Handwritten Character Recognition

Gabriel Chapel

Abstract

In this report, the linear program (LP), classification method, and testing algorithm used for identifying handwritten numbers are presented along with the results and corresponding accuracy levels. Manipulations to the process are also provided to display the relationship between accuracy and run-time.

1 Introduction

As technology continues to become more intelligent in modern society, more efficient methods for completing everyday jobs present themselves. One example of this can be observed in the current mail delivery system, which is growing in popularity due to the increasing accessibility of online shopping. This popularity calls for high speed and high accuracy, which largely relies on the ability of the delivery system, including sorting the mail. Unfortunately, this often includes human involvement, since people are sometimes the ones addressing the mail, which significantly reduces the predictability of the process. Human handwriting can vary substantially from one person to another but the system needs to be able to read and recognize the addresses. Here, we discuss methods of recognizing handwritten numbers and how the efficiency varies with accuracy.

Linear classification is a method used to autonomously determine which class an unknown object should be categorized into, e.g. what number an image is representing. When comparing only two classes, the problem is *binary*, but when comparing more than two, the problem is *multiclass*. In order to categorize the unknown items, there must be some known, or at least measurable, *features*, e.g. image characteristics. The features relate to the

dimension of the problem, where a two-dimensional problem has two features and a multidimensional problem has more than two. This also changes the separator, since a line is used in two dimensions to distinguish the classes but a hyperplane is used in more dimensions. In this case, we want to differentiate digits, so we have a multiclass problem with 10 different categories: the numbers 0 through 9.

For this problem, we use images of individual, handwritten numbers as the features, or more specifically, the pixels comprising the images. An example of an image is shown in Figure 1 below. Each image is treated as a 28×28

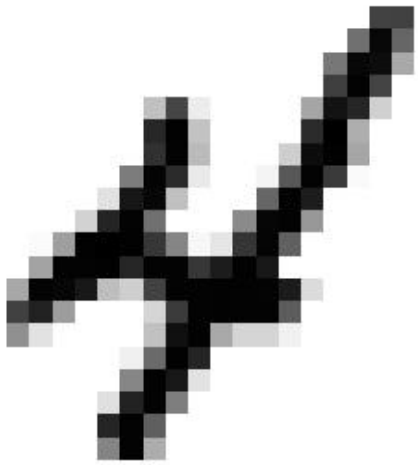

Figure 1: 28×28 Pixel Representation Number 4

matrix of grayscale pixels, so each element in the matrix contains a value in the range 0-255. For computational purposes, however, each image is reshaped into a 784×1 vector. This just allows multiple images to be used

for training and testing more easily, which is discussed in more detail later. The feature of each image, though, is the 784-element vector of pixel values:

$$\boldsymbol{x}_i = \begin{bmatrix} x_1^i & x_2^i & \dots & x_{784}^i \end{bmatrix}^T \tag{1}$$

where i represents the number of the image. This means we need to use a hyperplane to separate the classes. [2]

Since there are 10 classes, this problem is inherently multiclass. However, we can treat it, instead, as a collection of several binary classifiers, so instead of classifying between one number and nine others all at once, we can classify between just two at a time. In other words, we can classify between 0 and 1, 0 and 2, 0 and 3, and so on until all pairings have been made, up to a total of 45 classifiers. The architecture used for accomplishing this is known as Decision Directed Acyclic Graph (DDAG) and is discussed in more detail in a later section.

2 LP Formulation

With the classes and features identified, an LP can be formulated to define the hyperplanes. For a single binary classifier, this hyperplane can be expressed as,

$$\boldsymbol{a}^T\boldsymbol{x}_i - b = 0 \tag{2}$$

where $\boldsymbol{a}$ is the same size as $\boldsymbol{x}$ from Equation 1 and b is a scalar. This is used to divide the problem into the two classes as follows:

$$\boldsymbol{a}^T\boldsymbol{x}_i - b > 0 \tag{3a}$$

$$\boldsymbol{a}^T\boldsymbol{x}_i - b < 0 \tag{3b}$$

In the case of classifying between two numbers, for example 0 and 9, we assume that the region described by Equation 3a eliminates the larger number but **may** containthe smaller number and the region described by Equation 3b eliminates the smaller number but **may** contain the larger number. We emphasize the word "may" because that conclusion can only be made when considering only one binary classifier. That is, when dealing only with one binary classifier, the regions defined by 3a and 3b contain the smaller number and the larger number, respectively. Since this problem requires several

binary classifiers, though, this statement is only true after a process of elimination discussed further in the next section. As of now, the LP still needs some fine-tuning.

