Using Deep Learning to Examine the Classification of Historical Data Through Neural Networks: The Sudoku Puzzle

Ashby Mullin Department of Information of Science and Technology Doane University 1014 Boswell Ave. Crete, NE 68333 ashby.mullinconant@doane.edu

Abstract

Since there are several methods for determining the difficulty of any given Sudoku puzzle, correctly classifying a puzzles difficulty can be challenging. This research project seeks to generate a neural network capable of classifying Sudoku problems within a given scheme. This study focuses on using supervised learning techniques in conjunction with genetic algorithm methods implemented using the Keras and Numpy modules in a Python 3 environment. This paper will discuss the acquisition and manipulation of data for use in training neural networks, the automatic construction of neural networks, evolving populations of neural networks with genetic algorithm concepts, and how each contributes to generating a neural network that can best classify the difficulty of Sudoku puzzles.

Introduction

Sudoku is a logic puzzle enjoyed for the simplicity of rules, yet complexity and challenge of gameplay. Since there are several methods to solve a puzzle, accurately determining the level of difficulty of any given Sudoku puzzle can be challenging. This challenge has sparked interest in developing ways to compute the complexity of Sudoku puzzles. Dissention exists in the field with some individuals determining difficulty based on algorithmic or strategic techniques while others rely on human perceptions of difficulty and average time taken to solve puzzles. Some work has been done using genetic algorithms to solve and rate puzzles (Mantere & Koljonen, 2007), identifying challenges with efficacy. This challenge provides a unique opportunity to test machine learning through Neural Networks (NN).

A NN's ability to accurately and consistently categorize Sudoku puzzles into specific bins of difficulty given a pre-existing classification scheme would allow for the comparison of various classifications schemes. Previous research on NN's has looked at solving Sudoku puzzles (Mladenov, Karampelas, Pavlatos, Zirintsis, 2011), however, little work has explored combining NN with genetic algorithms to rate the difficulty of Sudoku puzzles. To that end, this research project seeks to generate a NN capable of correctly classifying Sudoku problems in a given scheme with above 90% accuracy given an unsolved Sudoku problem. While there are many ways to establish and evolve a NN, this study focused on using supervised learning techniques in conjunction with an island model genetic algorithm implemented using the Keras and Numpy modules in a Python 3 environment. The island model is especially appropriate for this type of study because the sparse research conducted combining NN's and evolutionary computation towards this categorization Sudoku problem.

Background & Conceptual Framework

Neural Networks

As research around NN's develops down divergent and convergent paths of research, it becomes imperative to define some parameters for this study and terms used moving forward consisting of NN's and genetic algorithms used for this study. We operationalize Neural Networks in multiple component parts noted in table 1.

| Component | Description |
|------------|---------------------------------------------------------------------------|
| Layer | A collection of nodes that will propagate information forward in the NN |
| Node | Take in inputs from the previous layer and applies an activation function |
| Activation | A mathematical expression that will transform the inputs to a node into |
| Function | an output for the next layer |
| Weight | A value that adjusts inputs to a node, indicating the node's significance |
| | Table 1: Components of a Neural Network |

Since the goal was to develop a NN that was capable of learning an existing classification scheme, a supervised learning approach was chosen. This approach involves using a labeled dataset, one that has already been classified, to train an NN. Characteristics of the data, called predictors, are used as inputs for the NN. The NN will process these inputs to produce an output value, which is considered the prediction. That prediction is compared to the expected value, also known as the target value or the label. The difference between the two is measured as error, and through back propagation, the weights are updated, and the NN continues this process with the next point in the dataset.

Genetic Algorithms

A genetic algorithm is a type of evolutionary computation where we have a population of objects, called members. Each member is defined by a list of traits, which are values that describe the object. A basic GA follows the following sequence of events: (Sivanandam & Deepa, 2008)

- 1) Create an initial random population
- 2) Evaluate fitness for the population
- 3) Store best individual
- 4) Create a mating pool
- 5) Create the next generations
- 6) if the optimal solution is found, stop
- 7) otherwise reproduce
- 8) perform mutation

The idea behind the island model is to have several populations that evolve, using a GA, independently for a set amount of generations called the migration interval. Then perform a migration where members from one population move to another. This helps to prevent premature convergence in any of the populations.

Developing the Program

Designing the NN and Processing Data

It would be possible to build a NN from the ground up. Specifying the arrays that would contain input values from and those that would contain the weights. This would also require specifying the activation function that would transform the inputs into outputs. Alternatively, there are several modules that can be used to assist with the creation of NNs such as SciKit-learn, Keras, and Tensor Flow. Keras was chosen because it allowed the specification of NN parameters without the need to write the code for every object of the NN (figure 1). Also, Keras has several activation functions readily available within the module. The build_model(A.1) algorithm in Appendix A was designed to generate an NN using passed in parameters.



