Solving Sudoku Puzzles with Recurrent Neural Networks

Richard Zhu
Princeton University
ryzhu@princeton.edu

Abstract

Recent advancements in machine learning have allowed for near-human or even superhuman performance in applications spanning chess-playing [12], protein folding [5], and natural language generation [2]. Sudoku, a comparatively simpler task, appears to be an interesting problem. The puzzle can be modelled as a constraint or propositional satisfaction problem (CSP/SAT) which provides the basis for many backtracking-based solvers with 100% accuracy on 3x3 puzzles.[7] However, for the general n x n case of the puzzle these algorithms are unproven and would at best run in exponential time with respect to n. This paper establishes a new state of the art test accuracy of 65.1% for 3x3 Sudoku puzzle solvers using a recurrent neural network (RNN) running in polynomial time. The accuracy is calculated across 10,000 unseen 3x3 Sudoku puzzles of medium difficulty (on average there are 48 blanks on the puzzle board) and is achieved with a shallow, bidirectional RNN with long-short term memory (LSTM) recurrent cell.

Contents

1 Acknowledgements

I would like to express my immense gratitude to Prof. Danqi Chen for the countless hours spent advising me on this project and for first teaching me - along with Prof. Sanjeev Arora - the wonders of machine learning through COS 324. This work is submitted in partial fulfillment of the requirements of the certificate offered by the Program in Applied and Computational Mathematics at Princeton University.

2 Introduction

Machine learning should, in theory, be able to achieve superhuman accuracy at solving n-by-n Sudoku puzzles ("the general Sudoku"), where n is the order of the puzzle. After all, aficionados solve these puzzles by picking up on patterns within the grid space and machine learning excels at pattern recognition. Given recent successes in applying machine learning methods to games like chess [12] or protein folding [5], Sudoku - a game that is simpler in both rule and state-space complexity - seems like a good candidate for some sort of deep neural network. Such a model, once trained would always be capable of solving an n-by-n puzzle in polynomial time, even if each forward step through the model solved only one additional cell. Research regarding neural network-based solvers have emerged recently, [10] [1] though most literature on the puzzle seeks to find solvers that can achieve 100% accuracy, even if they must operate in exponential time.

2.1 Sudoku

An n-by-n Sudoku is a puzzle played on an n^2-by-n^2 grid. Figure 1 shows a 3-by-3 Sudoku puzzle, played on a 3^2-by-3^2 grid.

		7	8		3			
		1		6				2
8	3			5				4
	2			7		9		5
5						1	2	
				9			3	
1		5	4		2			8
	9		6					
	7	3	5	8		4	6	1

Figure 1: Sample 3x3 Sudoku puzzle

The general Sudoku consists of rows (1), columns (2), and blocks (3) as indicated in Figure 2.

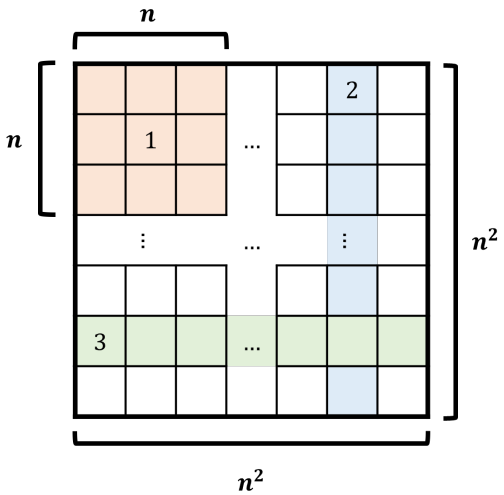


Figure 2: Schematic of n-by-n Sudoku puzzle

The rules of the game are as follows:

- **Rule 1:** Each block (indicated by the n-by-n section of Figure 2 labelled by 1, of which there are n^2 such sections) must contain each of the numbers from $1...n^2$.
- **Rule 2:** Each column (indicated by the column of n^2 cells labelled as section 2) must contain each of the numbers from $1...n^2$.
- **Rule 3:** Each row (indicated by the row of n^2 cells labelled as section 3) must contain each of the numbers from $1...n^2$.

Each number from $1...n^2$ appears exactly n^2 times in the puzzle and there are a total of n^4 cells.

The goal of the puzzle is to fill out values such that rules 1, 2, and 3 (above) are satisfied. A number of clues are given, in the form of cells already filled-in with the correct value (eg. the numbers in Figure 1 are the clues), with a minimum of $2n^2 - 1$ cells must have their values given in order for a single solution to exist. This paper assumes that the Sudoku puzzles presented to the model have a single, unique solution.

The various mathematical representations of the puzzle have been well studied and it has been demonstrated that the general problem is NP-complete [15] through Another Solution Problem (ASP)-completeness shown by reduction from Latin square completion. However, the question arises of how we can model the Sudoku puzzles to best serve a neural network solution, codifying the rules of the game into a set of mathematical constraints. Since existing methods of solving such puzzles rely primarily on search propagation, this approach is well-tested and has a strong likelihood of producing sure-fire solutions.

2.2 Mathematical Representations

The Sudoku puzzle is often modelled as either a constraint satisfaction problem (CSP) or a propositional satisfiability problem (SAT).

The puzzle can be posed as a CSP with $3n^2$ $ALL\_DIFFERENT()$ constraints. Let p[y, x] represent the value of the cell at the y-th row and x-th column (since the puzzle constraints are symmetric the rows can be enumerated starting from the bottom or the top, as can the columns from the left or right) of a completed Sudoku puzzle. For instance, if we take the top left corner to be (y, x) = (1, 1), then p[3, 1]=8. In order for that solution to be valid, the following constraints must be met.

n^2 row constraints:

$$\forall i \in [1, n^2]: \quad ALL\_DIFFERENT(p[i, 1], p[i, 2], ..., p[i, n^2])$$

n^2 column constraints:

$$\forall j \in [1, n^2]: \quad ALL\_DIFFERENT(p[1, j], p[1, j], ..., p[n^2, j])$$

n^2 block constraints:

$$\forall m, n \in [1, n]: \quad ALL\_DIFFERENT(p[i, j] \forall i, j \in D_m, D_n)$$
$$\text{where } D = \{[1, n], [n + 1, 2n], ..., [n(n - 1) + 1, n^2]\}$$

The CSP can then be solved using a backtracking solver to perform search in exponential time complexity [13].