To reduce the flexibility of the hyperplane and the influence of outliers as much as possible, we include a margin. Equations 3a and 3b become

$$\boldsymbol{a}^T\boldsymbol{x}_i - b > 1 \tag{4a}$$

$$\boldsymbol{a}^T\boldsymbol{x}_i - b < -1 \tag{4b}$$

so that the data points for each class are divided more precisely. If we maximize the margin, which is equal to $\frac{2}{||a||_2}$, we can ensure that there is only one hyperplane that divides the classes as evenly as possible and reduces the effects of outliers. Thus, $||a||_2$ will be included in the final LP.

We also need to include slack variables u and v to improve the accuracy of the separation. This transforms the above equations again into

$$\boldsymbol{a}^T\boldsymbol{x}_i - b > 1 - u \tag{5a}$$

$$\boldsymbol{a}^T\boldsymbol{x}_i - b < -(1 - v) \tag{5b}$$

with $u > 0$ and $v > 0$.

Now, the objective is to maximize the accuracy while also maximizing the margin, so a weighting variable γ is introduced. This allows for adjusting the contribution of the accuracy factor and, ultimately, decides how much the slack variables affect the LP relative to the margin. With each of these parameters taken into consideration, the LP is written as follows:

$$\begin{array}{ll} \text{minimize} & ||\boldsymbol{a}||_2 + \gamma(\mathbf{1}^T\boldsymbol{u} + \mathbf{1}^T\boldsymbol{v}) \\ \text{subject to} & \boldsymbol{X}_{small}\boldsymbol{a} - b \geq \mathbf{1} - \boldsymbol{u} \\ & \boldsymbol{X}_{large}\boldsymbol{a} - b \leq -(\mathbf{1} - \boldsymbol{v}) \\ & \boldsymbol{u} > 0 \\ & \boldsymbol{v} > 0 \end{array} \tag{6}$$

For training and testing purposes, we use multiple images of each number, so, assuming that m_1 is the number of images of the small number in the

classifier and m_2 is the number of images of the large number, the features are defined below.

$$\boldsymbol{X}_{small} = \begin{bmatrix} x_1^1 & x_2^1 & \dots & x_{784}^1 \\ x_1^2 & x_2^2 & \dots & x_{784}^2 \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{m_1} & x_2^{m_1} & \dots & x_{784}^{m_1} \end{bmatrix} \tag{7}$$

$$\boldsymbol{X}_{large} = \begin{bmatrix} x_1^{m_1+1} & x_2^{m_1+1} & \dots & x_{784}^{m_1+1} \\ x_1^{m_1+2} & x_2^{m_1+2} & \dots & x_{784}^{m_1+2} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{m_1+m_2} & x_2^{m_1+m_2} & \dots & x_{784}^{m_1+m_2} \end{bmatrix} \tag{8}$$

The slack variables $\boldsymbol{u}$ and $\boldsymbol{v}$ are of length m_1 and m_2, respectively, and γ is a scalar that is modified after testing to obtain the best results. It is also important to note that, unlike in a typical LP, the variables for this LP are $\boldsymbol{a}$ and b, and the $\boldsymbol{X}$ terms are already known. This is why the first two constraints in Equation 6 are slightly reorganized from how they are represented in Equations 5a and 5b.

With the LP formulated, the variables can be determined using labeled data in a method called *training*.

3 Training

Machine learning problems, such as this, require an initial input of known, or labeled, data to define the coefficients of the separating hyperplanes. This means that the classification of each data point is already known and is used for solving the LP. For this problem, we use a modeling system for convex programming called cvx, with which we solve the LP without the need for initial matrix representation. The cvx modeler we use offers two core solvers, SeDuMi and SDPT3, for solving LPs but it is evident when testing even just one classifier that SeDuMi is more time efficient, with a run-time of 147 seconds versus the SDPT3 run-time of 3792 seconds. Thus, we assume the use of SeDuMi for the remainder of the report. We also use the cvx low-precision setting to speed up the process.[1]

The training set is much larger than the testing set so that coefficients can be built from a representative amount of data and the classes can be correlated with a range of features. We use a training set of 60,000 images for this problem, which is comprised of roughly 6,000 images for each number (0 through 9), making a $60,000 \times 784$ total feature matrix $\boldsymbol{X}$,

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{X}_0 \\ \boldsymbol{X}_1 \\ \vdots \\ \boldsymbol{X}_9 \end{bmatrix} \tag{9}$$

where each subscript represents the images' classification. This means $\boldsymbol{X}_0$ contains all of the images of the number 0. Although this lists the images in numerical order, the matrix used in the MATLAB code is in a random order, so a list of labels is also provided. The list of labels is used to form a cell array in the order of $\boldsymbol{X}$ above.

