Notebook3

February 19, 2018

Contents

1 Information and Pitfalls in Feature detection

1. Misconceptions
   - I wrongly misconstrued what I had to do with this assignment, I had two wrong starts.
     (a) I thought I had to use Fast/Discrete Fourier Transformers.
     (b) I thought I had to implement the SIFT algorithm.
   - Due to these Misconceptions I have some insights with these two approaches
   - I eventually ended up on the discrete correlation algorithm which relied on convolutions.
2. Fourier Transformers what are they
   - A Fourier Transformer can be thought of as a formula that transformers a wave into the components that make up said wave.
   - For a concrete example, let's say that we wish to find out the ingredients of a drink
     – The Fourier transformer would allow us to split the drink into the base components (signals).
     – Note with this example, we can easily see that we can combine all the ingredients of the drink and get back our original drink. This is important because the Fourier Transformer has an inverse function, meaning that we can get our original signal back
   - Fourier Transformations are used for this as it can reduce a complex convolution like operation (cross correlation) into a few transformations and a single matrix multiplication

(a) Discrete Fourier Transformer

- Fourier Transformers are defined on continuous space, however we wish to do these operations on an image, so we need to do this over a **discrete** domain
  - The formula for the Discrete Fourier transformer is listed below, note that this formula is run on complex numbers
  - $X_K = \sum_{n=0}^{N-1} x_n \times e^{-i2\pi kn/N}$ where $\{x_n\} = x_0, \ldots, x_{N-1}$
  - This transformation will be denoted with F
  - Notice that each computation of X_k takes N computations, leaving the entire operation at $O(N^2)$

(b) Fast Fourier Transformer

- This is just a way computing the Discrete Fourier Transformer in O(N log(N)) instead of $O(N^2)$
- However there are some caveats, The variation used in Haskell's Repa library uses Cooley-Tukey algorithm which requires that N is a power of two as it's a divide and conquer algorithm which splits the problem into 2's.
- This means that in order to use a Fast Fourier Transformer, one must convert their image into a power of two

3. SIFT

- SIFT stand for Scale invariant feature transform, and there are many similar algorithms which preform better (such as speeded up robust features (SURF)).
- SIFT is more robust than what I ended up going with as it resistant to illumination changes and to a lesser degree affine changes, which would allow it to match many of the images seen in slide 2 and 3 of Lecture 4.
  - Affine transformation is a mapping $f : X \to Y$ where X and Y are Affine spaces s.t $f(x) = Mx + b$.
  - Basically these transformations preserve straight lines and plane.

4. Discrete Cross Correlation

- I ended up on implementing the two dimensional cross correlation, as I found it rather easy to comprehend (though a bit hard to compute with my tools).
- This Calculation can be thought of the convolution of a kernel over an image divided by the square root of the matrix Multiplication of the kernel squared times the square of the matrix multiplication of the vector the kernel is over.
  - This can be expressed as $r(x,y) = \frac{\sum_{x',y'}(Img(x',y') \times Ker(x+x',y+y'))^2}{\sqrt{\sum_{x',y'}(Img(x',y')^2 \times \sum x',y'(Ker(x+x',y+y')^2))}}$

– Intuitively this makes sense as a cross correlation, as if the image and kernel are similar at a sub-matrix, then that value would be close to one, and if they are further apart, the division would go towards closer to 0

2 Implementation of feature Detection

1. Lessons Learned

   - Not using a Fourier Transformer for a Cross Correlation is horribly slow, where even a 200x200~ image with a 25x25~ pixel takes around 30 seconds to compute and a 1200x800 image with a 112 x 107 kernel takes 30 minutes to compute
   - Also the Cross Correlation on its own is rather frail, or I ended up screwing up the logic on it

2. By hand

   - My very first attempt was rather silly, as I just took the convolution of two images, I won't say much more, outside of my first attempt being a failure.
   - My second attempt ended up implementing the Cross Correlation, however I needed to setup some tools first before I can compute it
     – My first tool was padArray, as I can't just rely on the convolution tool I ended up relying on for all previous computations

```
padArray :: Source r e ⇒ Int → Int → Array r DIM2 e → Array D DIM2 e
padArray row col arr = R.backpermute (Z :. i + 2 * row  :. j + 2 * col)
                                     getClosest arr
  where
    Z :. i :. j = R.extent arr
    getClosest (Z :. row' :. col') = ix2 roff coff -- col' - col is for getting
     where coff = min (max (col' - col) 0) (i - 1) -- the pos in the old array
           roff = min (max (row' - row) 0) (j - 1) -- from the new coordinates
```

     ∗ This function takes an array and creates a new array with the top and sides padded by that amount.

