project_report_final

May 5, 2023

0.0.1 CS/ECE/ISyE 524 — Introduction to Optimization — Spring 2023

1 Image Distance Metric: Dynamic Optimization Based L_2

Cole Dilanni (diianni@wisc.edu)

Ilay Raz (iraz@wisc.edu)

Nitzan Orr (nitzan@cs.wisc.edu)

0.0.1 Table of Contents

1.1 1. Introduction

Overview Our project is designing an image distance metric which takes two images and finds how to warp one to best align with the other using mixed integer programming (MIP). We achieve our results by assigning each pixel binary variables which determine where in the warped image the pixel will map.

Image similarity algorithms are important for many tasks such as image recognition, image retrieval, and object tracking. The most simple image distance is the pixel-wise L_2 distance in which two image matrices of the same size are subtracted from each other and the resulting L_2 is their dissimilarity. An image similarity algorithm should return a higher value for pairs of images considered dissimilar and a lower value for images which are very similar/the same.

One problem with a basic L_2 distance metric is the "rigidness" of the pixels. This is to say that two images are only considered similar if their values are similar and their pixel positioning matches exactly. A failure case would be a checkerboard image compared to the same image shifted right one pixel. Even though the content remains the same (there has only been a small translation), the L_2 distance metric would return an extremely high value since the pixel positioning does not perfectly align.

Related Works A central problem in computer vision is determining the distance between images. Many previous works have tried to provide intuitive results. Notable works include the tangent distance (Simard, 1993) and the Hausdorff distance (Huttenlocher, 1993). However, from the image recognition point of view, they suffer from a number of technical issues. Previous works have also proposed techniques such as Histogram Interaction similarity measures, Invariant Moments measure (similarity between edges), and Local Edge Representation measures (similarity between image gradient boundaries). A more promising related work has been the Image Euclidean Distance (IMED) proposed by (Li, 2008). It recognizes the issue with simple L_2 distance measure and proposes several improvements to it. Li takes into account not just the distance between individual pixels, but also their neighborhood which results in improved performance but not generalizable results. (Wang 2005) proposed an improvement to IMED, an Adaptive Image Euclidean Distance which better fits images of varying pixel intensity. Our method seeks to borrow ideas form Euclidean Distance measures and use optimization-based computation.

Proposed Method To this end, we propose a more general L_2 distance metric for computer vision which allows the pixels of one image to shift in a way which best aligns with the comparison image. In our method, pixel shifts must follow certain rules when shifting an image A to best align with image B: - All pixels from A must map to a point in the shifted version of A - All pixels in B must have a corresponding value in the shifted version of A - Pixels cannot cross during shift ($Apixel_{left}$ cannot be further right than $Apixel_{right}$ in the final shifted version of the image)

To test our algorithm, we will use the MNIST dataset (dataset for handwritten digits 0-9). This dataset is good because it's low resolution (for fast testing), and handwritten digits have the same semantic information, but are often warped compared to one another, given handwriting differences.

Another way we test our method is by comparing it to the output of optical flow. Optical Flow is an algorithm in computer vision which tracks the movement of pixels between consecutive, similar images. As both methods create a flow field of pixel displacement, we show our method and discuss how it compares to optical flow.

Our report is organized as follows: Section 2 is the Mathematical Model and a detailed description of the various parts of the MIP metric. Section 3 is our Julia implementation of the model, which we invite the reader to run. In section 4 we discuss the results of our method and show it's performance on a number of test cases. Further, we compare our method to optical flow. We also discuss the limitations of our method and what future opportunities exist. Finally, in section 5 we conclude our discussion and summarize our key take-aways.

1.2 2. Mathematical model

The problem of finding optimal image distance metrics comes from computer vision. Our implementation of this optimization problem will make use of MIP.

We formulate the problem as: Let P be the set of all (x, y) pixel coordinates, and I_S be the source image values, and I_T be the destination image, both indexed by P. We define $SHIFT$ to be a matrix of binary decision variables for each (x, y) coordinate representing if the source pixel will be shifted by (δ_x, δ_y) to match the target image.