One useful trait of cvx is that it accepts LP input in non-matrix form, so we can directly type in the formulation from Equation 6. Doing this once provides the coefficients for a single classifier, e.g. 0 vs. 9, so we must use cvx 45 times to train all the classifiers. This is done by merely replacing $\boldsymbol{X}_{small}$ and $\boldsymbol{X}_{large}$ in Equation 6 by components from Equation 9, $\boldsymbol{X}_i$, until each $\boldsymbol{X_i}$ is paired with one another. The subscript i represents the images' classification 0-9. For the first iteration, we set $\gamma = 1$, and change it after obtaining the results from testing. After each LP is solved, the classifier's coefficients are placed in cell arrays $\boldsymbol{A}$ and $\boldsymbol{B}$, which are both formatted in the same way:

$$\boldsymbol{A} = \begin{bmatrix} [] & \text{'0 vs. 1'} & \text{'0 vs. 2'} & \dots & \text{'0 vs. 9'} \\ [] & [] & \text{'1 vs. 2'} & \dots & \text{'1 vs. 9'} \\ [] & [] & [] & \dots & \text{'2 vs. 9'} \\ [] & [] & [] & \dots & \text{'3 vs. 9'} \\ [] & [] & [] & \dots & \text{'4 vs. 9'} \\ [] & [] & [] & \dots & \text{'5 vs. 9'} \\ [] & [] & [] & \dots & \text{'6 vs. 9'} \\ [] & [] & [] & \dots & \text{'7 vs. 9'} \\ [] & [] & [] & \dots & \text{'8 vs. 9'} \end{bmatrix} \tag{10}$$

These coefficients define the separating hyperplanes and are then used for testing, as described in the next section.

4 Testing

The separating hyperplanes found in the previous section can be used to classify test data. The test set used is a $10,000 \times 784$ matrix, containing roughly 1,000 images of each number, 0 through 9. To classify the images, we need to figure out which side of the hyperplane the data is located, or which side the features point to. For this, we simply input the test set $\boldsymbol{X}_{test}$ into the hyperplane equation and check whether the result is positive or negative. Then we use MATLAB's find() function to find the corresponding indices in $\boldsymbol{X}_{test}$, as such:

$$\begin{aligned} \text{notLarge} &= find(\boldsymbol{X}_{test}\boldsymbol{A}_{i,j} - \boldsymbol{B}_{i,j} > 0) \\ \text{notSmall} &= find(\boldsymbol{X}_{test}\boldsymbol{A}_{i,j} - \boldsymbol{B}_{i,j} < 0) \end{aligned} \tag{11}$$

Here, the variables notLarge and notSmall indicate the number in the classifier that is eliminated, so if the result is on the positive side of the hyperplane, the image being tested does not depict the larger of the two numbers, and if the result is on the negative side, the image does not depict the smaller number. As explained previously, the classifier only eliminates one number since there are more than two classes that need to be considered, and won't indicate a specific classification until there are only two remaining. So if, for example, 0 and 9 are being compared, the indices comprising notLarge **do not** correspond to any images of the number 9 and the indices comprising notSmall **do not** correspond to any images of the number 0. The subscripts i and j correspond to the two numbers being compared in the classifier, so $i = 0$ and $j = 9$ in this same example.

To narrow down the classes, we use a Decision Directed Acyclic Graph (DDAG), which filters out all of the numbers that an image is not, until it reaches the number that it actually is. This is shown in Figure 3 in the Appendix, where the notSmall inputs are displayed by blue lines and the notLarge inputs are displayed by red lines. There are many ways to implement the DDAG, but our first step is to start at the classification between 0 and 9 and then increment the i value in Equation 11 by one, following the straight path to the comparison of 8 and 9. During each increment, we find new notSmall indices, which we use to establish a new test set for the next classification, reducing the amount of images being classified after each increment. We also obtain new notLarge indices and corresponding images that are stored in a cell array for future use. The notSmall indices obtained when

$i = 8$ (comparing 8 and 9) reveal which images are images of the number 9. This conclusion can be made because the classifications involving 9 only contain notSmall inputs, shown with only blue lines entering each classifier. This is not true for the rest of the classifiers, which also receive contributions from the notLarge inputs. This is why we have them stored for future use.

