

# Image Recognition Using Barcode Analysis

By Liam Rea(100743012) and Philip Jasionowski(100751888)

Prepared for: Shahryar Rahnamayan  
Date : 11/4/2021

# Introduction:

The concept of using AI in a practical sense has always been intriguing. The usefulness of neural networks is evident through many articles and news reports that can be seen everyday. However we don't really get the chance to learn about this stuff in our current classes. So given the relative openness of this project, our team decided to learn about how to make an introductory neural network and use it to compliment our barcode, increasing accuracy substantially.

## Algorithms Explanation:

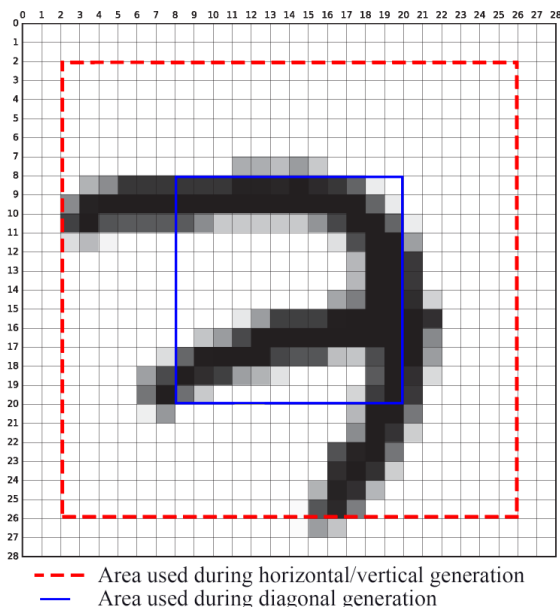
### Barcode generation:

For our barcode generation we used 2 for loops adding up the total for each row before moving on to the next. If the total for each pixel divided by the amount of pixels lands over our threshold a 1 is appended to the barcode array. The same is then done by cycling through the everything in the column before moving to the next one.

This gives a generated barcode for the 0° and 90° projections. When thinking about optimizations and glancing at the mnist photos, we realized that a majority had no data for the outside 2 pixels (the frame if you will) we decided to test what the average value was for a pixel on the frame to see if we could ignore those rows and columns and make the barcodes smaller. We wrote a quick script called *edgeChecker.py* which computes the average value of a pixel in this outside frame, it was found to be 0.28649. Since the value is so small it would almost never impact the barcode generation so we opted to skip it.

```
# horizontal barcode
for x in range(2, 26):
    temp = 0
    for y in range(2, 26):
        temp += image[x, y]
    if (temp / 24) < 48:
        barcode.append(0)
    else:
        barcode.append(1)
```

Sample Code that generates a horizontal barcode



For the 135° projection we wanted to keep the barcode as lightweight as possible so based on another similar script we found that the majority of pixels appear in the center (as to be expected) as such we decided to ignore the first and last 8 rows or columns of the image to cut down the length of the barcode generated from the diagonal (51 digits cut down to just 25). These “cuts” we made to the images can be scaled up or down based on the source data but we found that this amount of optimization yielded very accurate results whilst cutting down on storage and time complexity. Assuming we used 4 projections (horizontal, vertical and 2 diagonal projections) and generated

a digit for each one, each barcode would be 154 digits long, our cut down code generates barcodes only 73 digits long (52.6% smaller). This enables us to in theory store twice as many barcodes in any given database.

The time complexity of the barcode generation is  $O(n^2)$  as each segment of the barcode generator has a nested loop of sorts running through the columns and then the rows

### Search algorithm:

After being given The barcode generated in the previous step the search algorithm is used to trim and refine our pool of candidate images. So for each barcode in the

```
def ham(in1, in2):
    diff = 0
    for x in range(0, len(in1)):
        if in1[x] != in2[x]:
            diff+=1
    return diff
```

search

dataset, we

run a hamming distance calculation. This algorithm checks the difference between the two barcodes, the one generated from the original image, and the one in the search dataset currently in use. The algorithm totals the differences and returns it. If the total difference is less than 5, this means the

images are similar, and it should remain a candidate for the final output. As such, we save its position in the dataset in an array and continue to look at each barcode.