The first constraint represents the requirement that all source pixels are used at least once, the second constraint represents the requirement that all destination pixels are mapped to at least once, the third constraint requires that no pixel can map outside of the image area, and the last set

of constraints enforce that the relative order of the pixels is maintained: every pixel to the left of another one can not be shifted to a pixel which is to the right of the destination of the other pixel (and so too for pixels above/below one another). This is to say, pixels must not criss-cross.

The objective function tries to minimize the squared difference between the source pixels and the destination pixels in their mapped locations. There is also a regularization term in the objective which minimizes the number of shifts being used. This was to prevent the source image from unnecessarily mapping pixels to more destination pixels than necessary, as would be the case when the entire destination area is the same value of the source pixel. We enforce this constraint by minimizing the number of pixel shifts multiplied by a small factor ϵ. ϵ is used to allow the model to first focus on minimizing the difference between the two images and then focus on removing unnecessary shifts.

$$\min_{SHIFT_{x,y,\delta_x,\delta_y}} \sum_{(x,y,\delta_x,\delta_y)} (I_S(x,y) - I_T(x+\delta_x, y+\delta_y))^2 \cdot SHIFT_{x,y,\delta_x,\delta_y} + \epsilon * SHIFT_{x,y,\delta_x,\delta_y}$$

S.T.

$$\sum_{\delta_x,\delta_y} SHIFT_{x,y,\delta_x,\delta_y} \geq 1 \qquad \forall (x,y) \in$$

$$\sum_{P_x,P_y} SHIFT_{x,y,\delta_x,\delta_y} | x+\delta_x = P_x, y+\delta_y = P_y \geq 1 \qquad \forall (P_x,P_y$$

$$SHIFT_{x,y,\delta_x,\delta_y} | x+\delta_x < 1 \text{ or } x+\delta_x > ImgSize_x \text{ or } y+\delta_y < 1 \text{ or } y+\delta_y > ImgSize_y = 0 \qquad \forall (x,y) \in$$

$$\sum_{\delta_x,\delta_y} SHIFT_{x-\delta_x,y-\delta_y,\delta_x,\delta_y} \geq 1 \qquad \forall (x,y) \in$$

$$SHIFT_{x,y,\delta_x,\delta_y} > SHIFT_{x-1,y,\delta_x+i,\delta_y} \qquad \forall i \geq 2$$

$$SHIFT_{x,y,\delta_x,\delta_y} > SHIFT_{x,y-1,\delta_x,\delta_y+i} \qquad \forall i \geq 2$$

$$SHIFT_{x,y,\delta_x,\delta_y} > SHIFT_{x-1,y-1,\delta_x+i,\delta_y+j} \qquad \forall i \geq 2 \text{ or}$$

$$SHIFT_{x,y,\delta_x,\delta_y} > SHIFT_{x-1,y+1,\delta_x+i,\delta_y-j} \qquad \forall i \geq 2 \text{ or}$$

$$SHIFT_{x,y,\delta_x,\delta_y} \in \{0,1\}$$

1.3 3. Solution

First, we create sample input and output images of a checkerboard with black border. One of the images has the internal of the board shifted over by 1 column.

[4]:
```
# Make checkerboard as an example image, with one shifted one column to the right
origin = zeros(Int32, 12, 12)
origin[2:2:11, 2:2:10] .= 1
origin[3:2:11, 3:2:10] .= 1

destination = zeros(Int32, 12, 12)
destination[2:2:11, 3:2:11] .= 1
destination[3:2:11, 4:2:11] .= 1

origin[1, :] = origin[12, :] = origin[:, 12] = origin[:, 1] .= 1
destination[1,:]=destination[12,:]=destination[:,12]=destination[:,1] .= 1
```

```
;
```

Next, we create a function to generate our model for the given input and output images:

[5]:
```
using JuMP, Gurobi
function getDistance(img1, img2, ws, epsilon)
    image_width, image_height = size(img1)
    xIter, yIter, fIter = 0:(image_width-1), 0:(image_height-1), 0:(2*ws)

    m = direct_model(Gurobi.Optimizer())

    # Variable for each point, and each potential destination of each point
    @variable(m, v[xIter, yIter, fIter, fIter], Bin)

    # Objective to minimize change of color between img1 and img2 with epsilon␣
↪to minimize total change
    @objective(m, Min, sum(
            (img1[x+1, y+1] - img2[x + 1 + (xs - ws), y + 1 + (ys  - ws)])^2 *␣
↪v[x, y, xs, ys]
            for x in xIter, y in yIter, xs in fIter, ys in fIter
            if x + (xs - ws) >= 0 &&
               x + (xs - ws) < image_width &&
               y + (ys - ws) >= 0 &&
               y + (ys - ws) < image_height
        ) + epsilon * sum(v)
    )

    column(x) = Cint(Gurobi.column(backend(m), index(x)) - 1) # Magic function␣
↪to call Gurobi C-layer API with variables by index

    function addLeftConstraint(x, y, xs, ys)
        if xs != 2*ws && xs != 2*ws-1 # left pixel can't be more right than curr␣
↪pixel
            max1 = @variable(m, binary=true)
            max2 = @variable(m, binary=true)
            max3 = @variable(m, binary=true)
            v1 = [v[x,y,xss,yss] for xss in 0:xs, yss in fIter]
            v2 = [v[x-1,y,xss,yss]  for xss in (xs+2):(2*ws), yss in fIter]
            v3 = [max1, max2]
            Gurobi.GRBaddgenconstrMax(backend(m), "", column(max1), length(v1),␣
↪column.(v1), 0)
            Gurobi.GRBaddgenconstrMax(backend(m), "", column(max2), length(v2),␣
↪column.(v2), 0)
            Gurobi.GRBaddgenconstrMax(backend(m), "", column(max3), length(v3),␣
↪column.(v3), 0)
            @constraint(m, max1 + max2 <= max3)
        end
```

```
    end

    function addDownConstraint(x, y, xs, ys)
        if ys != 2*ws && ys != 2*ws-1
            max1 = @variable(m, binary=true)
            max2 = @variable(m, binary=true)
            max3 = @variable(m, binary=true)
            v1 = [v[x,y,xss,yss] for xss in fIter, yss in 0:ys]
            v2 = [v[x,y-1,xss,yss] for xss in fIter, yss in (ys+2):(2*ws)]
            v3 = [max1, max2]
            Gurobi.GRBaddgenconstrMax(backend(m), "", column(max1), length(v1),␣
↪column.(v1), 0)
            Gurobi.GRBaddgenconstrMax(backend(m), "", column(max2), length(v2),␣
↪column.(v2), 0)
            Gurobi.GRBaddgenconstrMax(backend(m), "", column(max3), length(v3),␣
↪column.(v3), 0)
            @constraint(m, max1 + max2 <= max3)
        end
    end

    function addDownLeftConstraint(x, y, xs, ys)
        if xs != 2*ws &&  xs != 2*ws-1
            max1 = @variable(m, binary=true)
            max2 = @variable(m, binary=true)
            max3 = @variable(m, binary=true)
            v1 = [v[x,y,xss,yss] for xss in 0:xs, yss in fIter]
            v2 = [v[x-1,y-1,xss,yss] for xss in (xs+2):(2*ws), yss in fIter]
            v3 = [max1, max2]
            Gurobi.GRBaddgenconstrMax(backend(m), "", column(max1), length(v1),␣
↪column.(v1), 0)
            Gurobi.GRBaddgenconstrMax(backend(m), "", column(max2), length(v2),␣
↪column.(v2), 0)
            Gurobi.GRBaddgenconstrMax(backend(m), "", column(max3), length(v3),␣
↪column.(v3), 0)
            @constraint(m, max1 + max2 <= max3)
        end
        if ys != 2*ws && ys != 2*ws-1
            max1 = @variable(m, binary=true)
            max2 = @variable(m, binary=true)
            max3 = @variable(m, binary=true)
            v1 = [v[x,y,xss,yss] for xss in fIter, yss in 0:ys]
            v2 = [v[x-1,y-1,xss,yss] for xss in fIter, yss in (ys+2):2*ws]
            v3 = [max1, max2]
            Gurobi.GRBaddgenconstrMax(backend(m), "", column(max1), length(v1),␣
↪column.(v1), 0)
