Handwritten Digit Bitmap Compression Using Genetic Algorithm and Perceptron Networks

Ethan Zhou, Jake Adicoff, and Mac Groves

Abstract: In this paper, we analyze the effectiveness of a Genetic Algorithm (GA) on improving the performance on Neural Networks (NN) in optical character recognition. We use a handwritten digit dataset, and use a Genetic Algorithm to compress the 32x32 bitmaps into a much smaller series of integers. Within our Genetic Algorithm, individuals represent compression schemes, and we use a single-layer feed-forward Neural Network (a *perceptron*) to train and test on that scheme. The network's final, trained accuracy is used as the fitness for the Genetic Algorithm. We find measured success in our goal. Notably, we determine that we can find compression schemes using 64 (and even 32) symbols that are better than the commonly used 8x8 integer maps, thus outperforming that naive form of compression.

1. Introduction

Genetic Algorithms are a nature-inspired optimization method that operate on symbol strings to find the highest scoring string based on a fitness function. The symbol strings encode possible solutions to a problem. Typically, a population of symbol strings is initially randomly generated. Then the population is *evolved* by breeding individuals together, which creates a new population called the next *generation*. By strategically preferring high-fitness individuals from the population for breeding, the genetic algorithm gradually finds better solutions over many generations, and eventually converges on the optimal symbol string.

The fitness function for our GA is a perceptron neural network. A perceptron network is a function approximation algorithm inspired by the function of neurons. A perceptron receives inputs through an *input node layer*, which is fully connected to an *output layer*. The output node(s) produce output based on the input node values and the *edge weights* connecting the input nodes to the output nodes. The output of a perceptron is its "best guess" of the output of the function it is approximating.

In order to encode how to compress the large 32x32 bitmap into a smaller number of NN inputs, We take *n* symbols and 'color' the 32x32 bitmap with these symbols, dividing the bitmap into *n* distinct regions. Using this colored map, we are able to make an integer vector of length *n* where each component of the vector corresponds to the sum of bits in the corresponding region. An individual in the genetic algorithm is region definition of the 32x32 bitmap, which is used to compress a set of bitmaps and use the perceptron network to evaluate how effective the compression scheme is, looking for high accuracy in digit recognition.

In section 2, we explain the optical digit recognition problem and the 32x32 bitmaps that we use for testing. In sections 3 and 4,we discuss the details of genetic algorithms and perceptron networks. In section 5, we describe our method of compressing bitmaps and how we use the genetic algorithm and perceptron network to find an effective compression scheme. In section 6 we outline the experiments we have done to explore our hybrid compression method, and in section 7 we present and discuss our findings. In section 8 we consider possibilities for further research. In section 9, we conclude with a summary of our findings and some closing remarks.

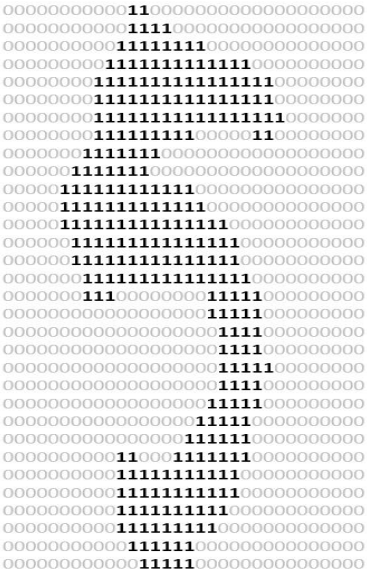

Figure 1: A sample of a 32x32 bitmap of a handwritten '5'. The 0's have been desaturated and the 1's have been put in boldface to emphasize the relevant information.

2. Optical Digit Recognition

The Optical Digit Recognition (ODR) problem is subset of the Optical Character Recognition (OCR) problem. OCR is concerned with identifying characters in images of handwritten or printed characters. While this may seem like a trivial problem for humans, OCR is a difficult problem for machines to solve. A major hurdle is that any given character can appear very different from two different sources. Among printed sources, different fonts can display a lot of variance (e.g. "a" versus "ɑ"), and this problem is even more significant for handwritten characters. Furthermore, OCR typically processes scans or photographs of words which can introduce artefacts to the data, like pixelation, blurs, etc. These complications make it difficult and unfeasible to hard-code a set of rules to identify every character accurately. Instead, a Neural Network is more well-suited for the OCR problem, since they can learn to identify patterns to accurately classify a large variety of input data into categories.

The 32x32 bitmap is illustrated in figure 1 above. This format essentially divides an image of a character up into a 32x32 grid, and assigns a 1 to every cell that has "ink", and 0 to empty cells. The corresponding target value is, of course, the value of the digits written in these images. Another format that may be used in the digit recognition problem is the 8x8 integer map. Although we do not use it, we benchmark our results against it, so it is worthwhile to describe this format. The 8x8 map is a compressed version of the original 32x32 bitmap. These 8x8 maps can be visualized as a lower-resolution grid that is placed directly over the higher resolution 32x32 bitmap, such that each cell in our 8x8 grid overlays a 4x4 sector in the 32x32 bitmap. Each cell in this 8x8 map contains the sum of these 16 bits (resulting in any value between 0 and 15). The target of these compressed character data is the same value as the uncompressed digit.

3. Genetic Algorithms

Genetic Algorithms (GA) are inspired by the effects of natural selection and mutation, found in genetics. A *population* is made up of *individuals*, which are strings that represent a candidate solution in the search space. The individuals are capable of *breeding* with each other and mutating. Each individual has a different probability of entering the breeding pool, based on how fit the candidate solution is. Fitness is evaluated with a *fitness function*, which is hugely important in the operation of a Genetic Algorithm. The better the fitness function, the higher the quality of the final solution. In the breeding pool, two individuals are used to produce children that have traits (characters from the strings) from both parents. These children are also subject to small random mutations that induce exploration of the solution space. The children thus produced constitute the next generation of the population, and this process is repeated to ideally converge on the optimal individual in the solution space. The details of the GA are best described in phases.