After completing the classification for $i = 8$, we decrease j from Equation 11 by one and repeat the process. This time, however, we need to include the notLarge inputs at each increment of i, so we append the stored notLarge images from the previous set of classifications to the current notSmall images and then update the stored values with the current notLarge images. When $i = 7$, the resulting notSmall images in addition to the stored notLarge images from the comparison between 8 and 9 (when $j = 9$) are the images of the number 8. This process is repeated, incrementally increasing i and decreasing j, until $j = 1$, when the classification reveals which images are images of 0. Notice that each incremental decrease of j results in a maximum i that is one less than the previous maximum.

With the images all classified, we can compare them to the original test set. Since they are in a different order, though, we use MATLAB's intersect() function to match the images with indices of the original test set. We then build a new array to hold the image numbers at each of these indices, formatting our solution like the given list of labels. We compare the two lists to find which results don't match and, more importantly, how many. With this, we can find the accuracy of the total multiclass classifier,

$$\text{accuracy} = \frac{10000 - (\text{Number of bad results})}{10000} \tag{12}$$

Using $\gamma = 1$, we obtain an accuracy of 91.83%, so we begin iterating to see if we can find a better result.

5 Iterations

The table below shows each iteration of γ with the corresponding accuracy and run-time. The run-time only measures the training of the classifiers, from defining the $\boldsymbol{X}_{small}$ and $\boldsymbol{X}_{large}$ to storing the coefficients in $\boldsymbol{A}$ and $\boldsymbol{B}$.

Table 1: Iterations of γ

γ	Accuracy	Run-time (sec)
1000	0.9130	10548
1	0.9183	10432
0.01	0.9212	10324
0.0001	0.9455	5821
0.00008	0.9464	5497
0.00006	0.9475	5494
0.00005	0.9468	5273
0.00004	0.9458	4961
0.00001	0.9378	3333

Though presented in order of decreasing γ, this is not actually how the iterations were performed. As stated previously, the first iteration was $\gamma = 1$. We then increased γ to 1000 to check the accuracy and found that the it decreased while the run-time increased. Thus, we tried decreasing γ to 0.01, which slightly improved both the run-time and the accuracy, so we continued to decrease γ to 0.0001. Again, this resulted in improvement, so we stepped further to 0.00001, where the results finally degraded again, suggesting that the best γ value is between 0.0001 and 0.00001. We iterated between these two values until we found the solution of $\gamma = 0.00006$, which we took as the best value.

We could continue to iterate in an effort to find an even better γ, but even varying it by 0.00001 only changes the accuracy by a tenth of a percent. Each iteration also takes a minimum of just under an hour, so any further iteration seems excessive. However, we do detail our efforts to manipulate the results in another way by *playing* with the problem.

6 Playing

As is evident by Table 1, training this classifier is a relatively long process, since the cvx iterates through about 12,000 equations (roughly 6,000 from both classes) and about 24,000 variables ($\boldsymbol{a}$, $\mathbf{1}b$, $\boldsymbol{u}$, and $\boldsymbol{v}$ for both classes) 45

times. Thus, to reduce the run-time, we have two options: eliminate some of the data points or eliminate some of the features. Since each image contains some amount of white-space surrounding the number, we choose to elimnate some features in an effort to reduce unnecessary data.

To illustrate how we accomplish this, we use a 5×5 pixel image as an example. This is shown below in the format that MATLAB uses to reshape arrays, where the numbers represent the row number if the matrix were reshaped into a 25×1 array.

$$\begin{bmatrix} 1 & 6 & 11 & 16 & 21 \\ 2 & 7 & 12 & 17 & 22 \\ 3 & 8 & 13 & 18 & 23 \\ 4 & 9 & 14 & 19 & 24 \\ 5 & 10 & 15 & 20 & 25 \end{bmatrix}$$

We assume that the number is relatively centered in the matrix, allowing us to remove the border. We define the indices we want to remove with the following vector:

$$remove = [\ 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 10 \quad 11 \quad 15 \quad 16 \quad 20 \quad 21 \quad 25\ ] \tag{13}$$

After removing these, the new image matrix becomes

$$\begin{bmatrix} 7 & 12 & 17 \\ 8 & 13 & 18 \\ 9 & 14 & 19 \end{bmatrix}$$

Though unnecessary in this 5×5 example, if we want to eliminate more white-space, we can just remove the next layer.

This is the process we follow to relate the change in accuracy to the change in run-time when varying the number of features. The table below displays how the results differ when removing layers of the 28×28 image's border. For these iterations, we maintain a constant weight $\gamma = 0.00006$.

Layers 3 and 5 are not included because the results show that the accuracy remains stable until about layer 9. We end the removal at layer 13

Table 2: Accuracy and Run-time After Removing Border Layers

Border Layers Removed	Accuracy	Run-time (sec)
1	0.9473	5214
2	0.9465	5030
4	0.9427	3929
6	0.9343	2302
7	0.9181	1606
8	0.8905	1094
9	0.8408	592.6
10	0.7607	325.0
11	0.6467	178.1
12	0.0.4828	109.1
13	0.2584	78.03

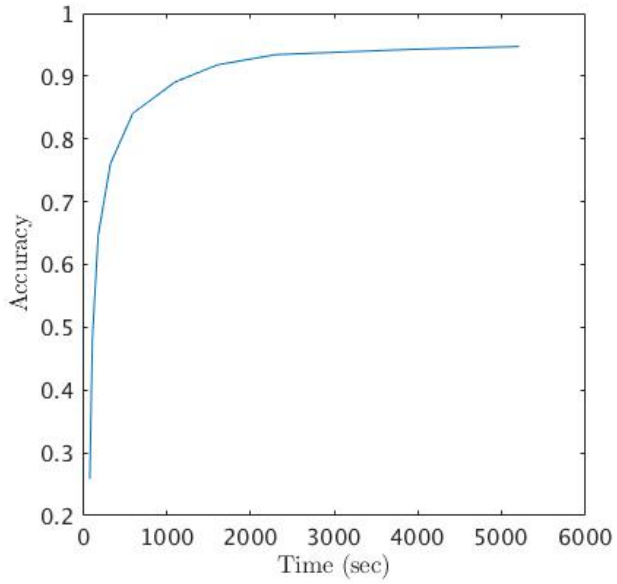

Figure 2: Relationship Between Accuracy and Run-time When Removing Image Border Layers

because removing 14 would completely delete the 28×28 matrix. It is clear that accuracy and run-time have an inverse relationship, but Figure 2 gives more insight into how the relationship may respond to other manipulation methods.

Although we do not evaluate it further here, this figure's resemblance of an L-curve suggests that there are available methods for determining an optimal balance between accuracy and run-time. It also illustrates that there is a relatively large region in which we can obtain high accuracy. We can eliminate 7 layers, meaning 588 features or 75% of the original features, and reduce the run-time by more than 70% while remaining above 90% accuracy.

7 Conclusion

Machine learning is becoming more and more relevant in today's society, so it is useful to understand some of the fundamentals in regards to the machine learning process. We presented an effective method of solving an LP, using the cvx convex modeler and its SeDuMi solver. With the solution, we trained a multiclass classifier to recognize handwritten numbers and used a DDAG to test it as a collection of binary classifiers. After iterating through several values for the weight parameter γ, we settled on $\gamma = 0.00006$, which resulted in an accuracy of 94.75% and a run-time of 5,494 seconds. We then manipulated the training set by removing the borders of the images and found that, by removing 7 layers, we can reduce the run-time by more than 70% while still maintaining an accuracy above 90%. There are several other ways in which the data can be manipulated, so it would be beneficial to explore the results of different manipulations in the future. Perhaps, instead of removing the border of the image evenly, we can take into consideration whether one side has more unnecessary data (white space) than another and reduce the loss of accuracy. We could also test methods of including more data in an effort to improve accuracy while minimizing the increase in run-time. With classification, there is clearly a trade-off between accuracy and efficiency, so it is important to understand the requirements of the problem. Our analysis, however, clearly illustrated the inverse relationship between accuracy and run-time and the importance of balancing the weight of the input parameters.

A Code

A.1 Training

```
close all; clear all;

% Add cvx files to path
addpath /Users/gabrielchapel/Downloads/cvx
addpath /Users/gabrielchapel/Downloads/cvx/structures
addpath /Users/gabrielchapel/Downloads/cvx/lib
addpath /Users/gabrielchapel/Downloads/cvx/functions
addpath /Users/gabrielchapel/Downloads/cvx/commands
addpath /Users/gabrielchapel/Downloads/cvx/builtins

% Load data
load('mnist.mat')

% Establish features
X0 = double(images(labels==0,:));
X1 = double(images(labels==1,:));
X2 = double(images(labels==2,:));
X3 = double(images(labels==3,:));
X4 = double(images(labels==4,:));
X5 = double(images(labels==5,:));
X6 = double(images(labels==6,:));
X7 = double(images(labels==7,:));
X8 = double(images(labels==8,:));
X9 = double(images(labels==9,:));
feat = {X0, X1, X2, X3, X4, X5, X6, X7, X8, X9};

% Set up LP
gamma = 0.00006; % Weighting variable
n = 784;
tic

% Train the data for each classifier
for i = 1:9
for j = i+1:10
```

```
feat1 = feat(i); % Feature of first object
feat1 = [feat1{:,:}];
feat2 = feat(j); % Feature of second object
feat2 = [feat2{:,:}];
m1 = length(feat1); % Length of first feature
m2 = length(feat2); % Length of second feature
cvx_begin
cvx_precision low % Low precision for faster performance
variables a(n) b u(m1) v(m2) % Set up variables for LP
minimize norm(a) + gamma*(ones(1,m1)*u+ones(1,m2)*v) % Linear program
subject to % Constraints
feat1*a - ones(m1,1)*b >= ones(m1,1) - u;
feat2*a - ones(m2,1)*b <= -(ones(m2,1) - v);
u >= 0;
v >= 0;
cvx_end
A{i,j} = a;
B{i,j} = b;
end
end
toc

% save DAG_Chapel A B
```

A.2 Testing

```
% Load parameters incase they are cleared
load('mnist.mat')
load('Dag_Chapel.mat')

% Cell array for storing tested results
solution{1} = []; solution{2} = []; solution{3} = []; solution{4} = [];
solution{5} = []; solution{6} = []; solution{7} = []; solution{8} = [];
solution{9} = []; solution{10} = [];
% Cell array for storing right side of each classification
other = solution;

% Image array for performing DAG
```

```
test_images_test = double(images_test);

% Start with 9 vs. 0 and end with 1 vs. 0
for j = 10:-1:2
for i = 1:j-1
notSmall = find(test_images_test * A{i,j} - B{i,j} < 0); % not i
notLarge = find(test_images_test * A{i,j} - B{i,j} > 0); % not j
other{i} = test_images_test(notLarge,:); % store right side of
%         classification for next iteration
test_images_test = [other{i+1}; test_images_test(notSmall,:)]; % append
%         left side of iteration to right side of previous iteration
end
solution{j-1} = other{i}; % add right side of final classification to next
%     iteration
solution{j} = [solution{j}; test_images_test];
test_images_test = other{1}; % start over with new set = right side of
%     first classification
end

% Find indices of solutions to compare with original
[check0, ia0, ib0] = intersect(images_test, solution{1}, 'rows');
[check1, ia1, ib1] = intersect(images_test, solution{2}, 'rows');
[check2, ia2, ib2] = intersect(images_test, solution{3}, 'rows');
[check3, ia3, ib3] = intersect(images_test, solution{4}, 'rows');
[check4, ia4, ib4] = intersect(images_test, solution{5}, 'rows');
[check5, ia5, ib5] = intersect(images_test, solution{6}, 'rows');
[check6, ia6, ib6] = intersect(images_test, solution{7}, 'rows');
[check7, ia7, ib7] = intersect(images_test, solution{8}, 'rows');
[check8, ia8, ib8] = intersect(images_test, solution{9}, 'rows');
[check9, ia9, ib9] = intersect(images_test, solution{10}, 'rows');

% Make a label array for the results
labelResult(ia0) = 0;
labelResult(ia1) = 1;
labelResult(ia2) = 2;
labelResult(ia3) = 3;
labelResult(ia4) = 4;
labelResult(ia5) = 5;
```

```
labelResult(ia6) = 6;
labelResult(ia7) = 7;
labelResult(ia8) = 8;
labelResult(ia9) = 9;

% Compare results with original
checkResult = labelResult'==labels_test;
badResult = find(checkResult == 0);
accuracy = (length(checkResult) - length(badResult))/length(checkResult);
```

A.3 Play Training

```
% Add cvx files to path
addpath /Users/gabrielchapel/Downloads/cvx
addpath /Users/gabrielchapel/Downloads/cvx/structures
addpath /Users/gabrielchapel/Downloads/cvx/lib
addpath /Users/gabrielchapel/Downloads/cvx/functions
addpath /Users/gabrielchapel/Downloads/cvx/commands
addpath /Users/gabrielchapel/Downloads/cvx/builtins

% Load data
load('mnist.mat')

% Remove borders from images
images_play = images;
removeNum = 7; % Number of layers to remove
rightSide = 784 - (28*removeNum - 1):784; % Right border
top = [];
bottom = [];
for i = 1:28
for j = 1:removeNum
top = [top, 28*i-(28-j)];      % Top border
bottom = [bottom, 28*i-(j-1)]; % Bottom border
end
end
leftSide = 1:28*removeNum; % Left border
remove = unique([leftSide top bottom rightSide]); % Columns to remove
images_play(:, remove) = [];
```

```
% Establish features
X0play = double(images_play(labels==0,:));
X1play = double(images_play(labels==1,:));
X2play = double(images_play(labels==2,:));
X3play = double(images_play(labels==3,:));
X4play = double(images_play(labels==4,:));
X5play = double(images_play(labels==5,:));
X6play = double(images_play(labels==6,:));
X7play = double(images_play(labels==7,:));
X8play = double(images_play(labels==8,:));
X9play = double(images_play(labels==9,:));
featplay = {X0play, X1play, X2play, X3play, X4play, X5play, X6play, X7play, X8play

% Set up LP
gamma = .00006;
n = 784 - length(remove);
tic
% Train the data for each classifier
for i = 1:9
for j = i+1:10
feat1 = featplay(i); % Feature of first object
feat1 = [feat1{:,:}];
feat2 = feat(j); % Feature of second object
feat2 = [feat2{:,:}];
m1 = length(feat1); % Length of first feature
m2 = length(feat2); % Length of second feature
cvx_begin
cvx_precision low % Low precision for faster performance
variables a(n) b u(m1) v(m2) % Set up variables for LP
minimize norm(a) + gamma*(ones(1,m1)*u+ones(1,m2)*v) % Linear program
subject to % Constraints
feat1*a - ones(m1,1)*b >= ones(m1,1) - u;
feat2*a - ones(m2,1)*b <= -(ones(m2,1) - v);
u >= 0;
v >= 0;
cvx_end
A{i,j} = a;
```

```
B{i,j} = b;
end
end
toc

save DAG_Play_Chapel Ap Bp
```

A.4 Play Testing

```
% Load parameters incase they are cleared
load('mnist.mat')
load('Dag_Chapel.mat')

% Remove borders from images
removeNum = 7; % Number of layers to remove
rightSide = 784 - (28*removeNum - 1):784; % Right border
top = [];
bottom = [];
for i = 1:28
for j = 1:removeNum
top = [top, 28*i-(28-j)];      % Top border
bottom = [bottom, 28*i-(j-1)]; % Bottom border
end
end
leftSide = 1:28*removeNum; % Left border
remove = unique([leftSide top bottom rightSide]); % Columns to remove

% Remove border from images_test
test_images_test = double(images_test);
test_images_test(:,remove) = [];
images_test_play = test_images_test;

save('DAG_Play_Chapel.mat', 'images_test_play', '-append')

A = Ap;
B = Bp;

% Cell array for storing tested results
```

```
solution{1} = []; solution{2} = []; solution{3} = []; solution{4} = [];
solution{5} = []; solution{6} = []; solution{7} = []; solution{8} = [];
solution{9} = []; solution{10} = [];
% Cell array for storing right side of each classification
other = solution;

% Start with 9 vs. 0 and end with 1 vs. 0
for j = 10:-1:2
for i = 1:j-1
notSmall = find(test_images_test * A{i,j} - B{i,j} < 0); % not i
notLarge = find(test_images_test * A{i,j} - B{i,j} > 0); % not j
other{i} = test_images_test(notLarge,:); % store right side of
%         classification for next iteration
test_images_test = [other{i+1}; test_images_test(notSmall,:)]; % append
%         left side of iteration to right side of previous iteration
end
solution{j-1} = other{i}; % add right side of final classification to next
%     iteration
solution{j} = [solution{j}; test_images_test];
test_images_test = other{1}; % start over with new set = right side of
%     first classification
end

% Find indices of solutions to compare with original
[check0, ia0, ib0] = intersect(images_test_play, solution{1}, 'rows');
[check1, ia1, ib1] = intersect(images_test_play, solution{2}, 'rows');
[check2, ia2, ib2] = intersect(images_test_play, solution{3}, 'rows');
[check3, ia3, ib3] = intersect(images_test_play, solution{4}, 'rows');
[check4, ia4, ib4] = intersect(images_test_play, solution{5}, 'rows');
[check5, ia5, ib5] = intersect(images_test_play, solution{6}, 'rows');
[check6, ia6, ib6] = intersect(images_test_play, solution{7}, 'rows');
[check7, ia7, ib7] = intersect(images_test_play, solution{8}, 'rows');
[check8, ia8, ib8] = intersect(images_test_play, solution{9}, 'rows');
[check9, ia9, ib9] = intersect(images_test_play, solution{10}, 'rows');

% Make a label array for the results
labelResult(ia0) = 0;
labelResult(ia1) = 1;
```

```
labelResult(ia2) = 2;
labelResult(ia3) = 3;
labelResult(ia4) = 4;
labelResult(ia5) = 5;
labelResult(ia6) = 6;
labelResult(ia7) = 7;
labelResult(ia8) = 8;
labelResult(ia9) = 9;

% Compare results with original
checkResult = labelResult'==labels_test;
badResult = find(checkResult == 0);
accuracy = (length(checkResult) - length(badResult))/length(checkResult);
```