```

```
            Gurobi.GRBaddgenconstrMax(backend(m), "", column(max2), length(v2),␣
↪column.(v2), 0)
            Gurobi.GRBaddgenconstrMax(backend(m), "", column(max3), length(v3),␣
↪column.(v3), 0)
            @constraint(m, max1 + max2 <= max3)
        end
    end

    function addUpLeftConstraint(x, y, xs, ys)
        if xs != 2*ws && xs != 2*ws-1
            max1 = @variable(m, binary=true)
            max2 = @variable(m, binary=true)
            max3 = @variable(m, binary=true)
            v1 = [v[x,y,xss,yss] for xss in 0:xs, yss in fIter]
            v2 = [v[x-1,y+1,xss,yss] for xss in (xs+2):(2*ws), yss in fIter]
            v3 = [max1, max2]
            Gurobi.GRBaddgenconstrMax(backend(m), "", column(max1), length(v1),␣
↪column.(v1), 0)
            Gurobi.GRBaddgenconstrMax(backend(m), "", column(max2), length(v2),␣
↪column.(v2), 0)
            Gurobi.GRBaddgenconstrMax(backend(m), "", column(max3), length(v3),␣
↪column.(v3), 0)
            @constraint(m, max1 + max2 <= max3)
        end
        if ys != 0 && ys != 1
            max1 = @variable(m, binary=true)
            max2 = @variable(m, binary=true)
            max3 = @variable(m, binary=true)
            v1 = [v[x,y,xss,yss] for xss in fIter, yss in ys:(2*ws)]
            v2 = [v[x-1,y+1,xss,yss] for xss in fIter, yss in 0:(ys-2)]
            v3 = [max1, max2]
            Gurobi.GRBaddgenconstrMax(backend(m), "", column(max1), length(v1),␣
↪column.(v1), 0)
            Gurobi.GRBaddgenconstrMax(backend(m), "", column(max2), length(v2),␣
↪column.(v2), 0)
            Gurobi.GRBaddgenconstrMax(backend(m), "", column(max3), length(v3),␣
↪column.(v3), 0)
            @constraint(m, max1 + max2 <= max3)
        end
    end

    # add constraints that if a pixel shift is outside image, it must be 0
    isOutside(x, xs, h) = x + (xs - ws) < 0 || x + (xs - ws) > h - 1
    for x in xIter, y in yIter, xs in fIter, ys in fIter
        if  isOutside(x, xs, image_width) || isOutside(y, ys, image_height)
            @constraint(m, v[x, y, xs, ys] == 0)
```

```
        end
    end

    # constraints that all source pixels must shift in at least 1 direction
    @constraint(m, mustShiftC[x in xIter, y in yIter], sum(v[x,y,:,:]) >= 1)


    # constraints that all dest pixels must be mapped to
    isInside(x, xs, h) = x - (xs - ws) >= 0 && x - (xs - ws) < h
    @constraint(m, ontoC[x in xIter, y in yIter],
        sum(v[x - (xs - ws), y - (ys - ws), xs, ys] for xs in fIter, ys in fIter
                if isInside(x, xs, image_width) && isInside(y, ys, image_height)
            ) >= 1
    )

    for x in xIter, y in yIter, xs in fIter, ys in fIter
        # Set initial values
        if xs == ws && ys == ws
            set_start_value(v[x, y, xs, ys], 1)
        else
            set_start_value(v[x, y, xs, ys], 0)
        end

        # No part constraint
        for xn in max(0, x-1):min(image_width - 1, x+1), yn in max(0,y-1):
↪min(image_height - 1, y+1)
            if xn != x || yn != y
                @constraint(m,
                    v[x,y,xs,ys] <= sum(v[xn, yn, xss, yss] for xss in max(0,␣
↪xs-1):min(2*ws, xs+1), yss in max(0,ys-1):min(2*ws,ys+1))
                )
            end
        end
        if y == 0 && x == 0
            continue
        elseif y == 0 # Left constraint
            addLeftConstraint(x, y, xs, ys)
        elseif x == 0 # Down constraint
            addDownConstraint(x, y, xs, ys)
        else
            addLeftConstraint(x, y, xs, ys)
            addDownConstraint(x, y, xs, ys)
            addDownLeftConstraint(x, y, xs, ys)
        end
        if y != image_height-1 && x != 0
            addUpLeftConstraint(x, y, xs, ys)
        end
```

```
    end

    return m
end
;
```