Initial population generation: A population of a user-specified size (usually 50-100 individuals) is generated randomly. The strings should be designed to be constrained in such a way that each string is a valid representation of a possible solution to the problem. For compression, an individual is simply an array of integers variables which are (initially) assigned a random number between 1 and *n*, where *n* is the number of symbols in the compression. The randomness allows for breadth in the exploration of a solution space.

Evaluation: Here, individuals are tested with the fitness function to determine the strength of each individual, or solution, in the population. The fitness of an individual allows the selection

phase (described next) to preferentially pick more fit individuals to be bred. Therefore, the fitness function defines the topography of the search space, and it is important that the fitness function properly reflects the problem. For compression, the fitness function uses trains a neural net on a set of compressed vectors. The fitness is then the accuracy of the net as it test of a separate set of similarly compressed vectors.

Selection: Individuals are chosen to enter the breeding pool based on some heuristic. We have implemented three heuristics in our code: Ranked Selection (RS), Tournament Selection (TS), and Boltzmann Selection (BS).

In RS, for a population of *n* individuals, the individuals are sorted in an array by fitness from least fit to most fit. Then, each individual is selected to enter the breeding pool with probability

$$\frac{i}{\sum_{j=1}^{n} j}$$

Where *i* is the rank of the individual's fitness (*i* can be thought of the index of the individual in a array sorted low to high by fitness). The denominator simply normalizes the probabilities. Individuals can be selected more than once to enter the breeding pool, so this type of selection allows more fit individuals to preferentially enter the breeding pool.

In TS, two individuals are chosen at random from the population and the more fit of the two is selected to enter the breeding pool. This type of selection has a significant drawbacks - low fitness individuals may be selected if two individuals of very low fitness enter a tournament

In the final selection heuristic, BS, individuals are chosen with probability

$$\frac{e^{f_i}}{\sum_{i=1}^{n} e^{f_i}}$$

Where f_i is the fitness of individual i in the sorted array. The denominator again serves to normalize the probabilities. This type of selection, like RS, allows fitter individuals to have a higher probability of entering the breeding pool. However, if there is a large variance in the fitnesses among the population, BS will favor individuals with disproportionately high fitness considerably more than RS.

Breeding: The selected individuals in the breeding pool are used to generate the next generation of individuals. All individuals in the pool are paired up with others in the pool, and each pair generates children to create the next generation. The children are generated via *crossover* and then undergo *mutation*. The algorithm may also automatically promote the most fit individual from the pool to the next generation without any change. This is called *elitism*.

Crossover: The child string's characters come directly from the parents, i.e. the character in position *i* in the child string is the same as character in position *i* of one of the parent strings. A *crossover probability* parameter is provided by the user to set the chance of crossover occurring. If crossover is not performed, then one or both of the parents enter the next generation unaltered. There are many crossover methods; the methods covered in this study are 1-point crossover and uniform crossover.

In 1-point crossover, an index *i* in the symbol string is chosen randomly. The child is generated by combining the characters from indices 0 to *i* from one parent with the characters from indices *i + 1* to the end of the other parent. N-point crossover selects N indices randomly,

which defines N+1 segments of the string. The child string is created by concatenating alternating segments from each parent. Uniform crossover also combines two parent individuals to form one child, but it traverses each character in both parent strings and selects one to pass to the child with equal probability.

Depending on how the breeding pool is constructed, crossover of two parents may generate two offspring, so that the next generation is the same size as the breeding pool. Otherwise, the breeding pool needs to be twice as large as the original population size to maintain equal population size per generation.

Mutation: Each symbol in the child individual has a random chance to be changed to a different value. The character mutation probability is usually low (~0.01) so that good symbol sequences aren't often destroyed by mutation. Mutation enables the Genetic Algorithm to explore new symbol assignments that may not exist in the original population, since crossover can only generate children with characters that already exist in the population. A low mutation probability will preference exploitation while a high mutation probability will preference exploration. The solution space in this specific problem is extremely large with many local maxima. For this reason, we implement a method in mutation that increases the mutation rate by a factor of 8 if the best solution has not changed over 5 consecutive generations, and then decreasing it by ½ per generation until it returns to the original rate. We are able to tell if we have a streak by using elitism.

Elitism: In order to not ever lose our best compression scheme in any single generation, we use elitism. In elitism, we first save the best individual from a generation immediately after testing fitness of all individuals. After selection, crossover, and mutation, that best individual is

reinserted in the population and takes the place of the individual with the lowest fitness. This means that as our genetic algorithm runs, the best solution in any generation is guaranteed to be the same or better than in all generations.

After the breeding phase is completed, the Genetic Algorithm has a new population of individuals. The algorithm can then either continue the process of evaluation, selection, and breeding to continue to search for better solutions, or terminate and return the most fit individual it has found across all generations.

4. Perceptron Neural Networks

Neural Networks are loosely inspired by the behavior of animal brains. Brains contain a large collection of interconnected neurons, each of which can interact with the neurons it's connected to. Neurons can both receive and send signals to other neurons. Neurons have an activation threshold that is a function of the their input signals, and when the threshold is met, the neuron will produce a signal of its own to send to downstream neurons. Neural networks mimic this behavior, and replaces the activation threshold with an *activation function*, which will be explained later. By adjusting the strengths of the connections between the artificial neurons in the network, the algorithm can "learn" to identify patterns from inputs to make classifications.