After we have scanned through each barcode, what will be remaining is a list of locations of the most similar images. In practice, this usually reduces the number of potential images by 90% - 98%. This should not be confused with an accuracy of 98% as this is merely a reduction in the number of candidates. Given the size of our dataset, 10 000 images, there is a very high chance that some images are not the same as the one we wanted to search for. This is where our neural network comes into play.

Different neural networks work in different ways, but generally they function under the same principle. Nodes are created by the network and are fed data. A function calculates if the weight of the incoming data is enough to trigger the function, and as such will either fire or not fire. The firings are passed on through the network and eventually arrive at a conclusion. In this case it outputs a number from 1 to 10.

```
# vertical barcode
# n^(6n)
# O(n^2)
for y in range(2, 26):      # n
    temp = 0
    for x in range(2, 26):  # n
        temp += image[x, y] # n
    if (temp / 24) < 48:     # n
        barcode.append(0)   # n
    else:                   # n
        barcode.append(1)   # n
```

Time complexity of vertical barcode that is generated

```

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(128, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(128, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(10, activation=tf.nn.softmax))

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

```

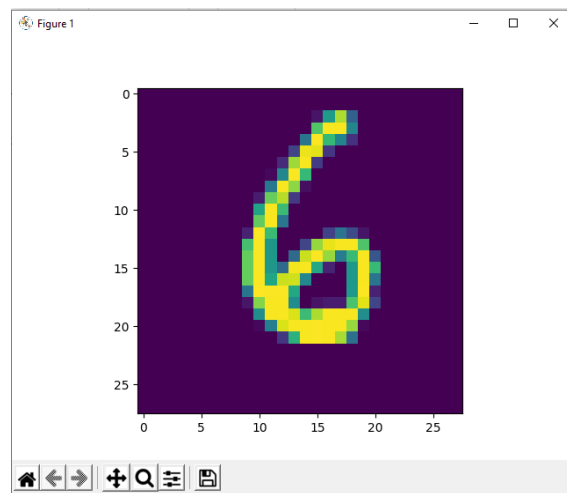
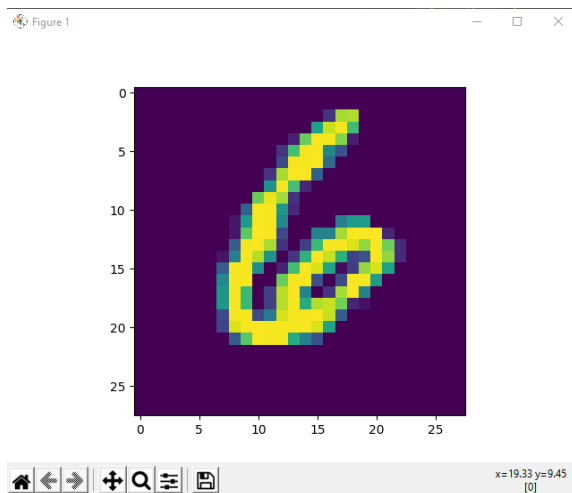
Our network consists of 2 layers, each with 128 nodes and ends in 10 nodes, one for each layer. By feeding it images we can train the AI to recognize images and what they are. However this requires a very large dataset to work with. Luckily, the MNIST dataset contains 60000 training images with labels which we can use to train our network. After training, the Network is able to make predictions with an accuracy of 94%.

```

Epoch 1/3
1875/1875 [=====] - 2s 972us/step - loss: 4.1995 - accuracy: 0.8200
Epoch 2/3
1875/1875 [=====] - 2s 947us/step - loss: 0.3534 - accuracy: 0.9237
Epoch 3/3
1875/1875 [=====] - 2s 979us/step - loss: 0.2421 - accuracy: 0.9406

```

Using this network, we can examine our trimmed pictures using the AI to more accurately identify related images. From our trimmed images, we examine each one and when two images are identified as being the same, it prints them out



Comparison between the original image(left) and the searched for image(right)

## Analysis:

After running 100 tests, our program came back with an accuracy of 96%, however this can go even higher if we allow it to find every match within its search. As of right now, the algorithm only returns the first result found in the set of potential matches. If we allow it to find all matches it will be a few percentage points higher than its current accuracy.