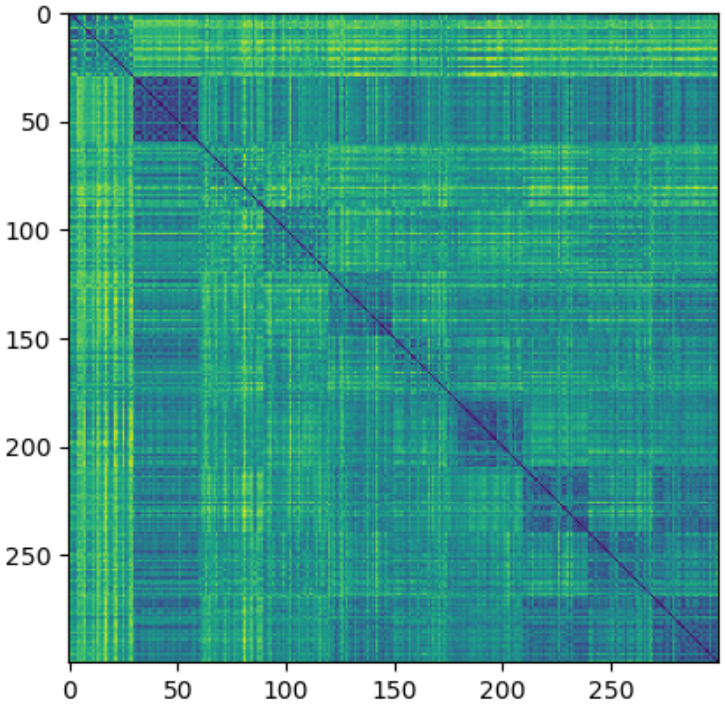
Finally, we run the model on the given images:

[ ]:
```
m = getDistance(origin, destination, 1, 0.01);
optimize!(m)
```

1.4 4. Results and discussion

1.4.1 4.A. Image Similarity

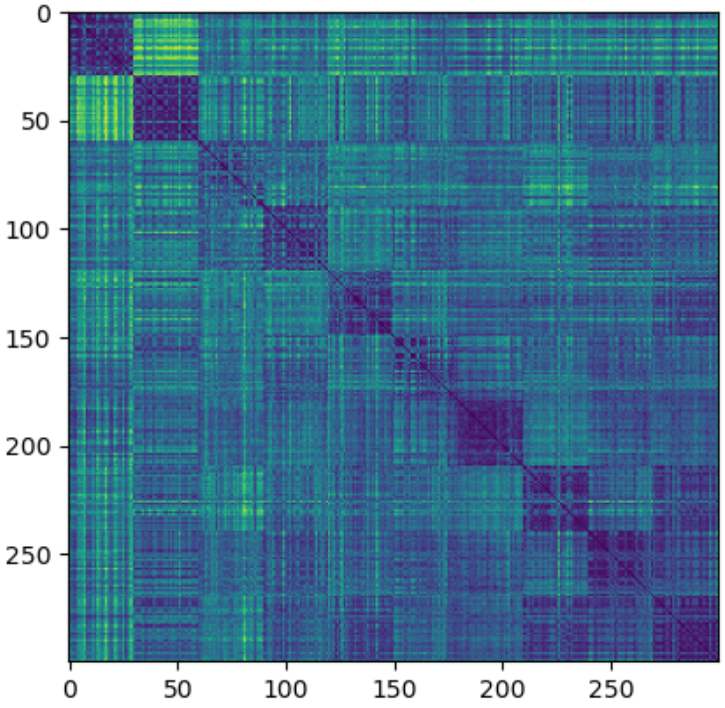
To test our optimization model, we found the difference between handwritten digit images in the MNIST dataset. We used 300 images total, with 30 images from each digit sorted 0-9. First, the image-wise differences were calculated using a basic L_2 difference. These distances were made into a similarity matrix below, where the value at index $[i, j]$ is the difference between image i and image j. Higher values are yellow, while lower values are blue. We observed that the distances between images of "1"s was most consistently low, while numbers such as "0", "2", "5", and "8" were found to have a high L_2 distance between images of the same class. This can be attributed to the lower variabilty of how participants could write "1".



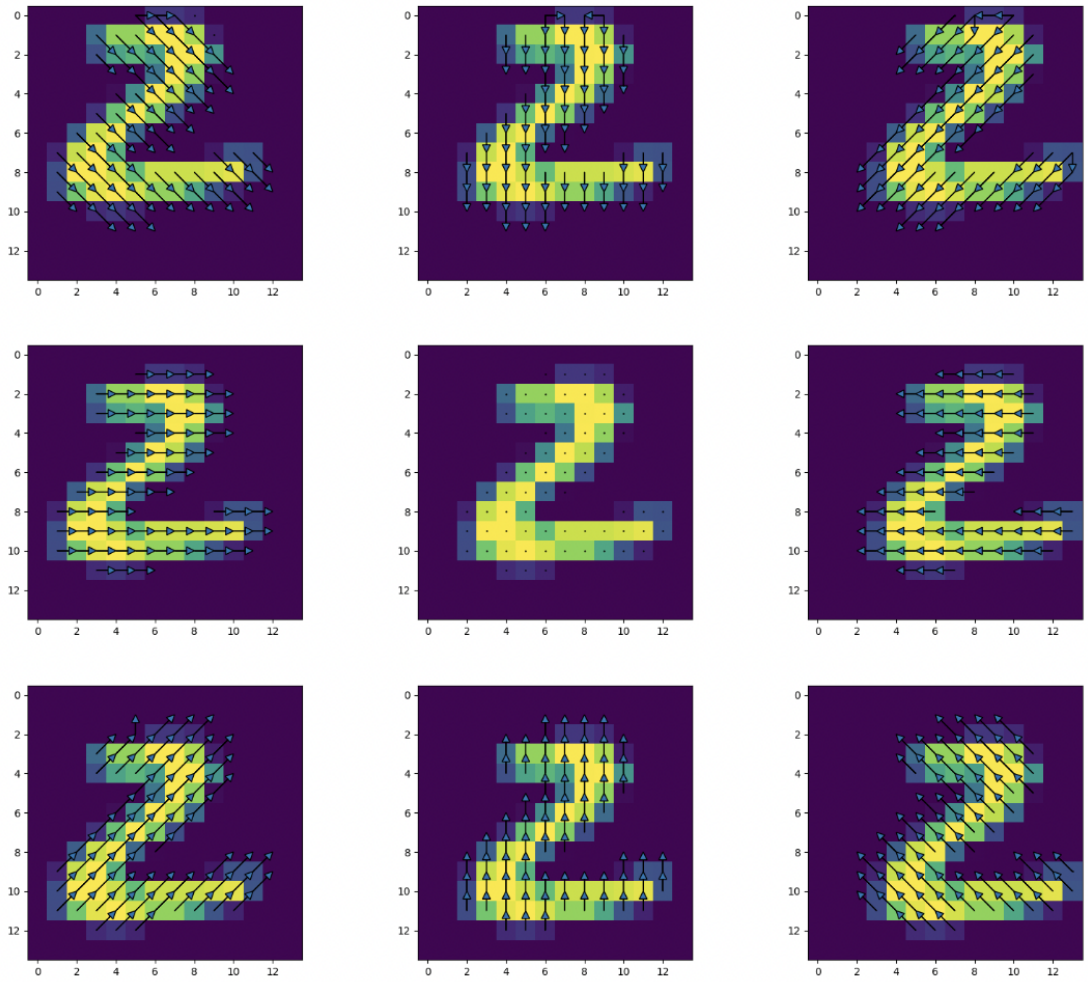
We then found the image-wise differences using our method. When using our model to find the