The perceptron consists of an *input node layer* that is connected to an *output node layer*. The two node layers are fully connected, i.e. there exists an *edge* between every input node and every output node. The two perceptron architectures we test will have a different number of nodes in the output layer: one with just 1 output node, and one with 10 output nodes. Each node can output a value in the range [0,1] inclusive. In 1-node version, the final output is interpreted

by multiplying the output node value by 10 and rounding down to the nearest integer. In the 10-node version, each node represents a digit, and the final output is the digit associated with the node with the largest value. We will just discuss the one output node model in detail, and extend the concept to the 10 node model.

The input of a perceptron is a vector of numbers. For our ODR problem, the input layer vector is a flattened (1-dimensional) version of the digits' bitmap. The j^{th} input is denoted as I_j. Every edge in the network (connecting I_j to the output node) has a weight W_j. These weights are initially set to a random value between -1 and 1. The output node takes the dot product of the input vector with the weight vector, and feeds that value to an activation function to produce some output. The activation function we use is a sigmoid function:

$$\vec{I} \cdot \vec{W} = x; g(x) = \frac{1}{1 + e^{0.5-x}}$$

We refer to the output of the net *g(x)* as *O*. The goal of a neural net is to set the values in the weights vector *W* such that given an arbitrary input vector *I* with a target value *T*, *O* is as close to *T* as possible. We also include an extra 'bias node' as part of the input vector. The value of this always 1, but its edge weight is updated along with all other edges during the training process. This bias node essentially allows the neural net to "tune" the slope of the output node's activation function, yielding better results.

The process of setting the values of the weight vectors is called *training*. A Neural Network is trained on a large set of inputs with known target values *T*. These inputs are first fed into the untrained net, which will not produce good output values *O*. However, this error *T-O* can

be used to update the network's edge weights, so that future similar inputs may have a smaller error. Each edge weight W_j connected to the output node is updated as follows:

$$W_j = W_j + (\alpha \times I_j \times Err \times g'(I \cdot W))$$

In this function, α is some learning rate, usually set between 0 and 1, and *Err* is equal to *T-O*. With each target/output pair (*T,O)*, there is an associated mean squared error surface. The above edge weight update equation is equivalent to a negative gradient descent on this error surface, so that error is minimized. One total iteration of this weight update process constitutes an iteration of training, or an *epoch.* To achieve the best results, a neural net is trained over many epochs, but not too many so that the neural net *overfits* the problem. Overfitting occurs when the neural net outputs values that reflect the training set well, but fails to generalize to new inputs.

If there are multiple output nodes, $\vec{I} \bullet \vec{W}$ is calculated for each output node, where each output node has its own weight vector. The update is repeated for each output node, and the target value *T* may be different for each node. Thus, each output node will respond differently to the same inputs, and the final output is encoded in some way by the state of each output node.

After training, we are able to evaluate the neural net using a new set of test inputs that is different than the training set. Input vectors are fed into the neural net, and we compare *T* and *O* values. Edge weights are not updated in this phase. We measure error as:

$$E = \frac{n - \sum_{k=0}^{n} a}{n}; \text{where } a \text{ is 1 } \textit{iff } T{=}O, \text{ 0 otherwise}$$

Where *n* is the number of test inputs. The more times the neural net's output matches the target output, the lower the error is.

5. Compression

The basic concept of our work is to compress the 32x32 (1024) bit images into smaller sets of integer inputs for the perceptron, similar to how the 8x8 integer maps compressed the 32x32 bitmap into a smaller 8x8 integer array. Fewer perceptron inputs allows for faster perceptron run time, which is important in high-volume applications, such as mail sorting or digitizing scans. However, naively dividing up the full size bitmap into 64 equal sectors and summing the values in each sector is too simple, and loses too much information in the compression process. Intuitively, not all regions in the full sized bitmap carry the same amount of information. The edges are usually blank, and the centers carry more important data. Treating all areas of the bitmap equally is unlikely to be the optimal compression method.

Instead, a less lossy compression method may involve defining regions of different sizes, covering different areas of the bitmap based on information density. Perhaps all of the bits along the edges could be separated and summed into just a few sectors, while the bits near the center may be divided and summed on a more granular scale. See Figure 1 for an illustration.

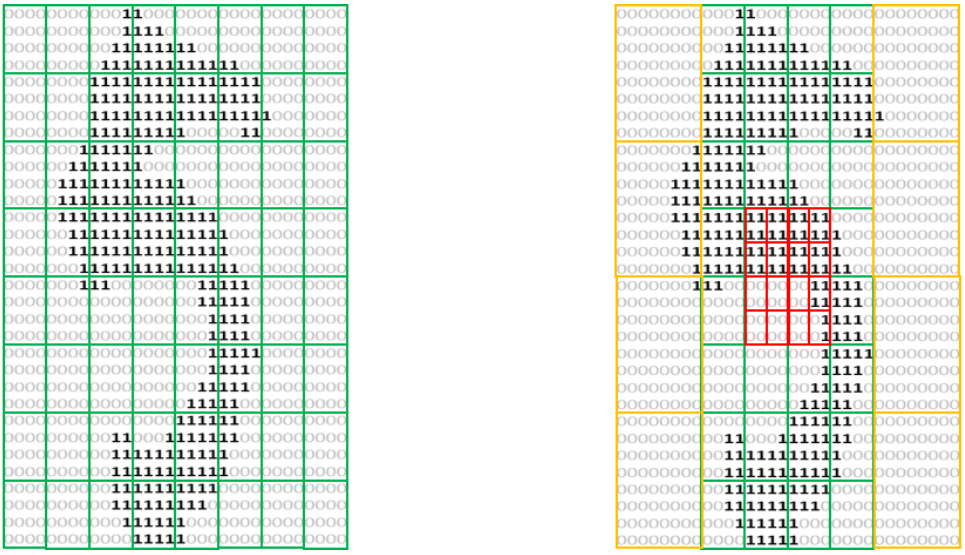

Figure 1: Illustration of the naive 8x8 uniform compression method (left, used in Project 4) and a possible variable size compression region (right). The bits in each region are summed and passed

as inputs to a perceptron neural network. The non-uniform compression region method may provide better neural net accuracy results, as the information density is not uniform across the bitmap. The optimal region division does not even have to consist of contiguous regions; each region may be an arbitrarily complex collection of bits.