This does come at a cost however and that cost is speed. While this program is extremely accurate, it is not the fastest. Most of the searching is done in  $O(n)$  time. While we can sort using an algorithm such as quick sort, or merge sort, the problem arises when making a comparison.

For example let's take the string 10001000 and 10010000. The hamming distance between them 2, making them very similar, however when sorted by an algorithm, there are 8 numbers that can fit between them, all of which have larger hamming distances when compared to the first number and are therefore less accurate. This problem is exponentially magnified when we start dealing with larger numbers, and as such is an unreliable way to find similar images.

```
for x in range(0, len(imgtest2)):
    dis = ham(test, imgtest[x])
    if dis <= 4:
        final.append(x)
```

```
def ham(in1, in2):
    diff = 0
    for x in range(0, len(in1)):
        if in1[x] != in2[x]:
            diff += 1
    return diff
```

The first part of our code relies on these two pieces, the loop and the method. The loop is to loop through the test dataset, which gives us a complexity of

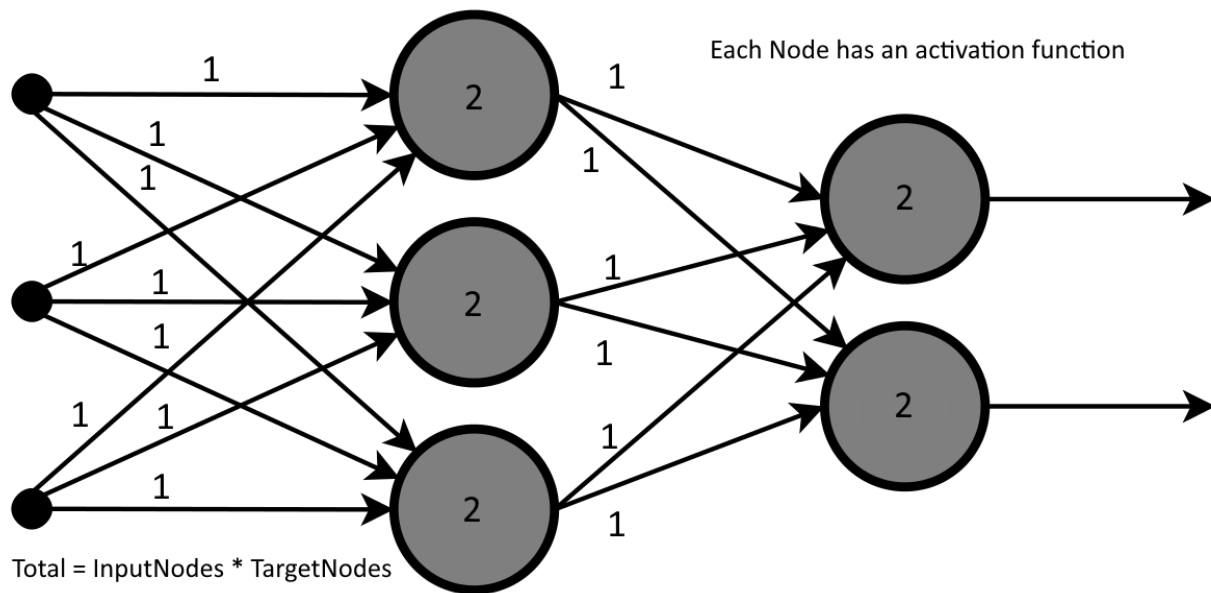
$$\text{Alg1} = (n(1+1+(\text{alg2}))+1+1) = 2n^2+6n+2$$

$$\text{Alg2} = 1+(n(1+1)+1+1)+1 = 2n+4$$

$$\text{Complexity} = O(n^2)$$

So to get the hamming distance for our barcodes, it runs in  $O(n^2)$  time. Afterwards, to search we look through the list of candidates, we perform another linear search through our candidates. This list could have some optimizations, such as storing more similar items to be looked at first rather than looking indiscriminately.