The Sudoku puzzle can also be modelled as a SAT problem. Lynce and Ouaknine 2006 demonstrate a polynomial-time solution to 3x3 puzzles using unit propagation and the failed literal rule.[7] In their extended encoding, which achieves 100% accuracy on 24,260 order 3 puzzles, they convert the puzzle into a SAT problem with the following constraints. For readability, we simplify the 9 constraints proposed by Lynce and Ouaknine into the 5 shown below, which completely constrain the problem. The following constraints encode the 3 rules mentioned in Section 2.1, where s_{xyz} is true iff the cell on the xth row and yth column contains the value z (with x and y ranging from $1...n^2$ and z ranging from $1...n^2$).

There is exactly one number in each entry:

$$\wedge_{x=1}^{9} \wedge_{y=1}^{9} \left(\vee_{z=1}^{9} s_{xyz} \wedge \wedge_{z=1}^{8} \wedge_{i=z+1}^{9} \left(\neg s_{xyz} \vee \neg s_{xyi} \right) \right) \tag{1}$$

Each number appears exactly once in each row:

$$\wedge_{y=1}^{9} \wedge_{z=1}^{9} \left(\wedge_{x=1}^{8} \wedge_{i=x+1}^{9} \left(\neg s_{xyz} \vee \neg s_{iyz} \right) \wedge \vee_{x=1}^{9} s_{xyz} \right)$$

However, upon closer introspection we know that if there is exactly one number in each entry and each number appears at least once in each row, each number must also appear exactly once in each row. Given equation (1), the above reduces to the extended encoding presented by Lynce and Ouaknine (which is actually much simpler than the minimal encoding stating that each number appears at most once in each row, $\wedge_{y=1}^{9} \wedge_{z=1}^{9} \wedge_{x=1}^{8} \wedge_{i=x+1}^{9} \left(\neg s_{xyz} \vee \neg s_{iyz} \right)$).

$$\wedge_{y=1}^{9} \wedge_{z=1}^{9} \vee_{x=1}^{9} s_{xyz} \tag{2}$$

Each number appears at most once in each column:
Again applying the reasoning used in the constraint represented by equation (2), and given equation (1), we get the extended encoding in Lynce and Ouaknine.

$$\wedge_{x=1}^{9} \wedge_{z=1}^{9} \vee_{y=1}^{9} s_{xyz} \tag{3}$$

Each number appears at most once in each 3x3 sub-grid: Given equation (1), the minimal encoding is,

$$\wedge_{z=1}^{9} \wedge_{i=0}^{2} \wedge_{j=0}^{2} \wedge_{x=1}^{3} \wedge_{y=1}^{3} \wedge_{k=y+1}^{3} \left(\neg s_{(3i+x)(3j+y)z} \vee \neg s_{(3i+x)(3j+k)z} \right) \tag{4}$$

$$\wedge_{z=1}^{9} \wedge_{i=0}^{2} \wedge_{j=0}^{2} \wedge_{x=1}^{3} \wedge_{y=1}^{3} \wedge_{k=y+1}^{3} \wedge_{l=1}^{3} \left(\neg s_{(3i+x)(3j+y)z} \vee \neg s_{(3i+k)(3j+k)z} \right). \tag{5}$$

The full extended encoding (as opposed to the more concise variant of the encoding reproduced here) is what achieved the group's high accuracy with the basic encoding producing only . New research has arisen regarding SAT solvers that rely on neural networks, [14] however to recreate the SATNet layer would pose additional challenges, including resolving a very large train-test accuracy differential in Wang et al. 2019's results, so we attempted instead to indirectly encode the rules within the model parameters.

2.3 Recurrent Neural Networks

Recurrent neural networks (RNNs) form the underlying architecture necessary to apply modern machine learning design patterns to variable length inputs. They are defined as multi-layer-capable neural networks with closed loop feedback, which enables a relatively small number of parameters (as opposed to a fully-connected layer operating on the same input) to encode relationships across the entire input sequence. This has enabled significant advances in applications involving prediction over some form of temporal progression, spanning language models, weather forecasting, music generation, financial market prediction, and dynamical system evolution. However, RNNs often struggle to encode higher order relationships and patterns in the way that a convolutional neural network might. Figure 3 show samples from the first convolutional layer of a deep Convolutional Neural Network (CNN) on a dataset of hand-written digits. We notice that the activations correspond roughly with the edges of various handwritten numbers and form the "building blocks" for recognition of digits.

Recurrent relational networks (RRNs) have seen significant success in solving Sudoku puzzles in an iterative, multi-step approach. RRNs have solved 96.6% of the hardest Sudoku puzzles. [10]

Machine learning, more broadly, has seen significant advances in achieving accuracy approaching or exceeding human performance: GPT-3 in the natural language generation realm [2], AlphaZero in the game-playing arena [12], and AlphaFold in the field of protein structure prediction [5].

The variable-sized input makes it difficult to design the architecture of the RNN with the CSP or SAT constraints embedded within the architecture directly. Given that the input is read as a sequence (the Sudoku puzzle is reshaped from an n^2 x n^2 grid of numbers to a 1 x n^4 vector in row-major order, before being run through one-hot encoding), many of the positional relationships are lost (i.e. the first and last element of any column are very important, but occur very far apart from each other in the sequence).

