Nicholas Barrett 12/3/2020
ASM 595 Final Project Nicholas.barrett@stonybrook.edu

# Sudoku Solver Comparisons

## Introduction and Goals

The projects original objectives were as follows. The plan is to use a convolutional neural network to create some general solvers for sudoku. Sudoku was chosen as there are some good/large data sets readily available for standard sudoku and some variants. The solvers should be able to solve multiple types of sudoku, and most importantly problems not included in the training set. I would like to compare the speeds for execution time and training time for different implementations like, one where the input is a board state, and the output is the full solution vs. the output being a single move. Creating an algorithmic solver or finding one to compare the machine learning based models is also a priority. Python seems like the best option for this as there are some great machine learning libraries and packages available as well as other conveniences that will help along the way. The plan for completion is to train different neural networks on the data, then create an algorithmic solver, and compare all the generated models on previously unseen board configurations, and corner cases. Some interesting results may be found by comparing the models on underdetermined board states and illegal board states. Time permitting, a GUI for easily making a board state for the models to solve is also something I would like to do.

Much of the previously described work was completed. I was able to train different models of neural networks and evaluate them on unseen data and collect the results. I was unable to build an algorithmic solver, but I did find one that was sufficient to test the models against. I did not develop a GUI, only a way to convert a string with spaces to the array format that the model takes as input. The work was done in python using the Tensorflow and Keras libraries and developed in the attached jupyter notebook files. The dataset that was chosen was from Kyubyong Park on Kaggle [1] and this is where my research started.

## Approach and Tools

I started this project knowing little specifics about machine learning, so I had to start with the basics. On the dataset page there are other peoples solutions and models for solving sudoku using the data set, the best machine learning one that I took inspiration from was Dithyrambe [2]. Using the ideas from Dithyrambe and more general introductory documents about keras [3] I was able to start developing a model and strategy for solving sudoku. The general idea is to process the data, feed it into a neural network as 81 numbers in to get 81 numbers out, which hopefully solve the input board state. The type of network that is needed is called a multi class classification network, meaning the network provides its estimate for every available class (the 9 numbers) for each of the 81 board position [4].

First, the data is available in csv format in a downloadable file, and to get that into python I used a pandas data frame as it is very straightforward. The data was then processed into a numpy array and then normalized, as machine learning models train better on normalized data with mean 0. I also split off 1% of the data to use as a comparison set later. Then to define the machine learning model, I used a sequential model, with 1 convolutional layer as input which took a 9x9x1 array, then 2 hidden dense layers, and dense 81x9 output layer. The input and middle layers are where there is room to play, but the output layer is most effective as a 81x9, because we want the output to be the confidence for each possible number (9) for each possible board position (81). Then the keras softmax function can be used to get the highest confidence guess for each number in each board space. When compiling the model using "sparse categorical crossentropy" (SCC) is the best loss function, as our output classes are mutual exclusive, and we don't need our targets to be one-hot encoded vectors, saving time on computation [5]. I also used accuracy as the metric and split off 20% of the data for generating the accuracy metric, note this is the second split of data, and this split is used solely for generating the accuracy and loss data. After setting everything up, we then fit the model to the data, while collecting information about the accuracy and the loss. For my dense model I trained it with a batch size of 64, for 10 epochs and collected the following results.

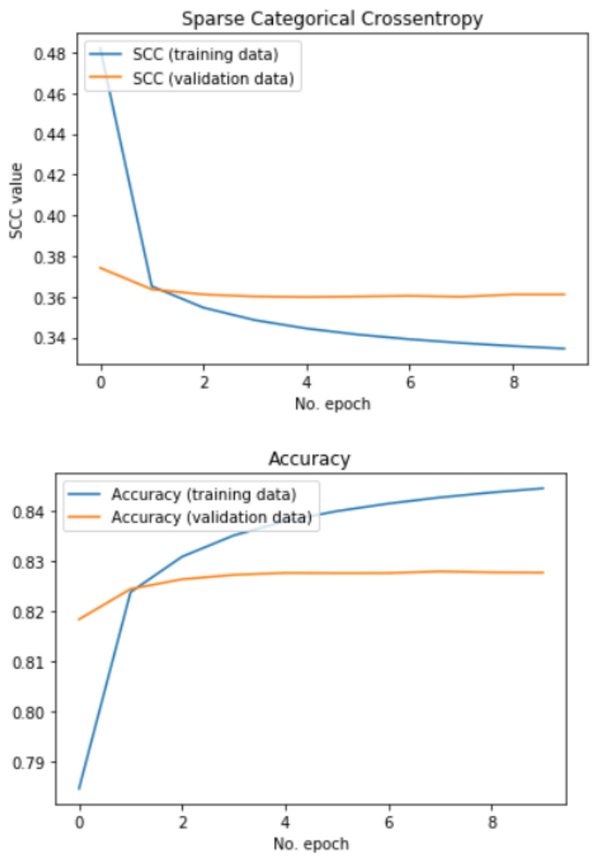
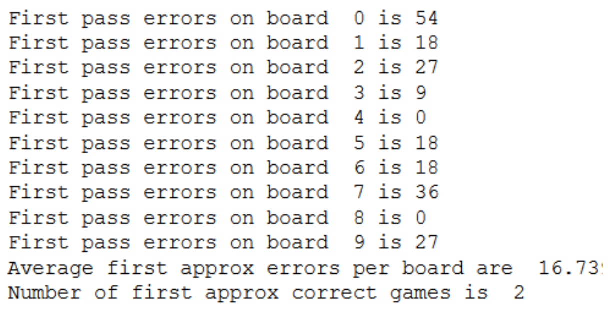


**Figure 1: Training Metrics**

The graphs are meant to visualize the training process to see how the model is doing over time [6]. As we can see the loss (SCC) goes down over the epochs, and training loss starts off high and drops quickly, then maintains a decline. The quick drop is expected as the first weight adjustment is the most significant. Note that the training data hasn't stabilized at a value and still has a negative slope at the last epoch, this means we could keep training the model and it would get better, but this many epochs took my machine 2 hours, so this is where we are stopping for now. The same is true for the accuracy plot, except with positive slopes (which is a good thing).

Based on the accuracy and the loss, it does not look like our model is going to solve anything directly off the bat, but it should be close. After testing 500 of the games the model hasn't seen, the model accurately completed 103 of the 500 on the first pass, which is fairly good, considering this was not the most sophisticated way to train a model. Dithyrambe's solution involved training his model based on removing 1 digit from the completed board, and training the model to find it, then taking 2 digits, and training on that, and so forth. This is a clever approach, but there is a way to achieve similar accuracy without that.

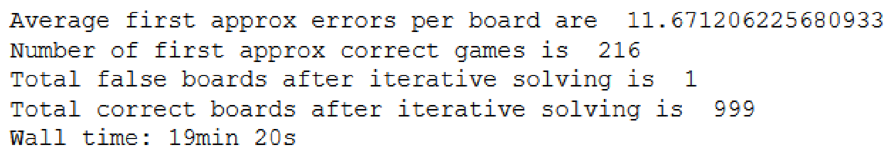
First to test the accuracy we must develop some scripts to test our solutions. That is straight forward, I used 2 functions, one that takes a board state, use the model to make a prediction base on the board state, format it correctly then spit it out. And another that takes a range of unsolved boards, and their solved counterparts, generates a prediction based on the input state, compares the desired output to the prediction, and report errors. The result of this script on 10 test inputs is the following figure.



```
First pass errors on board  0 is 54
First pass errors on board  1 is 18
First pass errors on board  2 is 27
First pass errors on board  3 is 9
First pass errors on board  4 is 0
First pass errors on board  5 is 18
First pass errors on board  6 is 18
First pass errors on board  7 is 36
First pass errors on board  8 is 0
First pass errors on board  9 is 27
Average first approx errors per board are  16.73
Number of first approx correct games is  2
```