In order to generate these compression regions, we plan on using a Genetic Algorithm. The individuals in our GA will be a string of 1024 characters, with each character being one of a fixed number of symbols. For the purposes of this explanation, we will use 20 symbols: the characters ‘a’ through ‘t’. (In our experiments, we test a variable number of symbols, and use integers as our symbols.) The positions of every ‘a’ character defines region ‘a’, and so on for each of the symbols. Each individual’s region map is used to compress the training set of 32x32 image bitmaps into a list of 20 values: all of the bits in region ‘a’ are summed to produce a value for node ‘a’, and so on. These 20 node values are then fed into a perceptron to train for about ~10 epochs. (We found that this was adequate for the perceptron in project 4.) The trained perceptron digit recognition accuracy is then used as the GA individual’s fitness. This process is repeated for each individual, which completes one fitness evaluation cycle for the Genetic Algorithm. The rest of the GA process (selection, crossover, mutation, etc…) is then executed, and the a new iteration will begin. Ideally, the GA will produce high fitness individuals (and therefore high perceptron accuracy) after many generations.

The final product will be the GA’s compression map, which can be used to pre-process handwriting data for neural network evaluation. This method should be faster than the full 32x32 bitmap evaluation because fewer input nodes mean that fewer calculations will be required for the perceptron to make its classification. Ideally, our perceptron would have a comparable character recognition accuracy than the 32x32 map, despite the from compression.

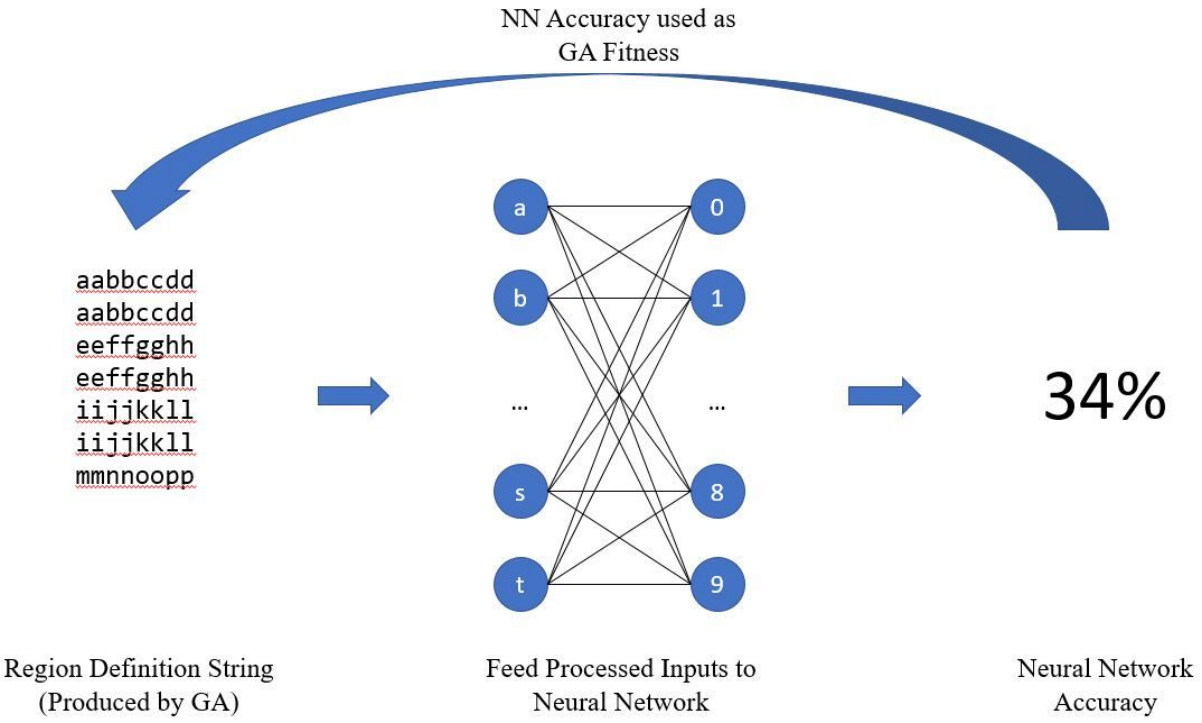

Figure 2: A flowchart outlining how the GA and Perceptron interact. The GA individuals determine how the 1024 bit image is compressed into 20 perceptron input nodes, and the trained perceptron accuracy is the individual's fitness. (The region definition string in this figure is an example of an uniform division.)