     – With this tool in hand we can compute the normalizedConvolution

```
normalizedConv :: Array r DIM2 Double → Array U DIM2 Double → m (Array U DIM2
normalizedConv arr ker = do
  let Z :. ik :. jk = R.extent ker
  arrExtended      ← R.computeUnboxedP $ padArray (ik `div` 2) (jk `div` 2) ar
  normKern         ← mmultP ker ker
  let normKernSum   = normKern `deepSeqArray` sumAllS normKern
  let extracted     = repaExtractWindows2D ik jk arrExtended
```

```
let fn subarr     = sumAllS (subComp *^ ker)
                    / √ (normKernSum * sumAllS (mmultS subComp subComp))
      where subComp = computeUnboxedS subarr
R.computeUnboxedP $ R.map fn extracted
```

* This function is a bit more involved, however it should resemble the verbal description of the cross correlation in section 1.
* The functions that end in P get processed in parallel, as being parallel on such a slow algorithm is paramountl
* extractWindows2D is how I get an array with subarray the same size as our kernel, I run this on the padded array, as from here we can just apply the cross correlation algorithm.

– Now that we have our normalizedConvolution function, we can now calculate the imageCorrelation.

```
imageCorrelation ::  Double → FilePath → FilePath → IO (Array U DIM2 Double)
imageCorrelation min path1 path2 = do
  x     ← readIntoRepa path1 >>= toGreyP
  y     ← readIntoRepa path2 >>= toGreyP
  let x' = R.map fromIntegral x
  y'    ← R.computeUnboxedP $ R.map fromIntegral y
  convd ← normalizedConv x' y'
  computeUnboxedP $ transpose $ filterMin convd min
```

* Here we load the two images and turn them grey.
* Then we just normalize the two functions and remove all pixels that are below a certain value.

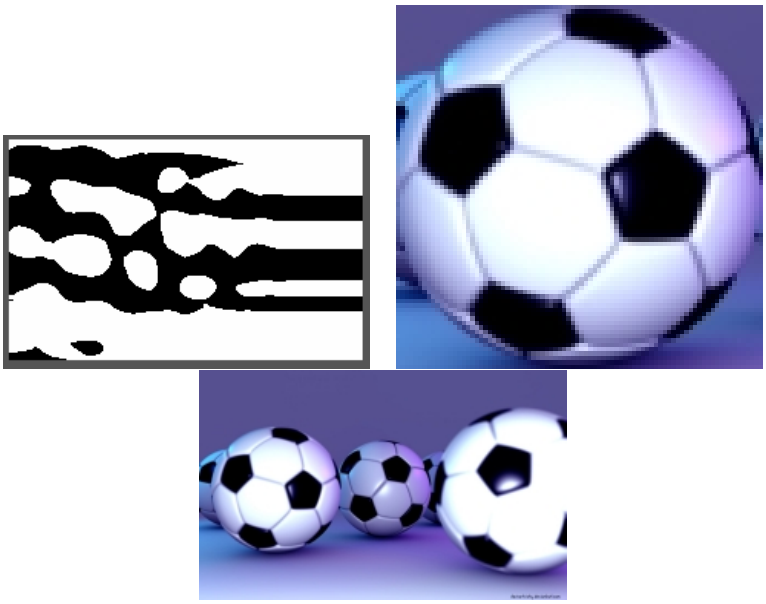
– We can now run this algorithm. Sadly this algorithm doesn't actually produce the results we want.

* I ran the kernel on the single soccer ball seen below on the image full of soccer balls
* The output is the black and white blob which is obviously an incorrect detection. Furthermore it is all black if I turn the measly 0.012 minimum value to 0.02

- Sadly I haven't figured out what went wrong in my algorithm. It might just be the case that the cross correlation algorithm is not supposed to used in such a matter and is thus rather frail for what I'm trying it for.

3. OpenCV

- This implementation had it's own set of pains, which will bet discussed more at the end.
- I'm still not fully sure how to manipulate the Mat data structure provided by OpenCV.



```
cvMatrix8 = fmap (fromImage . extractLumaPlane) . loadRGBJPG

cvMatrix16 = fmap (fromImage . extractLumaPlane) . loadRGB16


correlate arr ker = do
  arr' <- cvMatrix8 arr
  ker' <- cvMatrix8 ker
  let correlation = runExcept $ matchTemplate arr'
                                              ker'
                                              MatchTemplateCCoeff
                                              MatchTemplateNormed
  case correlation of
    Left _ -> error "erorr in transformation"
    Right m -> CV.withWindow "test" $ \win -> CV.imshow win arr'
```

- thankfully Haskell's OpenCV offers me a frmImage function which allows me to transform my image into a openCV format.
- I've tried a byte String transformation as well, but sadly that caused issues with the matchTemplate function openCV provides.
- the correlate function is where the magic occurs, as we read in an array and a kernel and computes the cross correlation between them. Note that this is the same algorithm as the one I outlined above, however there are many variations that openCV is willing to work with

- from here we just read the correlation into an openCV window where it is supposed to be displayed.
  - I couldn't easily turn this back into an image due to the double type and my inability to try to coerce the type
- Sadly, it seems the openCV bindings given are broken (at last on my machine), as cvMatrix8 actually gives me an array out of bounds errors for whatever reason, and I don't see an easy way to fix it. So sadly this approach too leads in failure