Now for the AI. We can calculate the amount of calculations that the neural net performs using the number of nodes that are receiving input. So for the first layer we have 128 input nodes, all connected to the next layer via weights. These weights are a function which multiplies the saved weight by 1, the output of the previous function. The number of multiplications equals the number of neurons of the previous layer multiplied by the number of neurons for the next layer. In our case that's  $128 \times 128$  or 16384 multiplications.

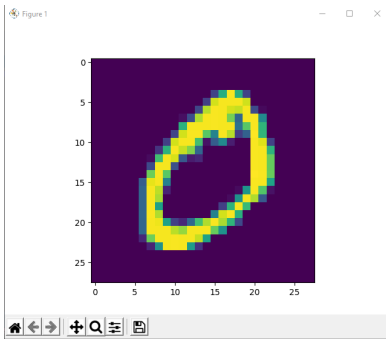
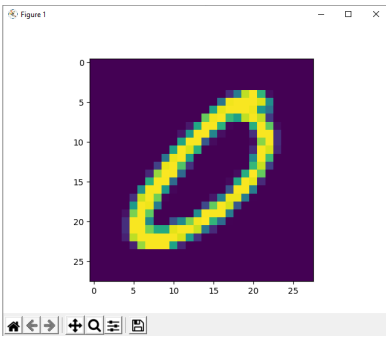
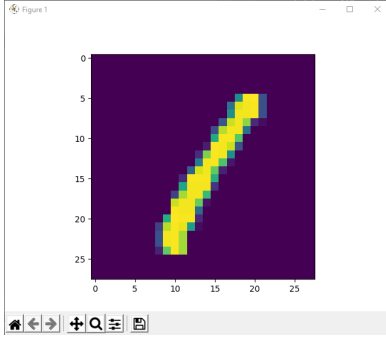
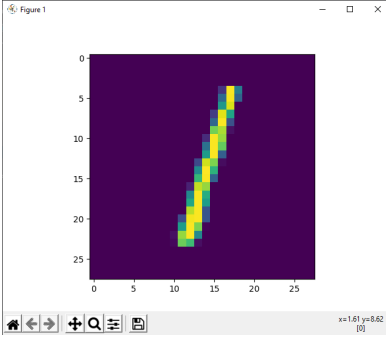
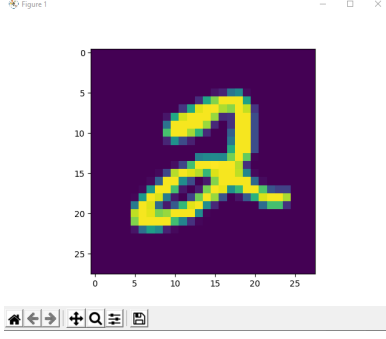
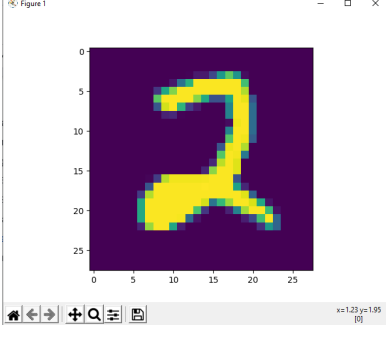
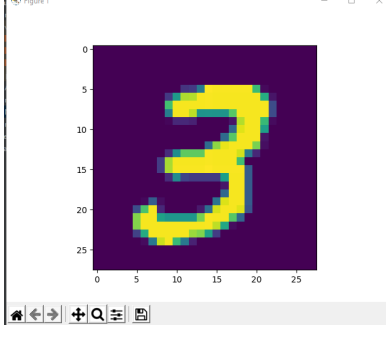
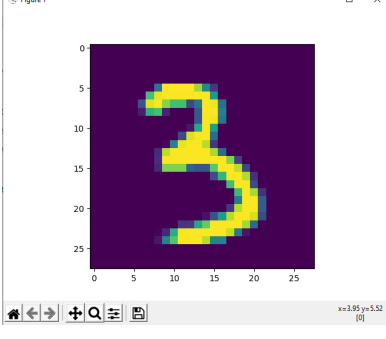


The next layer does the same thing. 128 neurons to 10 gives us 1280 multiplications. Additionally, the activation function for each neuron runs in  $1+1$  time, so an additional  $512+1$  operations will be performed.