**Figure 2: First Pass Errors**

Our first pass accuracy is low, but some of the digits are correct on every board, so let us assume that the highest confidence digit is correct on the board, and then solve iteratively. Meaning take the highest confidence cell that our model predicts, and then insert only that cell into the board, and run it through again. To do this I adapted an algorithm based on a similar approach from Shive Verma [7]. The scripts takes a board state, and while there are zeros (meaning blank) on the board state, predicts a completed board, find the cell with the highest confidence, replace the cell on the board with the highest confidence answer, then repeat. Now we can test the first prediction and the final predictions to see if we have improved in accuracy. The result of the combined functions on 1000 new sudokus is the following figure.



```
Average first approx errors per board are  11.671206225680933
Number of first approx correct games is  216
Total false boards after iterative solving is  1
Total correct boards after iterative solving is  999
Wall time: 19min 20s
```

**Figure 3: Error Comparison**

Now that is substantially better performance, and all we had to do is iterate repeatedly, with 1.16 seconds per solve due to the iteration. Using this approach, the increased the accuracy from 216/1000 to 999/1000. The effectiveness of this method speaks for itself and is why both Dithyrambe and Shiva used some version of it [2] [7].

## Results

For comparisons, I was unable to come up with an algorithmic solution in time, but it is a solved problem so I used a version by James McGuigan, to compare the run times and accuracy that he had [8]. For another neural network architecture using convolutional layers instead of a dense layer structure, we can look at Shivas model and run it this framework, the resulting a table that was reported in the slide is as follows.

| Model | Training Time | End Loss | End Accuracy | Tested Accuracy | Solve Time |
|---|---|---|---|---|---|
| Dense Architecture | 53min 38sec | .345 | .8446 | .999 | 1.15 sec |
| Convolutional Architecture | 57min 41sec | .3600 | .800 | .998 | 1.15 sec |
| Algorithmic | N/A | N/A | N/A | 1.000 | 155ms to 5sec |
| Dithyrambe Dense | N/A | .0181 | .9939 | .997 | N/A |

**Table 1: Solver Comparison**

The algorithmic solver has large variance in solve times as it must do many more computations for harder sudoku, while the iterative ML models we implemented scale better with board state. The algorithmic solver is also infallible, while if not properly trained, machine learning models are not. It is worth noting that the time to code a machine learning model is substantially less than an algorithmic solution, if the data is readily available.

As for the accuracy of machine learning models, the results are similar using a iterated method, with slight variations in tested accuracy. Dithyrambe's training method brought his training accuracy and loss to better numbers than I could get with my dense, or Shiva's convolutional architecture, but the tested accuracy remained about the same. The disparity could be a result of his model overfitting the data by training it on the "same" board ~30-50 times, due to just adding one number at a time, even though that's what he was trying to avoid. Given the end slopes of the accuracy and loss, it may be possible to train my model for longer and with a bigger data set and achieve higher accuracy before substantial diminishing returns.

Given these results it is unlikely that without substantially more data or compute, it would be possible to solve sudoku without iterating, but more work could be done to see if there is a way to determine the fastest and most accurate number of cells to jump ahead, instead of one at a time. For me personally this project was incredibly fun and I am grateful for the opportunity to choose a topic I find fascinating.

## References

[1] https://www.kaggle.com/bryanpark/sudoku/

[2] https://www.kaggle.com/dithyrambe/neural-nets-as-sudoku-solvers

[3] https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/

[4] https://machinelearningmastery.com/multi-class-classification-tutorial-keras-deep-learning-library/

[5] https://datascience.stackexchange.com/questions/41921/sparse-categorical-crossentropy-vs-categorical-crossentropy-keras-accuracy

[6] https://www.machinecurve.com/index.php/2019/10/08/how-to-visualize-the-training-process-in-keras/

[7] https://towardsdatascience.com/solving-sudoku-with-convolution-neural-network-keras-655ba4be3b11

[8] https://www.kaggle.com/jamesmcguigan/z3-sudoku-solver