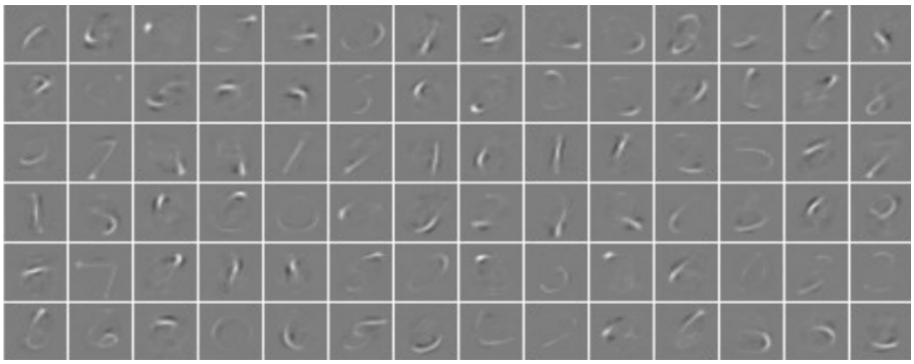
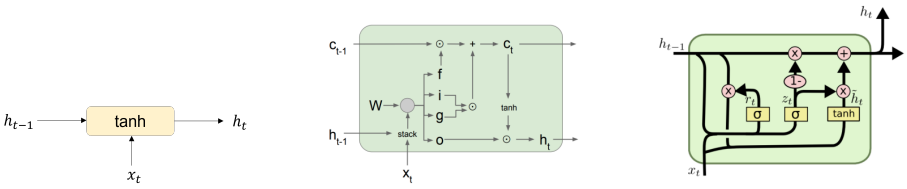


Figure 3: Samples from the first convolutional layer of a CNN, after training on the MNIST dataset. Graphic from [9]

3 Methods

3.1 Architecture

We use three primary RNN cell architectures: vanilla RNN (with simple tanh activation), long-short term memory (LSTM)[4], and gated recurrent unit (GRU)[3]. While the first is a good baseline, the latter two were expected to outperform given the explicit features designed to encode long-range dependencies.



(a) Vanilla RNN cell (b) LSTM cell (c) GRU cell

Figure 4: 2 Figures side by side

Each of the three cells is shown above (LSTM and GRU cell illustrations from [8]), with x_t being the input at time step t (whether from the input sequence or the output from the previous layer), h_t being the value of the hidden vector at time step t, and c_t being the value of the memory cell - a vector representing long-range dependencies used in the LSTM cell. The output of a cell (not pictured), y_t, has the same value as h_t.

3.2 Data and pre-processing

A dataset of 1 million 3x3 Sudoku puzzles (with solutions) is used for this report and is available at `https://www.kaggle.com/datasets/bryanpark/sudoku`. A sample puzzle/solution pair is shown in Table 1. There are a few properties of the dataset, as can be noted in the aforementioned table. The data takes the form of an 1x81 vector, with each cell taking on a single value from 1...9. A 0 in the problem represents a blank cell and there are no 0's in the solution. We split this dataset into three segments, with the train set being the first 10,000 entries, the development set being the second-to-last 10,000, and the test set being the last 10,000. The first two sets are "seen" during the training process, meaning that we tweak model parameters and hyperparameters to maximize accuracy/minimized loss on the train and development sets, respectively. More specifically, the train set is used for training the model and we perform gradient descent of the RNN on this set. The development set is used to determine optimal hyperparameters and the test set is used to evaluate the quality of our model's predictions.

1 million Sudoku games dataset	
Puzzle	600120384008459072000006005000264030070080006940003000310000050089700000502000190
Solution	695127384138459672724836915851264739273981546946573821317692458489715263562348197

Table 1: Sample puzzle/solution pair in the 1 million order 3 puzzle dataset

We perform one-hot encoding (OHE) on each of the sets. We add an extra dimension to the dataset with a single 1 at the 1-index (an index starting at 1 rather than 0) representing the value of the cell. For example, take train_x to represent the matrix containing all the puzzles in the train set before encoding and train_xenc to represent the same matrix after encoding. If train_x[1, 2] = 5, that is the 2nd element in the 1st example (in 1 x 81 representation) has a value of 5, then train_xenc is constructed such that train_xenc[1, 2, 5] = 1, and such that train_xenc[1, 2, i] = 0 $\forall i | i \neq 5$. The resulting dimensions of each set, before and after OHE, is indicated in Table 2.

Set	Dimension (before one-hot)	Dimension (after one-hot)
Train	N_{train} x d_{puz}	N_{train} x d_{puz} x n^4
Development	N_{dev} x d_{puz}	N_{dev} x d_{puz} x n^4
Test	N_{test} x d_{puz}	N_{test} x d_{puz} x n^4

Table 2: Dimensions of puzzle/solution matrices in dataset partitions

The variables in Table 2 are defined as follows:

1. N_{train}: the number of train examples, in this case 10,000
2. N_{dev}: the number of development examples, in this case 10,000
3. N_{test}: the number of test/validation examples, in this case 10,000
4. d_{puz}: the number of cells in the puzzle, in this case 81
5. n: the order of the puzzle, as defined previously, in this case 3. We note that in general the value of d_{puz} is equal to n^4, however we keep these as separate variables for clarity.
6. d_{batch}: the size of the batch. A value of 256 has been found to work particularly well.

We also split each set into batches (ranging from 256-2048 examples per batch) to increase the complexity of models (i.e. hidden dimension size and number of layers) given limited GPU memory and to implement mini-batch gradient descent for smoother loss curves (and thus steadier descent to a loss optimum). This does not add any additional dimensions to the data sets, but rather creates a list of matrices with number of examples, I, equal to the batch size.

The input to all models noted below is a single batch of one-hot embedded examples (with dimension d_{batch} x d_{puz} x n^4). The output is also a single batch of embedded examples with the same dimesnion. However, the values along the third dimension are not one-hot but rather are real-values probability-like values, which are subjected to a softmax function that converts these values into probabilities, summing to 1 along the 3rd axis. These values represent the probability of the 1-index of that cell (along the third dimension) being the value in the Sudoku puzzle. The index with the maximal probability is taken to be the prediction, using PyTorch's max function. This function performs an *argmax* over the 3rd axis and returns the index. In each of the five iterations we perform course adjustments of hyperparameters such as hidden dimension, number of layers, and the number of epochs in search of an optimal architecture.

To measure accuracy, we divide the number of correct cells (cells where the prediction value matches the gold value) by the number of total cells in the puzzle. We decide not to use F-1 scores or other metrics, since we care only about maximizing the number of correct values in a multi-class classification problem.

For all models, we use cross entropy loss during training, stochastic descent with Adam optimization, and a learning rate scheduler with adjustable decay. We use an initial learning rate of 0.01 throughout this paper and there is no dropout applied during training.

3.3 Iteration 1: Unidirectional vanilla RNN

Experimentation began with a vanilla RNN - an Elman RNN with tanh activation, a 2-layer variant of which is illustrated in Appendix B.1. Each cell is governed by the following relation, where h_t, x_t are defined previously. W is the weight matrix associated with the recurrent cell, and b represents a bias term. In this case there are two weight matrices and biases, one for the input, x, and one for the hidden state vector, h.

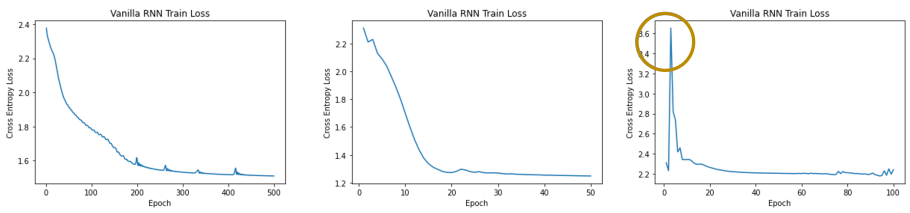
$$h_t = tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh})$$

We append a single fully-connected layer across the hidden dimension to make a more effective judgement of the final predicted as a function of the entire hidden state/output. This turns out to be very effective, and we maintain this fully-connected layer across each iteration of the architecture. A summary of three top performing model configurations is shown in Table 3. These examples were chosen because they give a good picture of the progression of test accuracy across hidden dimension, layer number, and epoch number adjustments. We note that epochs run very quickly at this layer and hidden dimension size, and large epoch numbers are required for suitable results. This architecture is named RNN-FC, since it is the vanilla RNN configuration with a fully-connected layer. Numbers are appended to this abbreviation to distinguish different hyperparameter configurations.

Unidirectional vanilla RNN performance			
	RNN-FC1	RNN-FC2	RNN-FC3
Architecture:	Elman RNN with tanh ending with fully connected layer	Elman RNN with tanh ending with fully connected layer	Elman RNN with tanh ending with fully connected layer
Hidden dimension:	5	100	243
Layers:	5	2	3
Epochs:	500	50	100
Training time (per epoch):	0.156s (GPU)	0.235s (GPU)	1.47s (GPU)
Terminal training loss (cross entropy):	1.5100	1.2484	2.2392
Train accuracy (10k puzzles):	34.84% (0.552s)	**49.98% (0.422s)**	11.10% (0.728s)
Validation accuracy (10k):	34.83% (0.406s)	**49.78% (0.453s)**	11.10% (0.922s)
Test accuracy (10k):	34.85% (0.526s)	**49.87% (0.473s)**	11.10% (0.769s)

Table 3: Performance of unidirectional vanilla RNN across various hyperparameter ranges (hidden dimension size, layers, and epochs). Bolded values represent the highest accuracy.

Observing the train-loss progression over epoch number for each of these (see Figure 5, we note that both RNN-FC1 and RNN-FC2 appear that they could benefit from additional epochs as the training loss does not appear to have reached steady state. In the RNN-FC3, there appears to be an anomalous spike which could be explained by the much larger dimension size.



(a) RNN-FC1 (b) RNN-FC2 (c) RNN-FC3

Figure 5: Cross entropy loss versus epoch number during training

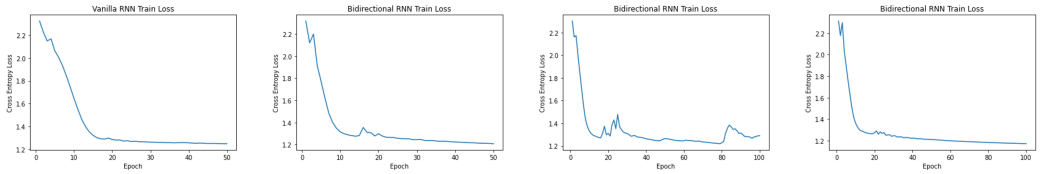
3.4 Iteration 2: Unbatched bidirectional vanilla RNN

BRNN-FC performances with varying test accuracies are shown in Table 4. We stop measuring time taken to run accuracy tests due to the similarity in numbers once accuracy calculations are vectorized. We notice interestingly that the network with the most parameters performs worst.

Bidirectional vanilla RNN performance				
	RNN-FC2	BRNN-FC1	BRNN-FC2	BRNN-FC3
Architecture:	Elman RNN with tanh ending with fully connected layer	Bidirectional Elman RNN with tanh ending with FC	Same as left with 100 epochs	Same as left with learning rate decay=0.995
Hidden dimension:	100	100	100	100
Layers:	2	2	2	2
Epochs:	50	50	100	500
Training time (per epoch):	6.63 s (CPU)	0.797 s (GPU)	0.778 s (GPU)	0.783 s (GPU)
Terminal training loss (cross entropy):	1.248	1.2079	1.2911	1.1037
Train accuracy (10k puzzles):	11.05% (34.877s)	52.38% (43.97s)	50.15% (0.487s)	**58.12%**
Validation accuracy (10k):	11.01% (35.771s)	52.13% (40.284s)	50.02% (0.521s)	**57.21%**
Test accuracy (10k):	11.06% (177.77s)	52.11% (212.909s)	50.06% (0.626s)	**57.22%**

Table 4: Performance of bidirectional vanilla RNN.

The loss of each model during training is shown in Figure 6. Of note is the significant volatility in the BRNN-FC2 training process.



(a) RNN-FC2 (b) BRNN-FC1 (c) BRNN-FC2 (d) BRNN-FC3

Figure 6: Cross entropy loss versus epoch number during training

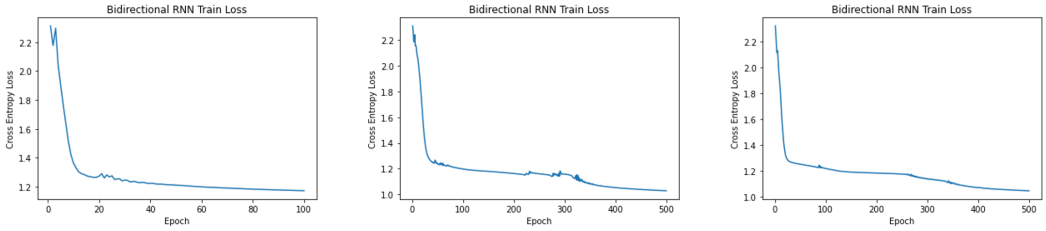
3.5 Iteration 3: Unbatched bidirectional LSTM/GRU

Vanilla RNN cells ”forget” information occurring at timesteps far away from the current one. However, LSTM and GRU models have the capacity to delete unused and keep important data using hidden state and memory cell vectors. Input, forget, reset, update, and output gates handle the writing and reading of information to this memory. The top performing GRU and LSTM models are compared to the current BRNN-FC in Table 5.

Bidirectional LSTM and GRU performance vs. vanilla RNN			
	BRNN-FC	BLSTM-FC	BGRU-FC
Architecture:	Same as left with learning rate decay=0.995	Bidirectional LSTM with FC	Bidirectional GRU with FC, decay=0.995
Hidden dimension:	100	100	100
Layers:	2	1	1
Epochs:	500	500	500
Training time (per epoch):	0.778s	0.794s	0.854s
Terminal training loss (cross entropy):	1.1037	1.0291	1.0447
Train accuracy (10k puzzles):	**58.12%**	56.05%	53.73%
Validation accuracy (10k):	**57.21%**	55.61%	53.23%
Test accuracy (10k):	**57.22%**	55.54%	53.34%

Table 5: Performance of bidirectional LSTM and GRU vs. vanilla RNN.

The loss of each model during training is shown in Figure 7. The LSTM and GRU losses have not yet stabilized and training for additional epochs could further improve test accuracy. the bidirectional RNN continues to outperform the LSTM and GRU models, though this may be due to the additional layer in the former.



(a) BRNN-FC (b) BLSTM-FC (c) BGRU-FC

Figure 7: Cross entropy loss versus epoch number during training

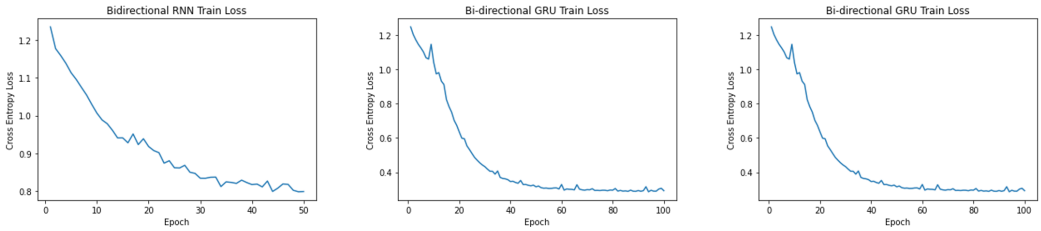
3.6 Iteration 4: Batched bidirectional GRU

Investigating various architectures for a GRU-based RNN, the model appears to reach a substantially smaller loss upon batching (2/3 reduction). There is a marked improvement in train accuracy by adding the additional layer, reinforcing the theory presented in the previous iteration. However the overfitting outpaces that of the bidirectional BRNN-FC3 and test accuracy lags behind that of BRNN-FC3. A subset of batched bidirectional GRU trials are shown in Figure 6.

Batched bidirectional GRU performance vs. vanilla RNN			
	BRNN-FC3	BGRU-FC1	BGRU-FC2
Architecture:	Bidirectional Elman RNN with tanh ending with FC with learning rate decay=0.995	Bidirectional GRU with FC, decay=0.995	Bidirectional GRU with FC, decay=0.998
Hidden dimension:	50	100	100
Layers:	2	1	2
Epochs:	100	500	100
Batch size	256	N/A	256
Training time (per epoch):	1.550s	0.854s	0.863s
Terminal training loss (cross entropy):	0.7988	1.0447	0.2932
Train accuracy (10k puzzles):	**59.60%**	53.73%	63.74%
Validation accuracy (10k):	**58.69%**	53.23%	58.46%
Test accuracy (10k):	**58.73%**	53.34%	58.50%

Table 6: Performance of bidirectional GRU vs. vanilla RNN.

The loss of each model during training is shown in Figure 8.



(a) BRNN-FC3 (b) BGRU-FC1 (c) BGRU-FC2

Figure 8: Cross entropy loss versus epoch number during training

Surprisingly, the BRNN-FC3 despite having the highest accuracy appears to have plenty of room for improvement, compared to the BGRU-FC1 and -2 plots which appear to have stabilized around 1 and 0.29 train loss respectively.

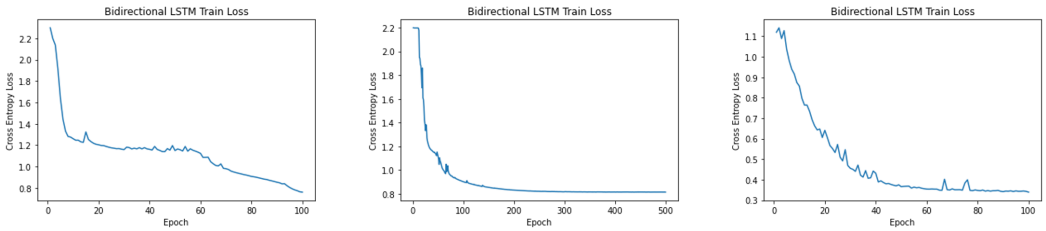
3.7 Iteration 5: Batched bidirectional LSTM

LSTM model performance is shown in Figure 7. Running a few different variants gives increasingly strong results. It appears that there is a delicate balance across learning-rate decay adjustments (strong effect on terminal loss). It also appears that RNN's that are too deep are hard to train, which results in lower test accuracies and more overfitting.

Batched bidirectional LSTM performance			
	BLSTM-FC1	BLSTM-FC2	BLSTM-FC3
Architecture:	Bidirectional LSTM with FC, decay=0.998	Bidirectional LSTM with FC, decay=0.998	Bidirectional LSTM with FC, decay=0.998
Hidden dimension:	500	100	100
Layers:	2	5	3
Epochs:	100	100	100
Batch size	1024	512	256
Training time (per epoch):	1.007s	2.092s	1.673s
Terminal training loss (cross entropy):	0.9358	0.8141	0.3392
Train accuracy (10k puzzles):	66.49%	63.18%	**68.04%**
Validation accuracy (10k):	62.96%	60.63%	**64.94%**
Test accuracy (10k):	62.97%	60.62%	**65.06%**

Table 7: Performance of bidirectional LSTM.

The loss of each model during training is shown in Figure 9. However, while BLSTM-FC2 and -3 stabilize around 0.81 and 0.34 training loss, BLSTM-FC1 appears to have much more potential for decreasing loss.



(a) BLSTM-FC1 (b) BLSTM-FC2 (c) BLSTM-FC3

Figure 9: Cross entropy loss versus epoch number during training

4 Discussion and Conclusion

Iteration 1 has a particularly low test accuracy in RNN-FC3 of 11.10% which is about the same as randomly guessing values. Given that across all 1 million puzzles in the dataset, 41% of the grid is filled out already, we would expect an RNN to achieve at least this score at minimum. All the RNN would have to do is learn that any non-zero values in the puzzle corresponded directly to values in the solution. Though this does appear to be a somewhat non-trivial task, our RNN. We expect a poor score due in large part to the unidirectional layer of the RNN. Since predictions for the first few cells are generated without "seeing" the input cells later in the sequence, despite having strong conditional dependence on those values (i.e. the first value in a column is highly dependent on the last value in that column), we expect the RNN to be effectively guessing at random large parts of the puzzle, especially towards the upper left corner of the puzzle. RNN-FC3 likely achieves a low accuracy and high training loss due to the size of the model, and low number of epoch trained. However, the

training loss plateaued, meaning that additional epochs did not have a significant effect on terminal training loss. Looking at the confusion matrix (Figure 10), RNN-FC3 is essentially only predicting 2's, 4's, and 6's, which concurs with our hypothesis.

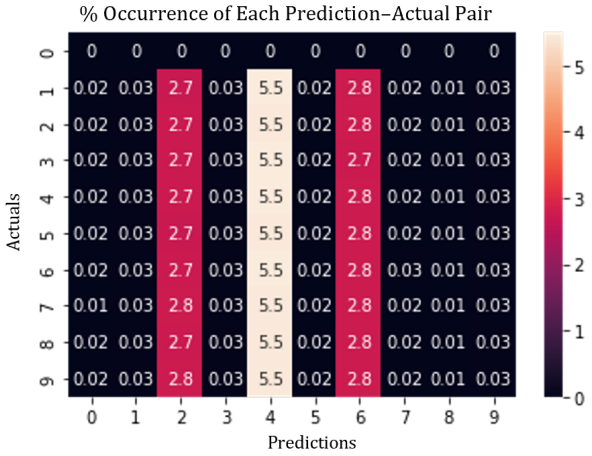


Figure 10: Confusion matrix across prediction-actual pairs in RNN-FC3

Iterations 2-5 contain various iterative approaches to improving the score through manipulating the underlying architecture. Intuitively, the under-performance by the GRU relative to the LSTM cell is justified since the latter has an explicit cell state vector for long-term memory in addition to the hidden state vector.

The confusion matrix in Figure 11 shows the final % occurences of each prediction-actual pair in the model that had the highest test accuracy overall. The correct pairs occur an order of magnitude more often than individual wrong pairs, however there is still much work to do.

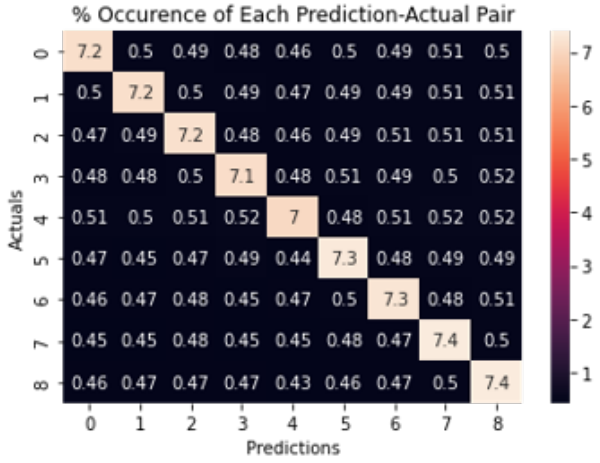


Figure 11: Confusion matrix across prediction-actual pairs in top performing BRNN-FC model

We establish a new state of the art with 65.06% accuracy on the "1 Million Sudoku Puzzle" dataset. Others have achieved training accuracy of 46.7% on similar 3x3 Sudoku puzzles with 46-49 blanks in each puzzle - very close to the average 48 blanks in our dataset, however the test dataset consisted of only 100 puzzle-solution pairs. [1]

The code used in this paper can be found at `https://github.com/RichardZhu123/independent_work`

5 Future Work

In the future, we could consider a customized RNN cell, additional experiments on the effect of number of layers/hidden dimension, and the effect of accuracy on training size for various dimensions. Next steps may also include performing additional experiments with more complex architectures (such as the RRN) as opposed to merely tuning hyperparameters. Though there was reasoning behind the number of layers chosen (each layer encodes a different level of relationship) and the minimum hidden dimension size used (dimension of at least n^4), there could have been a more mathematical justification for these values. The use of skip connections may also be investigated to improve predictions on multi-layered neural networks.

Although we attempted to directly run the RNN on the CSP and SAT problem, we were unable to find a suitable way to configure the RNN (specifically our top-performing architecture, the LSTM) model to solve this problem in polynomial time.

References

[1] Charles Akin-David and Richard Mantey. Solving sudoku with neural networks. *Stanford University - CS230*, 2018.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, and et al. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[3] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[4] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[5] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, and et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.

[6] Stan Kelly-Bootle. Anything su doku, i can do better: The new puzzle craze from japan is sweeping the world, and testing our boolean logic. *Queue*, 3(10):56–ff, dec 2005.

[7] In^es Lynce and Jo¨el Ouaknine. Sudoku as a sat problem. *International Symposium on Artificial Intelligence and Mathematics*, 9, Jan 2006.

[8] Karthik Narasimhan. L11: Recurrent neural networks (ii), Jan 2022.

[9] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015.

[10] Rasmus Berg Palm, Ulrich Paquet, and Ole Winther. Recurrent relational networks for complex relational reasoning. *CoRR*, abs/1711.08028, 2017.

[11] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. *Interspeech 2014*, 2014.

[12] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, and et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[13] Milos Simic. How to solve constraint satisfaction problems, Feb 2022.

[14] Po-Wei Wang, Priya L. Donti, Bryan Wilder, and J. Zico Kolter. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. *CoRR*, abs/1905.12149, 2019.

[15] Takayuki Yato. Complexity and completeness of finding another solution and its application to puzzles. *University of Tokyo Graduate School of Science*, 9, Jan 2003.

A Complexity of n-by-n Sudoku and an ML solution

A.1 Upper bound on state-space complexity

The generalized n-by-n Sudoku puzzle (see Figure 12) is an NP-complete problem [15] with an upper limit of $(n^2!)^{n^2-1}$ on the number of possible board configurations, including illegal ones. This is a tighter bound than the value proposed by Kelly-Bootle in the Association for Computing Machinery (ACM)'s Queue journal of $(n^2!)^{n^2}$ [6]. Our value was calculated as the product of the number of potential configurations in each column. The first column has $n^2!$ possible configurations since the first cell can take on n^2 possible values. The second cell in that column can only take on any of the values $1...n^2$ except for the value of the first cell, so it can only take on $(n^2 - 1)$ values. We know that for the cells in the last column, there will only be one possible value such that each row contains the numbers $1...n^2$. Thus, we only care about the first $n^2 - 1$ columns and determine the upper limit of $(n^2!)^{n^2-1}$.

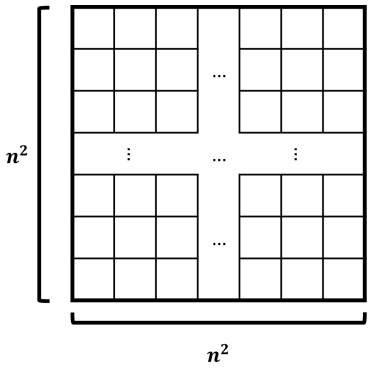


Figure 12: A generic n-by-n Sudoku puzzle

A.2 Time complexity of an ML approach

The time complexity of training an LSTM model on a single example is $O(n^2)$ [11], which can be generalized to a vanilla RNN and GRU given the identical underlying architecture - differing only in the RNN cell structure. We assume that the number of training examples required for high accuracy follows a power law dependency on n, that is αn^β. This means that training takes $O(n^{\beta+2})$ time.

A single layer of a recurrent neural network architecture takes $O(n^4)$ operations and as we add additional forward/backward layers the time complexity scales by a constant. Thus, the time complexity at inference (a full forward pass getting us from an input puzzle to a solution output) will also be $O(n^4)$

Thus, the time complexity for recurrent neural networks (including vanilla, LSTM, and GRU) for both training and inference is

$$\begin{cases} O(n^4) & \beta \leq 2 \\ O(n^{\beta+2}) & \beta > 2 \end{cases}$$

which is polynomial time in the dimension of the puzzle, regardless of the value of β.