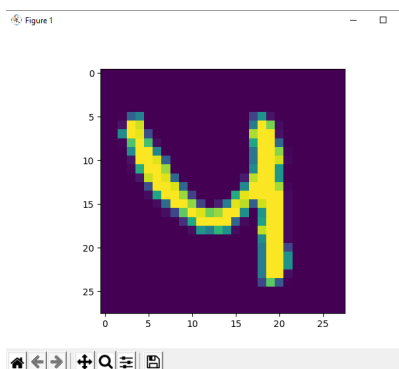
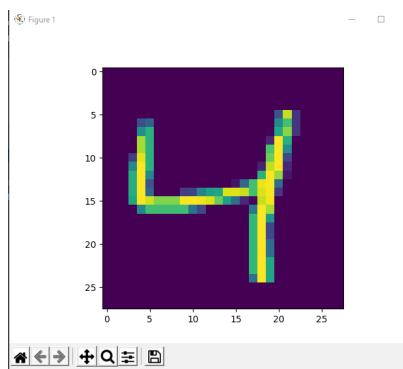
In total  $16384(n)+513$  operations will be performed by the neural network where  $n$  is the size of the image. So for small images like these, the network is not very efficient. However, for larger images, the complexity remains at  $O(n)$  time, meaning that it will scale decently with larger images.

Now, this network must loop through each item in the results list, which takes  $O(n)$  time. So the search in total becomes  $O(n^2)$ . Not very efficient when it comes to larger datasets, the trade off for this being accuracy. While the program doesn't scale very well, it is highly accurate to the point where images that it confuses tend to confuse observers as well.

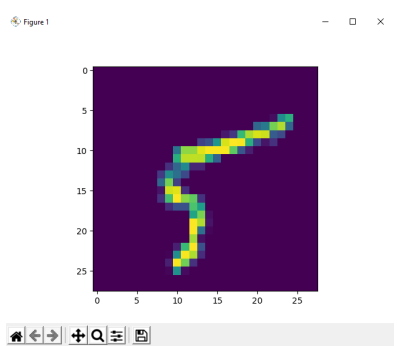
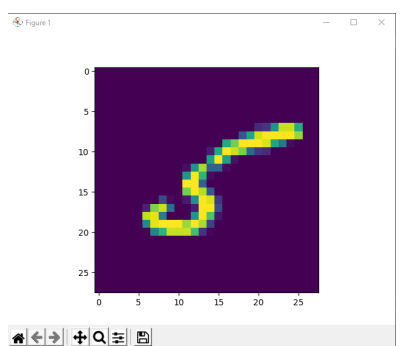
# Examples: First return on digits.

	Searched Image	First Returned Image
0		
1		
2		
3		

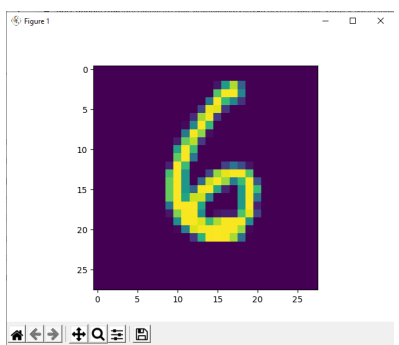
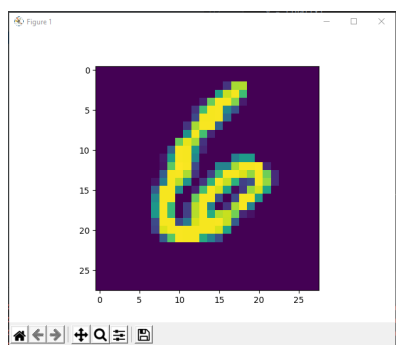
4