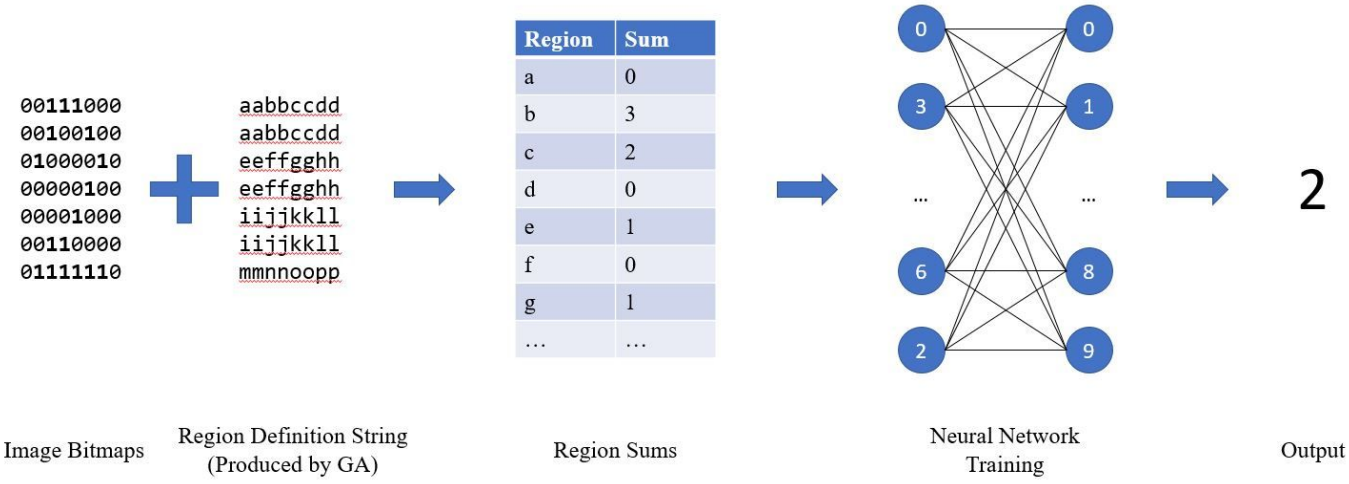

Figure 3: A flowchart detailing how the perceptron inputs are generated. Every single bitmap in the training set is preprocessed using the GA's compression region map, summing all the bits in each region. The perceptron then learns on this compressed input set. By later GA iterations, the perceptron would ideally have a high character recognition accuracy with very few inputs.

6. Experimental Methodology

Our primary goal was to see how few symbols we could use while still obtaining high OCR accuracy with our perceptron network. We benchmarked our results against the 8x8 compression method used in Project 4, which resulted in an about 66% recognition accuracy with good learning parameters. We used a NN **learning rate of 0.01** and **10 output nodes** for all of our experiments, as that was the best parameter combination that we tested in Project 4. We used the results from Project 1 to guide our GA parameters, which were: **rank selection, 0.8 crossover probability,** and **0.01 mutation probability**.

Prior to seeing how few symbols could produce a high perceptron accuracy, it was important to choose between the uniform crossover and N-point crossover strategies. Uniform crossover could potentially introduce more exploration since the individuals are more radically different per generation, but if the optimal compression maps are highly organized into contiguous regions, uniform crossover could destroy these highly organized structures. On the other hand, N-point crossover would preserve more structure from generation to generation, provided *N* is significantly less than 32x32. In order to test which strategy is better suited for our problem, we ran tests with **8, 16, 32,** and **64 symbols**, and comparing the results of using **uniform crossover versus N-point crossover**. Preliminary testing suggested that different values of N (in the range from 1 to 100) did not have a significant effect, so **20 point crossover** was selected for this comparison test. **Four trials** of **100 individuals** and **250 generations** were used for all tests, and the final fitness results were averaged over these four trials.

Using the results from these tests, we were able to identify a good candidate of number of symbols for more in-depth GA search, which would need to strike a balance between having a

relatively high perceptron accuracy and a low number of symbols. If a symbol set had a particularly low accuracy, it is unlikely that more individuals and generations would result in significant improvements. On the other hand, while having more symbols necessarily increases the theoretical accuracy ceiling, more symbols results in more perceptron inputs, which is detrimental because of the increased computation time per character. With these considerations, we decided to run a more thorough GA search with **200 individuals** and **375 generations** using **32 symbols**. We decided to increase individuals more significantly than generations, because we noticed that the fitness scores generally plateaued in the later generations, so having more individuals would likely be more beneficial than more generations.

We also did some quick testing to test if a different mutation boost factor (See Section 3, Mutation subsection) to escape local maxima was beneficial. The tests quickly revealed that factors larger and smaller than 8 were ineffective at moving individuals away from local maxima.

7. Results and Discussion

In the first part of this study, the performance of uniform crossover was compared to 20-point crossover. Both crossover types had nearly identical runtimes. There was not a significant change in performance between 20-point crossover and uniform crossover. Uniform crossover gave better results for 8 symbols and 32 symbols, while 20-point crossover gave better results for 16 symbols and 64 symbols. The variability of the algorithm obscures any significant differences in performance between the two crossover types.

The evolution trends for the two crossover types were similar (Figures 4, 6). Individuals with 8 symbols achieved just over 40%, but the fitness was highly variable. Several trials failed to produced an 8-symbol individual with a fitness above 30% after 200 generations. This variability is likely due to the enormous size of the search space. The fitness of the population increased throughout the evolution and very little plateauing was observed.

Individuals with 16 symbols were able to achieve up to 80% fitness, but the results were similarly variable. Evolution with 20-point crossover tended to plateau after 50 generations, while evolution with uniform crossover tended to see improvement until the final generation. We hypothesize that the different behavior is due to the increased exploration of the uniform crossover that enables the algorithm to climb out of local minima. The fact that individuals with 16 symbols are able to achieve 80% fitness is significant because the 8x8 compression method with 64 input nodes implemented in the previous project was unable to achieve over 70% accuracy with the perceptron. Therefore, with one fourth the number of inputs, the 16-symbol compression method was able to reach significantly better perceptron fitness than the 8x8 compression.

Doubling the number of symbols to 32 resulted in a substantial increase in the upper bound of the fitness. All 32-symbol trials were able to achieve over 80% fitness and some trials reached over 90% fitness. More variability in the trial performance was observed for 20-point crossover than for uniform crossover. Again, we hypothesize that this is due to early conversion with 20-point crossover because of inferior exploration.

The largest number of symbols that we tested was 64, which results in the same number of input nodes as the 8x8 grid compression method used in Project 4. All 64-symbol trials

achieved fitness values over 90% and the best individual had over 93% fitness. These trials converged early and plateaued, although small improvements did occur at later generations. The fitness level of the 64-symbol individuals is significant because it is close to the best fitness observed for full-zeid 32x32 bitmap perceptron inputs, but 16-times fewer perceptron inputs are required. A fitness level of 93% may be suitable for some applications of digit recognition. Additionally, it is possible that even higher accuracy could be achieved if the algorithm was run with more individuals for more generations. The average final accuracy values per number of symbols used can be found in Figures 5 and 7.

Overall, there was a positive correlation between the number of symbols and the fitness of the algorithm. This is intuitive, because more symbols gives the perceptron more information about the map and makes it more likely that the information required to differentiate digits will be conveyed. Adding hidden layers and backpropagation could help improve the network's accuracy even further on lower-symbol maps.

We selected 32 symbols to run longer GA searches on, as it could already exceed 90% accuracy. We ran several tests with twice as many individuals and 1.5 times more generations, and of 16 tests, the best compression map discovered was able to produce a 93.2% character recognition accuracy. This is very close to the 94.6% accuracy the network was able to achieve using 1024 inputs, so we achieved comparable results with almost 1000 fewer nodes. Our accuracy is also significantly higher than the 65.8% accuracy the 8x8 grid compression scheme was able to produce, even though we use half as many input nodes. A visualization of this compression map can be found in Figure 8. Note that this compression map does not appear to be orderly, yet it produces very good results when used to compress the full-size bitmaps.

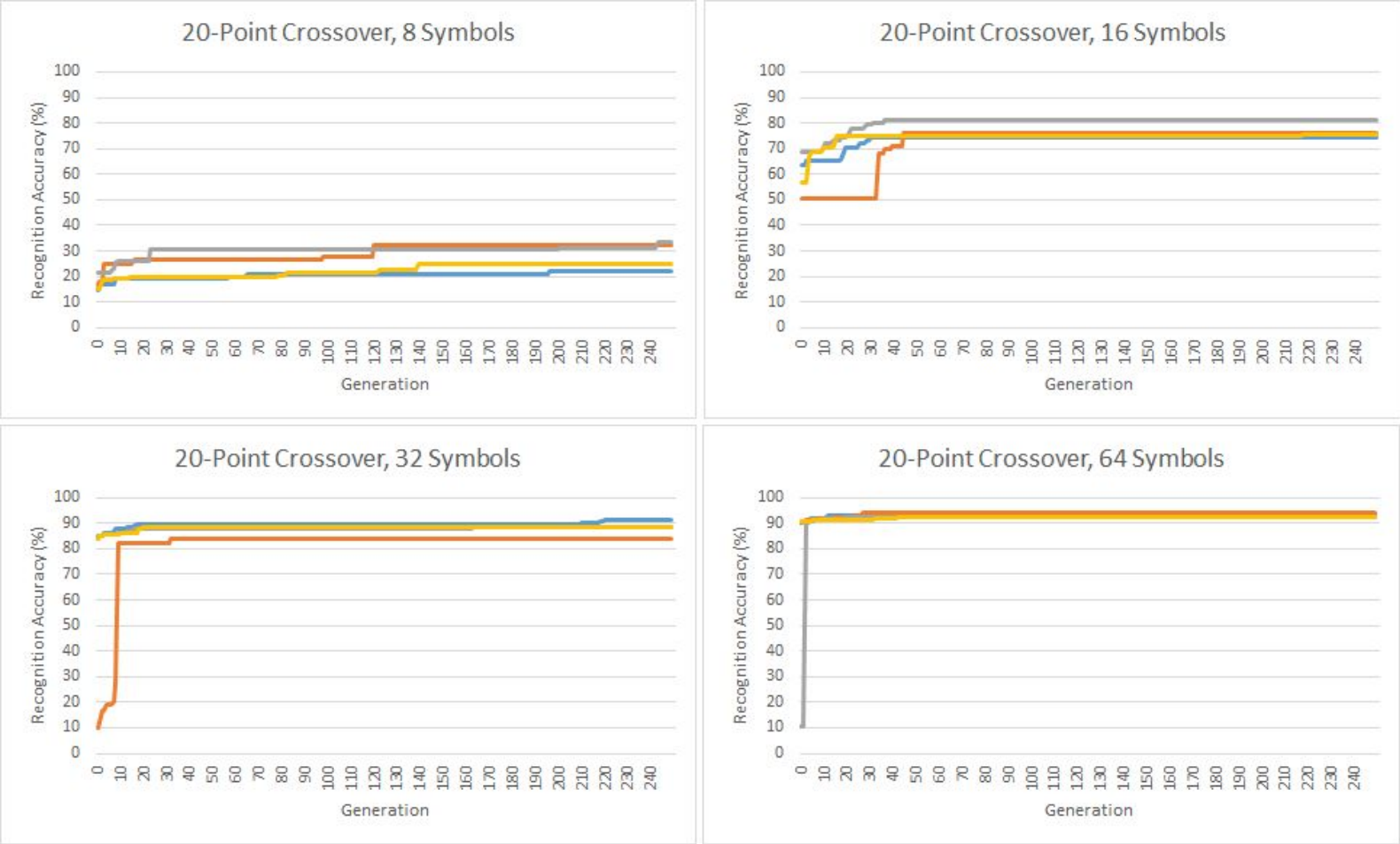

Figure 4: Results of the 20-Point Crossover tests, using NN parameters: learning rate of 0.01, 10 output node, GA parameters: rank selection, 0.8 crossover probability, and 0.01 mutation probability. Each chart displays results of four trials, plotting the perceptron digit recognition accuracy, when using the compression scheme of each generation's Best Individual.