image similarity between image A and image B, we pass the images into the model in both orders, first mapping image A to B then vice versa. We then take the maximum image difference found as the final difference between the images. This image similarity matrix kept the low distances between images of "1"s, but also made many other digits more similar to each other. This is likely because our model corrected any small handwriting differences between images to better align images with semantically similar content.

One finding to note is that the "0" and "1" digits were found to be the most different from each other, which is consistent with human perception of the digits. Overall, the difference matrix values were lower than that of the L_2 distance metric because our model acts as a pseudo-L_2 difference, but optimizes the pixel placement to minimize the image difference. It is only possible for our model to achieve approximately the same or lower values in the similarity matrix as the baseline L_2 distance method since an image where no pixels are shifted (the L_2 difference) is within the feasible set of our model.



To confirm the optimization model works as intended, we translate a single digit image by one pixel in all directions and map it to the original image. Since the images of the MNIST dataset have some background padding, each image should be able to map to the original without difficulty (having all pixels moving in the same direction except for some border pixels). In the example below, we display the pixel movement of pixels with value greater than 0 (not part of the background) at the various translations. We observe that the pixel displacement correctly aligns the displaced images with the original. Also, the image without displacement correctly does not shift any pixels.



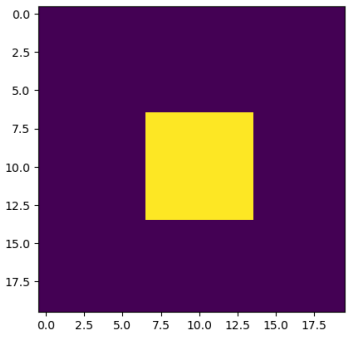
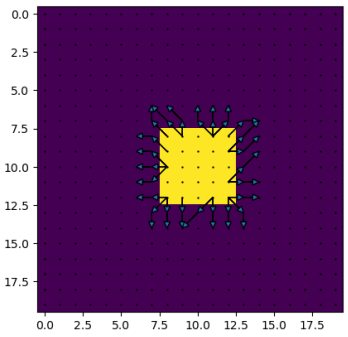
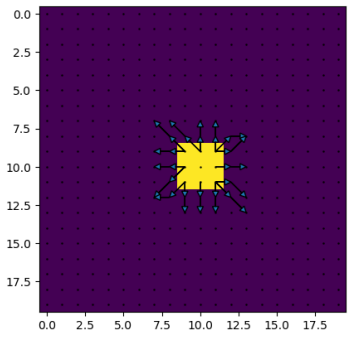
One limitation of our approach is scalability. We had to confine the possible window where each pixel could shift to the 9x9 area directly around the starting location. We also had to downsize the MNIST images to 14x14 to ensure the runtime of our algorithm was less than 1 min per distance calculation. In practice, it is expected that pixel content will shift more than 1 pixel in any direction and that the images being compared will be much higher resolution (1000's of pixels)