Figure 1: Visualization of Neural Network

Additionally, a sufficiently large data set of already graded puzzles would need to be vectorized. Meaning, that critical characteristics need to be chosen as predictors (inputs for the NN) and a method devised for extracting the predictors from the data set as a vector of information. The characteristics chosen to be used as predictors were the numbers given in an unsolved puzzle, known as givens. To this end, get_predictors (A.2) was designed to capture the givens as the number provided in each column, row, box, and number class. The parameter passed to this algorithm is a csv of strings containing the puzzle id, unsolved puzzle, solved puzzle, and difficulty delimited by a semi-colon. Additionally, get_targets(A.3) was designed to create a vector of the target values for use in the training process.

Acquiring a better set of training data

The initial dataset had 30,000 sudoku problems classified as very easy or easy. An entity classifying sudoku puzzle rarely uses only two classifications especially ones so close together. This realization instigated a search for a data set with a greater range of difficulties. Which, led to the discovery of a sudoku generator called QQWing Sudoku (Ostermiller) that would create and grade the difficulty based on algorithmic techniques and heuristics discovered during while solving the puzzle. This generator provided advantages by allowing for the generation of simple, easy, intermediate, and expert level puzzles either randomly or at a specified level. Additionally, this program enabled the generation of as many puzzles as needed to build a data set and could write them to csv. One issue which arose was the string output by the generator used periods instead of zeros to denote blank spaces in the unsolved puzzle. Also, the grades were provided as words as opposed to numerical values. Algorithm A.4 was designed to convert the new format of the data to the schema used in the initial dataset.

Implementing a Genetic Algorithm

At this point, the program was able to generate an NN capable of training on data sets given the parameters of the NN were specified. The first attempt at finding involved set intervals of parameters and testing each combination. This took multiple trials, time, and effort resulting in no indications to which direction the parameters should be adjusted. Even with the design constraints of having two hidden layers, and no layer having more than 36 nodes, the search space contained 8,503,056 permutations. Thus, it was decided that a GA should be used to reduce the amount of time searching for the optimal design. The first step was design and implementation of the generation function (A.5) that would generate a population. This algorithm generates random parameters and constructing them into an array that will be returned and housed in an array of members.

This array of members is then used in fitness(A.6), an algorithm that iterates through each member building and training an NN using the traits of the member. The fitness algorithm will then return an array with each member's traits with the training accuracy concatenated to the end of the list of traits. The array with the accuracies is then passed to a selection(A.7), which use tournament selection. This a selection method where the fitness of two members is compared using their accuracies, and the member with higher accuracy is kept in the population.

Once a list of stronger members has been compiled, the list is passed to the breed algorithm (A.8). Breed runs through the members passed in and selects two to be parents that will create a child member. The child member is created by going through and randomly selecting a trait from one of the parents to fill the trait of the child. The child also has a .001 chance of mutating where a random trait is selected and then replaced with a new random value. This creates a new population and completes the generation. This cycle will repeat for a specified number of generations, known as migration interval. At the end of

the current migration interval, a list of all current member in each population is compiled, a migration (A.9) algorithm will select the strongest quarter of each population to stay, the weakest quarter will migrate, be removed from the current population and added to another population. After the migration, the breed algorithm will be run on the new populations, and the next migrations interval will begin and continue for the set number of generations. After all migration intervals have been accomplished the program examines all current member of each population; returning the traits and accuracy of members with the highest accurateness.

Results

These results are based on running the program with three migration intervals of five generations and four populations of eight members. The dataset for this run consisted of 4,000 puzzles equally distributed across simple, easy, intermediate, and expert difficulties in a random order. After about 90 minutes of runtime, the following results were produced.