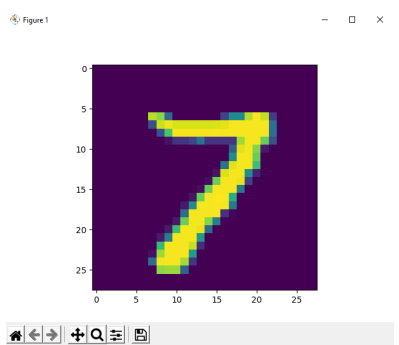
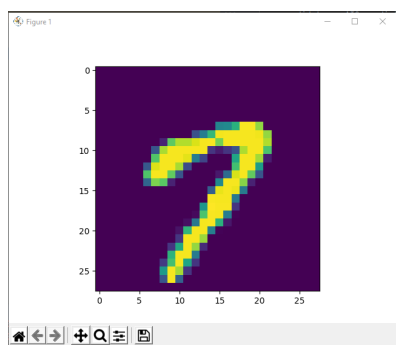
5



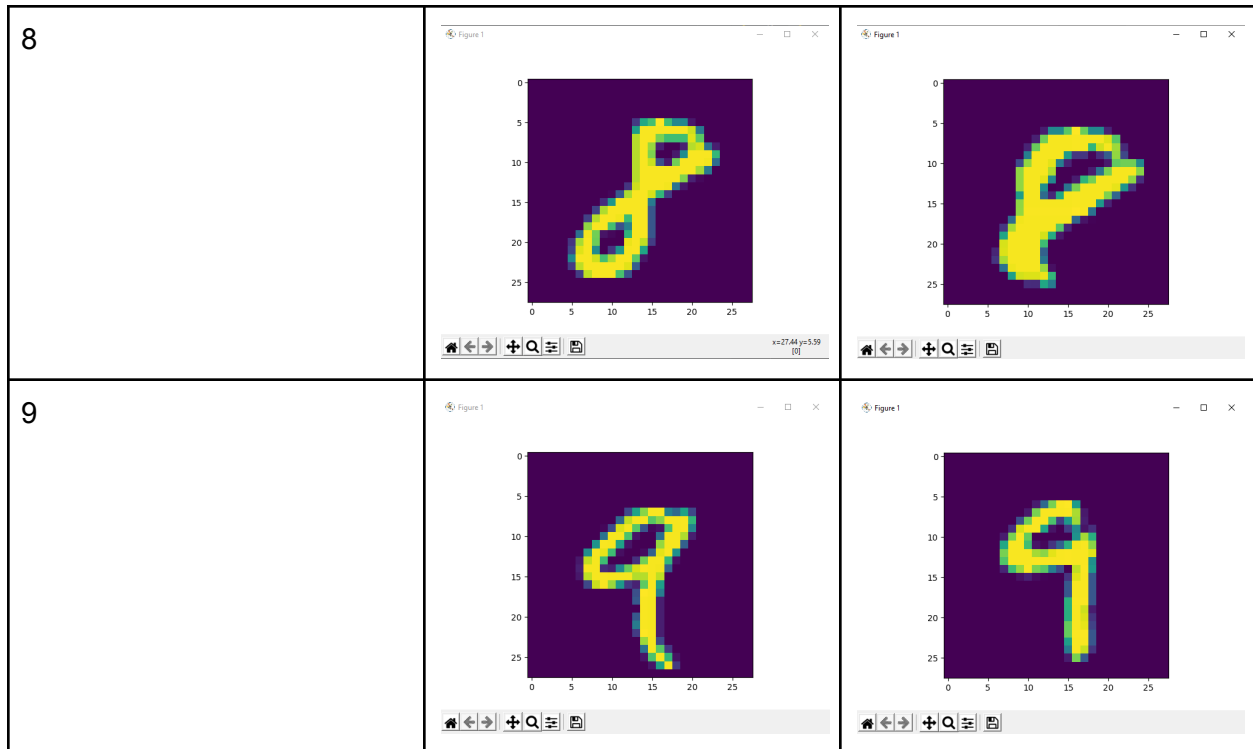
6



7







## Conclusion

Overall, we have successfully completed the problem objective. We managed to use AI to complement our barcode system to take our algorithm to an extremely high accuracy. While this does come at a cost of a  $O(n^2)$  compute time, we believe the 96% accuracy is worth the extra compute time. This assignment has exposed us to problems and solutions that can be extremely useful in our future, such as hamming distance and basic neural networks. This project was very interesting and useful to our development as software engineers and we look forward to doing more projects like it.