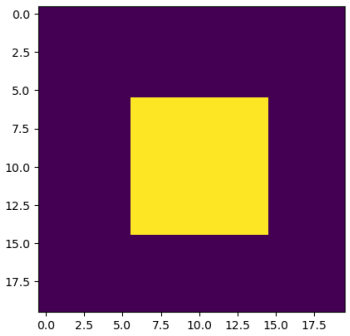
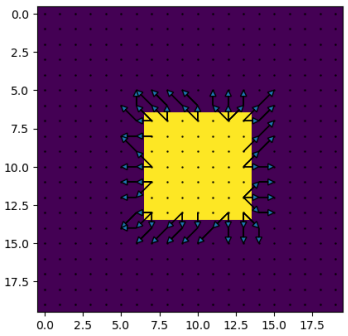
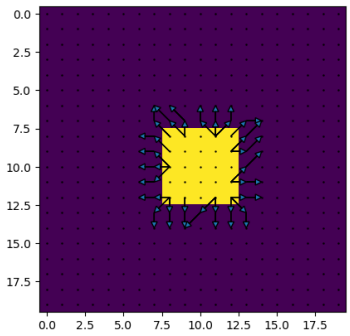
An assumption that our approach makes is that pixels do not criss-cross. However, this assumption only holds if images are similarly oriented. When comparing the distance between two images which are mirrored or rotated, our model will not work as intended because the pixels are not allowed to change their relative positioning with respect to the x and y axes. Considering a horizontally flipped image, it is clear that the optimal repositioning of pixels should be to cross all pixels with respect to the x axis.

1.4.2 4.B. Image Transformation

We further tested our algorithm on image transformation with synthetic data showing transformations such as scaling. Our method can be used as an optimization-based method to compute pixel velocities in videos. We compare our algorithm with the commonly-used pixel-tracking algorithm, Optical Flow. We show test cases below of the output of the pixel movement necessary to achieve the Target Object Size. Our algorithm performs similar to optical flow for this synthetic data.

Although, we note that optical flow does not map to all pixels in the destination image while our method does.

Target Size 1	Our Method	Optical Flow
		

Target Size 2	Our Method	Optical Flow
		

These test cases show how the pixels in the source image need to move in order to transform to the target image. An arrow for each pixel determines how that pixel in the source image should move in order to attain the target image. No Arrow means no movement is necessary. One peculiarity of our method is that a single pixel can spread itself across its neighboring region. As can be seen in the middle column, certain pixels map into two or three different neighboring locations, expanding the pixel. In contrast, optical flow relies on the assumption that a pixel in the source image will be found (along with its neighbors) in one location in the subsequent image. This requirement limits optical flow to consider a one-to-one mapping whereas our method is more flexible and supports one-to-many pixel mappings. Like Optical Flow, our method determines the magnitude and direction of pixel movement, thus allowing velocity computation on dynamic scenes. Our method can be used as a replacement to optical flow. Currently, our method is not nearly as performant, as optical flow can run at real-time speeds (>30 FPS) on larger images.

1.5 5. Conclusion

We have proposed a novel Mixed Integer Programming Optimization-based image distance metric. Our method successfully showed how it can align similar looking images in order to aid with image similarity metrics and optical flow computation. First, our digit similarity matrix shows a proof of

concept that this method can be used to improve digit classification if a user was to use k-nearest-neighbor classification. Second, our method can compute pixel-level movement between subsequent images which makes it a candidate replacement to the commonly-used optical flow algorithm. Lastly, this method of image alignment has applications as a general image distance between various images which involve non-global stretching/expanding of the image. Another example use case is image registration for biomedical images, since tissue samples are commonly stretched between sample images, but maintain the global positioning of features relative to each other.

Moving forward we would like to expand our method by making the model more efficient and scalable. Our runtime seems to increase exponentially with respect to the image and search window sizes, making it infeasible for high resolution images. We would also like to consider how our method could be modified to handle cases involving image flipping/rotation. One potential idea comes from Scale-Invariant Feature Transform (SIFT) features, in which the orientation of the image would first be calculated, and then the pixel shifts would be constrained with respect to the orientation of the image instead of the x and y axes.

[ ]: