1}{K}\sum_{x=0}^{K-1} f(2x+1)\,W_K^{ux} W_{2K}^u\right\}$$

$$F(u) = \frac{1}{2}\left\{F_{\text{even}}(u) + F_{\text{odd}}(u)\,W_{2K}^u\right\}$$

Use this to get the first K terms ($u = 0..K-1$), then re-use these parts to get the last K terms ($u = K..2K-1$):

$$F(u+K) = \frac{1}{2}\left\{F_{\text{even}}(u) - F_{\text{odd}}(u)\,W_{2K}^u\right\}$$

Only have to do $u = 0..M/2-1$, $x = 0..M/2-1$ and a little extra math.

The Fast Fourier Transform

Computational Complexity:

Discrete Fourier Transform	$O(N^2)$
Fast Fourier Transform	$O(N \log N)$

Remember:
The Fast Fourier Transform is just a faster *algorithm* for computing the Discrete Fourier Transform—it is *not* any different in result.

Implementation: Padding, Fast Fourier Transform
└The Fast Fourier Transform

What About 2-D?

Remember, we can use separability of the Fourier Transform to break a 2-D transform into $2N$ 1-D transforms:

DFT	$O(N^4)$
DFT using separability	$O(N^3)$
FFT using separability	$O(N^2 \log N)$

Convolution vs. Filtering

Remember the convolution theorem: we can implement convolution using frequency-domain filtering.

For an $N \times N$ image and a $K \times K$ kernel:

Convolution	$O(N^2K^2)$
FFT using separability	$O(N^2 \log N)$

Even if we have to pad the $K \times K$ kernel to $N \times N$, we can do three $O(N^2 \log N)$ operations faster than one $O(N^2K^2)$ one for sufficiently large K.

Break-even point: around $K = 7$