Average Maximum Accuracy Achieved Using 20-Point Crossover

8 symbols	16 symbols	32 symbols	64 symbols
28.073%	76.655%	88.014%	93.048%

Figure 5: Average of the four final accuracy values from each trial, per 20-Point Crossover experiment. Increasing number of symbols increases identification accuracy, as expected.

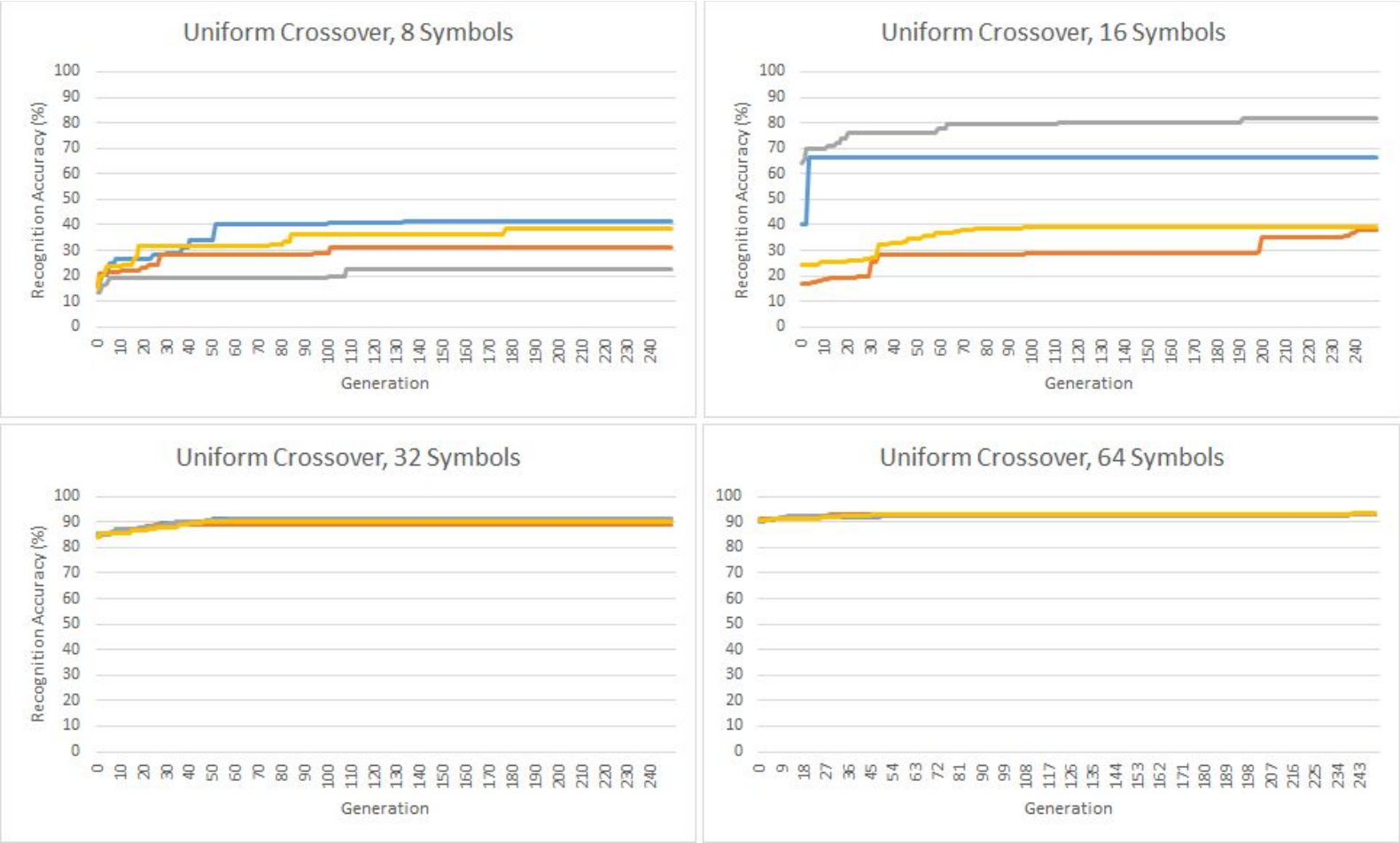

Figure 6: Results of the Uniform Crossover tests, using NN parameters: learning rate of 0.01, 10 output node, GA parameters: rank selection, 0.8 crossover probability, and 0.01 mutation probability. Each chart displays results of four trials, plotting the perceptron digit recognition accuracy, when using the compression scheme of each generation's Best Individual.

Average Maximum Accuracy Achieved Using Uniform Crossover

8 symbols	16 symbols	32 symbols	64 symbols
33.329%	56.313%	90.531%	93.159%

Figure 7: Average of the four final accuracy values from each trial, per Uniform Crossover experiment. Increasing number of symbols increases identification accuracy, as expected.