| ILA | HLN1 | HLA1 | HLN2 | HLA2 | OLA | | | | |
|--------------|------|----------|---------------|---------|----------|--|--|--|--|
| Population 1 | | | | | | | | | |
| Tanh | 20 | Softplus | 5 | Sigmoid | Softsign | | | | |
| Tanh | 7 | Softsign | 5 | Sigmoid | Relu | | | | |
| Tanh | 20 | Softplus | 5 | Sigmoid | Softsign | | | | |
| Tanh | 7 | Softsign | 5 | Sigmoid | Relu | | | | |
| Tanh | 7 | Softplus | 5 | Sigmoid | Relu | | | | |
| Tanh | 20 | Softplus | 5 | Sigmoid | Relu | | | | |
| Tanh | 20 | Softsign | 5 | Sigmoid | Softsign | | | | |
| Tanh | 7 | Softsign | 5 | Sigmoid | Relu | | | | |
| Population 2 | | | | | | | | | |
| Tanh | 20 | Softplus | 5 | Sigmoid | Softsign | | | | |
| Tanh | 20 | Softsign | Softsign 5 Si | | Softsign | | | | |
| Tanh | 27 | Softsign | 4 | Sigmoid | Relu | | | | |
| Elu | 27 | Softsign | 4 | Relu | Relu | | | | |
| Tanh | 20 | Softsign | 5 | Sigmoid | Softsign | | | | |
| Tanh | 20 | Softsign | 5 | Sigmoid | Relu | | | | |
| Elu | 27 | Softsign | 4 | Sigmoid | Relu | | | | |
| Elu | 20 | Softplus | 5 | Relu | Softsign | | | | |
| Population 3 | | | | | | | | | |
| Elu | 27 | Softsign | 4 | Relu | Relu | | | | |
| Elu | 27 | Softsign | 4 | Relu | Relu | | | | |
| Elu | 27 | Elu | 16 | Relu | Softplus | | | | |
| Elu | 27 | softsign | 4 | Relu | Relu | | | | |
| Elu | 27 | softsign | 4 | Relu | Relu | | | | |

| Elu | 27 | Softsign 16 Re | | Relu | Relu | | | | |
|--------------|----|----------------|----|------|----------|--|--|--|--|
| Elu | 27 | Elu | 16 | Relu | Softplus | | | | |
| Elu | 27 | softsign | 4 | Relu | Relu | | | | |
| Population 4 | | | | | | | | | |
| Elu | 27 | Elu | 16 | Relu | Softplus | | | | |
| Elu | 27 | Elu | 4 | Relu | Relu | | | | |
| Elu | 27 | Softsign | 16 | Relu | Relu | | | | |
| Elu | 27 | Softsign | 4 | Relu | Relu | | | | |
| Elu | 27 | Elu | 16 | Relu | Softplus | | | | |
| Elu | 27 | Elu | 16 | Relu | Relu | | | | |
| Elu | 27 | Softsign | 4 | Relu | Relu | | | | |
| Elu | 27 | Elu | 16 | Relu | Relu | | | | |

Table 2: Structure of NN's for each population after last migration interval Note: ILA-Input layer activation function, HLN-Hidden layer no. of nodes, HLA-Hidden layer activation function, OLA-Output layer activation function

Table 2 shows that the structures of the NNs converge reasonably quickly for small populations. In table 3 there is an indication of an increase in accuracy, but given the state of the populations after the third migration interval, it seems unlikely to find a design that could achieve 90% accuracy without using a much larger population.

| Migration Interval 1 | | | Migration Interval 2 | | | Migration Interval 3 | | | | | |
|----------------------|------|------|----------------------|------|------|----------------------|------|------|------|------|------|
| Pop | Pop | Pop | Pop | Pop | Pop | Pop | Pop | Pop | Pop | Pop | Pop |
| 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 |
| .254 | .239 | .254 | .25 | .254 | .254 | .247 | .249 | .254 | .254 | .272 | .257 |
| .254 | .254 | .254 | .254 | .254 | .254 | .262 | .25 | .24 | .254 | .257 | .264 |
| .254 | .254 | .249 | .248 | .254 | .253 | .000 | .245 | .254 | .000 | .251 | .264 |
| .254 | .254 | .254 | .254 | .254 | .254 | .261 | .25 | .000 | .253 | .246 | .255 |
| .254 | .262 | .254 | .244 | .254 | .254 | .27 | .241 | .245 | .254 | .256 | .263 |
| .254 | .254 | .251 | .268 | .254 | .259 | .000 | .242 | .240 | .251 | .262 | .000 |
| .254 | .254 | .252 | .254 | .254 | .242 | .252 | .245 | .254 | .000 | .267 | .269 |
| .254 | .254 | .254 | .254 | .254 | .245 | .234 | .265 | .251 | .254 | .262 | .266 |

Table 3: Accuracy of NN's after each migration interval

Conclusions

There is much more to be explored. However, it would seem that based on the initial run of the program any model following these design constraints might not reach a design that will achieve 90% accuracy and if it is possible, the amount of time to find such a design in the current iteration of the program will be significant.

Future Research

The future goals of this project are to implement the program using parallelism to reduce the wall clock time taken for the program to run. Integrate implementations that will allow for the use of a softmax activation function, which is based on logit, and outputs a probability matrix rather than a single value and could work better for classification.