B Illustrations of RNN architectures

B.1 RNNFC

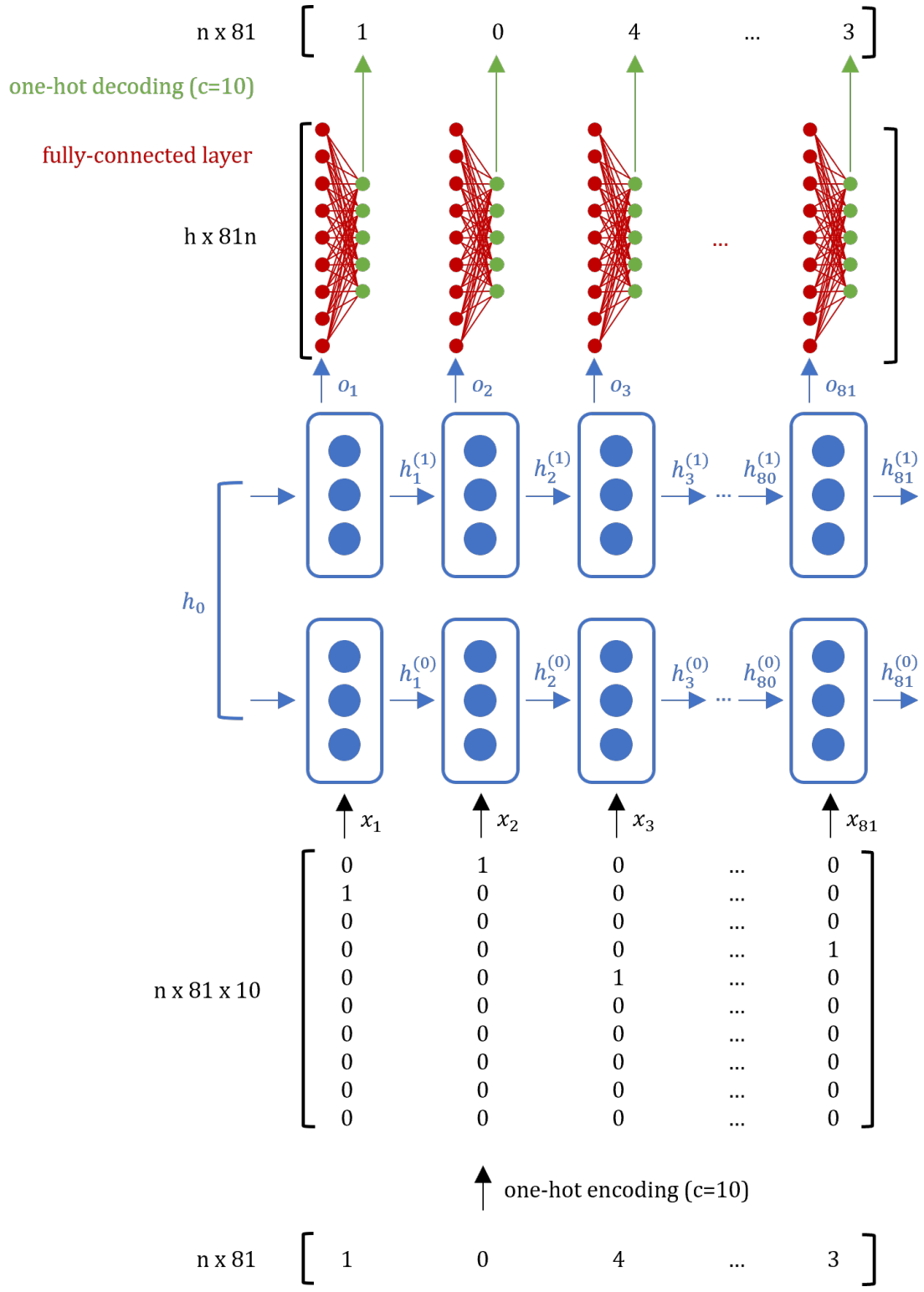


Figure 13: Schematic of uni-directional RNN with fully-connected layer, with 2 layers

B.2 BRNNFC

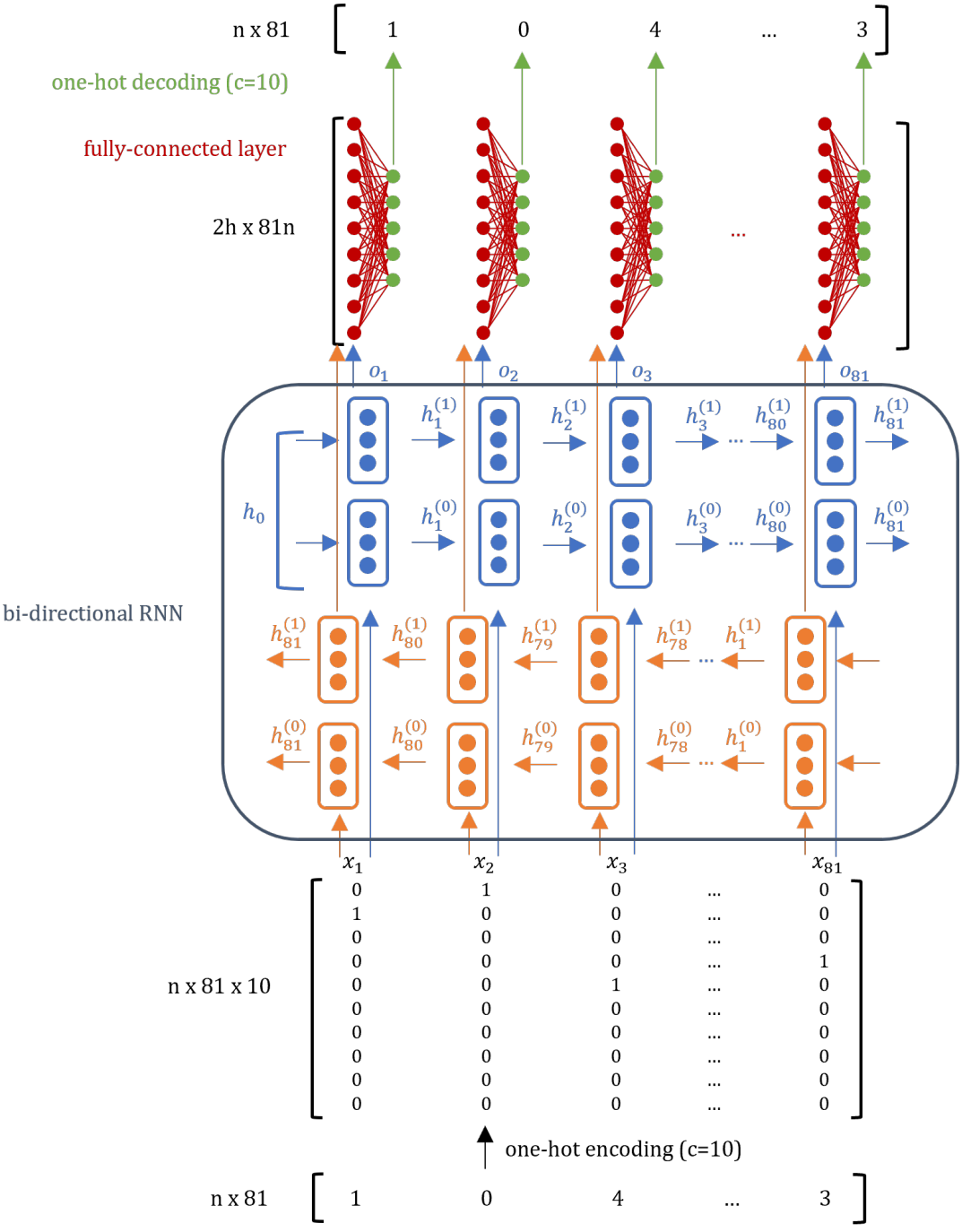


Figure 14: Schematic of bi-directional RNN with fully-connected layer, with 2 layers

C Illustration of a solve puzzle in the test set

Puzzle

		7	8		3			
		1		6				2
8	3			5				4
	2			7		9		5
5						1	2	
				9			3	
1		5	4		2			8
	9		6					
	7	3	5	8		4	6	1

Solution

2	5	7	8	2	3	2	6	1
5	4	1	4	6	7	8	8	2
8	3	6	2	5	2	1	6	4
1	2	6	3	7	8	9	3	5
5	9	6	6	6	4	1	2	4
6	8	7	5	9	5	4	3	5
1	6	5	4	3	2	6	9	8
8	9	8	6	1	8	5	5	7
2	7	3	5	8	9	4	6	1

Sample prediction (bidirectional RNN w/ FC), black indicates provided values, green indicates corrected predicted values, and red indicates incorrectly predicted values.

Figure 15: Example of a puzzle solved by a BRNN-FC