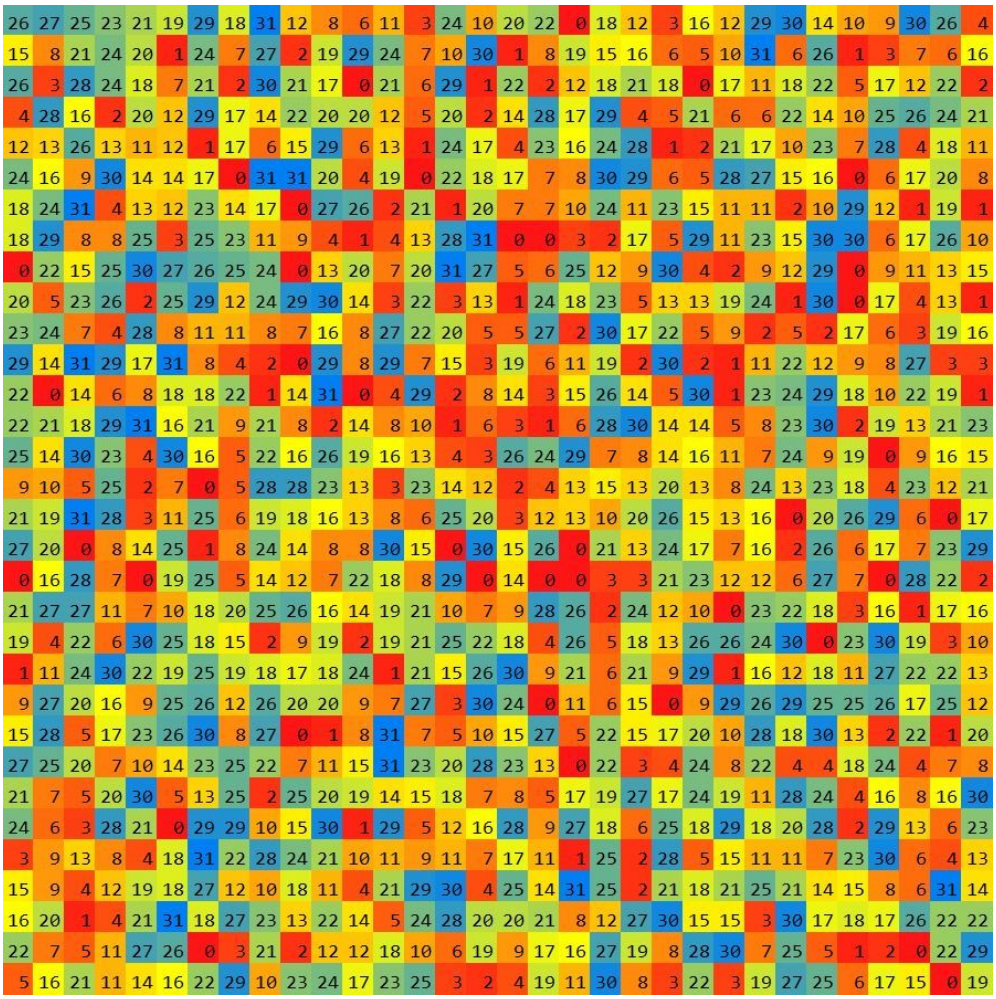

Figure 8: The 32-Symbol compression map, produced by a Genetic Algorithm that our Perceptron network used to compress the full 32x32 bitmap into 32 inputs, and was able to identify handwritten characters with 93.21% accuracy. Each integer has a unique color assigned to it, so that all symbols with the same value has the same color. Note that there does not seem to be many well defined contiguous regions, and that the map is largely disorganized.

8. Further Work

We have some ideas for future work in this area. The first is completely time-performance based. The ability to come up with good compression schemes in the way we have described is time dependant - we get better results with more generations. Since training and testing on a neural net for every individual in every generation is the most time consuming part of the algorithm, we believe a multi-threaded fitness update in GA could speed up the

algorithm immensely. The fitness update for individuals does not need to be a sequential process, since the fitness of one individual does not affect the fitness of any others in the population. Therefore, we can run this operation in parallel. Using Bowdoin's HPC grid, we could speed up our runtime considerably (a node with 56 cores would allow us to run 56x faster). Faster computing would then allow us to use more generations of GA in a reasonable amount of time.

Simply speeding up our algorithm and adding more generations is not enough to increase the effectiveness of our compression methods. We saw many tests where the population converged to a local maximum and was never able to move away from it. We propose a method of manipulating the mutation rate to avoid this problem. We would increase the mutation rate by some amount for each successive generation where the best individual is the same. This method, combined with a large number of generations, may help stagnation in the search process. If after many generations, a better solution is not found, mutation rate would be very high allowing for a very wide exploration of the search space. Given enough generations, this would (ideally) yield a new best result.

Finally, we also believe that added a hidden layer to our perceptron may improve the fitness of our algorithm and make it possible to compress the images with even fewer symbols than for the perceptron. A hidden layer would slow down the runtime of this algorithm, but once a satisfactory individual is discovered, that individuals compression vector could be applied to digit recognition problems without rerunning the algorithm. The mission behind this project is to find the most effective compression vector that can be combined with a neural network to achieve highly accurate digit recognition.

9. Conclusion

Our results show that genetic algorithm bitmap compression algorithm is an effective means producing compression schemes for use with character recognition neural networks. After training a single-layer neural network for only 10 epochs, we were able to achieve better than 93% accuracy with only 32 symbols. It is likely that we can identify even better compression schemes by running the genetic algorithm search with more individuals for more generations, and/or by adding hidden layer in the neural network.

The uniform crossover and N-point crossover methods produced similarly performing compression maps, but the uniform crossover may be less likely to plateau. Low mutation probabilities of around 0.01 were optimal for most generations, but increasing the mutation rate after plateauing for 5 generations helped the algorithm escape local minima in early generations. Rank selection outperformed tournament selection for this implementation, and boltzmann selection was not practical for this problem because the fitness values were very larger, which would cause one individual to dominate the population.

This compression is significantly better the 8x8 integer map compression implemented in Project 4. The neural network also trains and tests significantly faster with 32-symbols as opposed to 32x32 bitmap inputs, which makes this algorithm more suitable for rapid digit recognition. We hypothesize that the 32-symbol compression vector achieves better accuracy than the 8x8 integer map input because the compression vector is evolved to recognize specific, small elements of bitmaps that are particularly useful for differentiating digits that the 8x8 grid is incapable of identifying.