B Figures

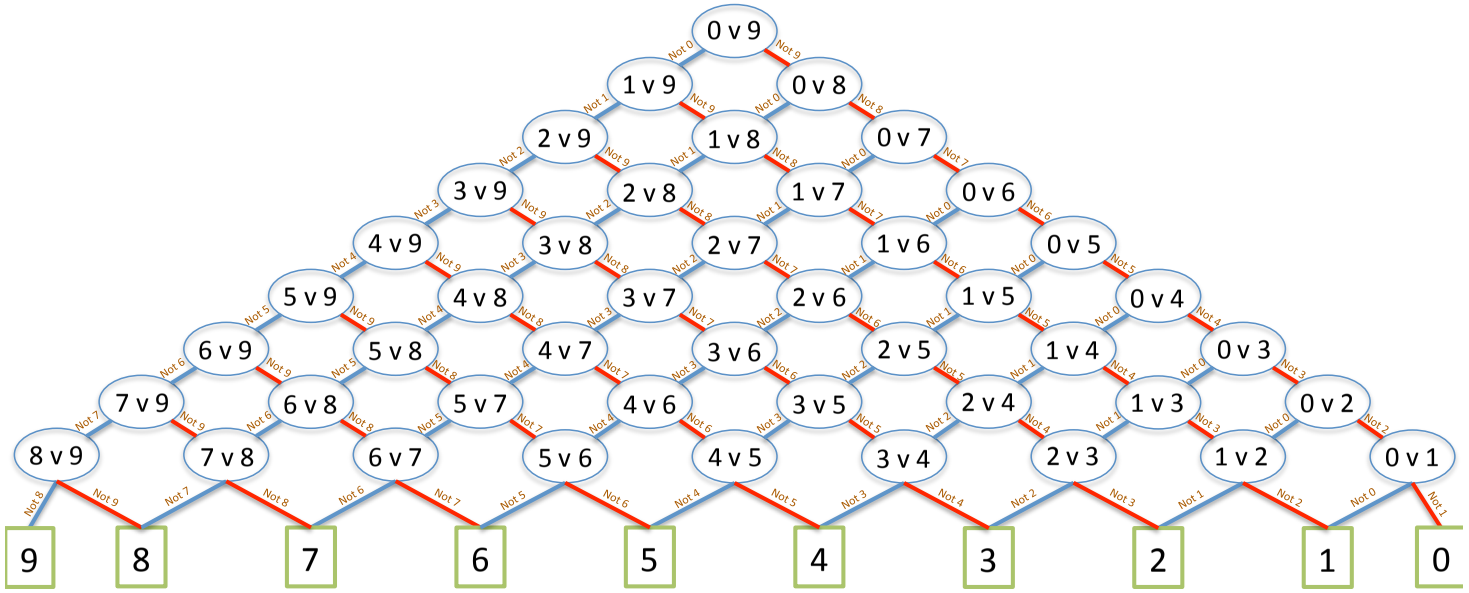

Figure 3: DDAG for Determining the Class of Each Image (blue indicates notSmall inputs and red indicates notLarge inputs)

References

[1] Michael Grant and Stephen Boyd. CVX: Matlab software for disciplined convex programming, version 2.1. `http://cvxr.com/cvx`, March 2014.

[2] Shalom Ruben. Optimal design lecture, May 2018.