References

- Becker, D., Bowne-Anderson, H., & Roy, Y., (2018). *Deep Learning in Python*. Retrieved from https://www.datacamp.com/courses/deep-learning-in-python
- Mantere, T., & Koljonen, J. (2007, September). Solving, rating and generating Sudoku puzzles with GA. In Evolutionary Computation, 2007. CEC 2007. IEEE Congress on (pp. 1382-1389). IEEE.
- Mladenov, V., Karampelas, P., Pavlatos, C., & Zirintsis, E. (2011). Solving sudoku puzzles by using hopfield neural networks. Proc. of ICACM, 11, 174-179.
- Ostermiller, S. (2014). QQWing sudoku. Retrieved from https://qqwing.com/
- Sivanandam, S. N., & Deepa, S. N. (2007). *Introduction to genetic algorithms*. Springer Science & Business Media.
- Whitley D., Rana S., Heckendorn R.B. (1997). *Island model genetic algorithms and linearly separable problems*. In: Corne D., Shapiro J.L. (eds) Evolutionary Computing. AISB EC 1997. Lecture Notes in Computer Science, 1305. Springer, Berlin, Heidelberg. doi:10.1007/BFb0027170

Appendix A: Algorithms

A.1 build_model(info[0,..,6]):

instantiate a model add input layer to model add a hidden layer 1 to the model add a hidden layer 2 to the model add output layer to model **return** model

A.2 get_predictors(train):

instantiate a list called data instantiate a numpy array of 36 zeros called inputs open a file stream for train called file initialize a csv reader for file called reader **for** each row in the reader: save current row as sort

for each number in sort:

if the number is not zero:

increment counts in inputs associated with column, row, box, and number add inputs to data reset inputs

return data

A.3 get_targets(train):

instantiate a list called targets open a file stream for train called file initialize a csv reader for file called reader **for** row in reader: add the target value for current row to targets **return** targets

A.4 cleaner():

instantiate a list called output set to_clean to file name to be converted open a file stream for to_clean called file create a reader to read file set count to 0 for row in reader: set orig to row's element 0 set sol to row's element 1 set grade to row's element 11 set new to empty string for row in orig: if character at row is a period: add a 0 to new else: add number at row to new if grade is simple: set grade to 1 if grade is easy: set grade to 2 if grade is intermediate: set grade to 3 if grade is expert: set grade to 4 add string of ["count";"new";"sol";"grade"] to output increment count by 1 open a writer to write to a file "to_clean"clean.csv

for each row in output: write row to file using writer **return**

A.5 gen_pop(pop_size):

instantiate a list called population for i from 0 to pop_size: set ila, ola, hla1, hla2 to a random number between 0 and 9 set hln1, hln2 to a random number between 1 and 36 add a list of form [ila, hln1, hla1, hln2, ola] to population return population

A.6 **fitness**(population, predictors, targets):

instantiate a list called evaluated instantiate an early stopping monitor for each member in population instantiate test_model using build_model() with parameters specified by member compile test_model set model_training to the record accuracy ratings from fitting test_model add member traits concatenated with last element of model_training to evaluated

return evaluated

A.7 selection(pop)

instantiate a list called selected

set i to 0

while there are at least two members left in the pop:

if the current member's accuracy is greater than the next member:

add current member to selected

else:

add next member to selected increment i by two **return** selected

A.8 breed(pop):

copy pop into a new list called new_pop instantiate a list called child **for** each member in pop: create a list of mutations set parent 1 to current member **if** there is a member left in pop:

set parent 2 to the next member

else:

set parent 2 to the first member of pop

for each trait:
set contrib to parent 1 or 2 randomly
if contrib is parent 1:
 copy current trait from parent 1 to child
if contrib is parent 2:
 copy current trait from parent 2 to child
if a random number between 1 and 1000 is less than 10:
 choose a random trait
 replace trait in child with trait from mutations
 add child to new_pop
 reset child
return new_pop

migration(record, populations, pop_size): instantiate a lists called new record, migrators, and stayed for each population: set start to the start of the current population in record copy population to a list called pop set keep to a quarter of pop_size instantiate lists called currentM and currentS while keep is greater than 0: set idW and idS to 0 for each member of pop: if the current member's accuracy is less that the member at idW: set idW to the current members index if the current members's accuracy is greater than the member at idS: set idS to the current members index add member at index idW in pop to currentM add member at index idS in pop to currentS delete members at indices idW & idS from pop decrement keep by 1 add currentM and currentS to migrators reset currentM and currentS for each population: if the current population is the last population: create a list called temp using stayed of current pop and migrators of first pop else: create a list called temp using stayed of current pop and migrators of next pop breed temp for each member in temp: add member to new_record

A.9