SAT Solver using Recurrent Neural Network

Sumanth Katnam vvk5231@psu.edu
Mahfuza Farooque mff5187@psu.edu

Pennsylvania State University, Department of Computer Science

Abstract

In many areas of computer science and engineering where effective search is necessary, Boolean SAT solvers are essential tools. This frees up users from having to develop their own search algorithm, and it also makes use of the startling efficiency of contemporary SAT solvers. In both existing and future approaches for such assertions, neural networks (NNs) play a key role in the analytical solving and heuristic determination of propositional logical statements. This project aims to implement a Recurrent Neural Network (RNN), a subset of artificial neural networks where node connections are arranged in the form of a temporal sequence and either a directed or undirected graph, to solve Boolean SATs and report the findings and further improvement suggestions.

1 Introduction

SAT solvers and neural networks are two of computer science's greatest breakthroughs. Both strategies have become widely useful techniques that have significantly improved a wide range of real-world issues. In computer vision, natural language processing, and robotics, neural networks are frequently used to recognize objects in images [1], convert spoken word to written text [2], translate between natural languages [3], control robotic limbs [4], and handle numerous other problems. SAT solvers are frequently used to establish long-standing conjectures in pure mathematics, manage network security protocols, check many various sorts of correctness properties of both hardware and software, and solve challenging planning and scheduling problems [5] emerging from a variety of domains.

2 Background Information

2.1 Recurrent Neural Networks (RNNs)

Recurrent neural networks (RNNs) [6, 7] are neural networks that operate on arbitrarily long sequences of inputs that all have the same size. A finite sequence of input vectors $\{x_i \in \mathbb{R}^{d_{in}}\}$ (for a fixed d_{in}; d_{in} = dimensions of the input vector) is given to an RNN,

which then outputs a vector $\{y \in \mathbb{R}^{d_{out}}\}$ (for a fixed d_{out}; d_{out} = dimensions of the output vector) that is meant to contain data about the entire input sequence. The output y is referred to as the input sequence's embedding.

An RNN is parameterized by a neural network $N \in \mathbb{R}^{d_{in}+d_{out}} \to \mathbb{R}^{d_{out}}$. It computes y from the input sequence $\{x_i\}$ as follows. First it initializes a hidden-state vector $h \in \mathbb{R}^{d_{out}}$, usually to all zeros. Then it traverses the input sequence $\{x_i\}$, each time updating the hidden-state values h:

$$h \leftarrow N(h, x_i)$$

After traversing the input sequence, it returns the final value of its hidden-state h. The following is a pictorial representation [8] of the same that shows the flow of inputs through the neural network.

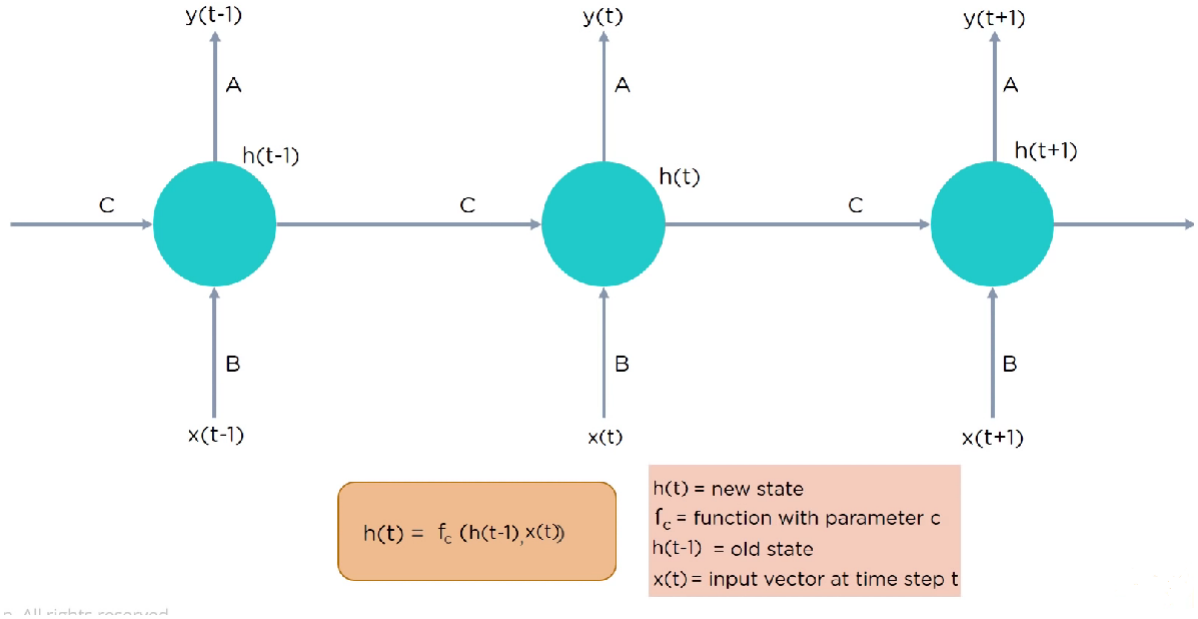


Figure 1: RNN workflow

Here $x(t)$ input x at the time interval t and $y(t)$ represents the output y and $h(t)$ is the hidden state value that is used for the next input $x(t+1)$.

2.2 Conjunctive Normal Form (CNF)

Conjunctive Normal Form (CNF) [9] is a conjunction of one or more clauses, where a clause is a disjunction of literals; otherwise put, it is a product of sums or an AND of ORs. As a canonical normal form, it is useful in automated theorem proving and circuit theory. All conjunctions of literals and all disjunctions of literals are in CNF, as they can be seen as conjunctions of one-literal clauses and conjunctions of a single clause, respectively. Similar to the disjunctive normal form (DNF), a formula in CNF can only

contain the propositional connectives and, or, and not. Only a propositional variable or a predicate symbol can be used before the not operator because it can only be used as part of a literal. An example of conjunctive normal form is as follows:

All of the following formulas in the variables A, B, C, D, E and F are in conjunctive normal form:

$$(A \vee \neg B \vee \neg C) \wedge (\neg D \vee E \vee F)$$
$$(A) \wedge (B \vee D) \wedge (B \vee E)$$
$$(\neg B) \wedge (C)$$

Here the symbols $\vee$ is the disjunction, $\wedge$ is the conjunction, and $\neg$ is a negation of the literal that it is tagged to.

2.3 Goal of the Project

Using one-hot encoding of the CNFs, we aim to feed it to the RNN to train it. The Temporal dynamic behaviour of the RNN allows them to use their internal state (memory) which helps them to train the model effectively using current input and hidden state data from the previous inputs. This Neural Network will be further used to determine the satisfiability of a test dataset to determine the accuracy of the trained model.

3 Related Work

3.1 SAT Solver algorithms

There have been many significant SAT solvers and they often begin by converting a formula to a CNF. They are often based on core algorithms such as the DPLL algorithm [10], but incorporate a number of extensions and features.

DPLL Solvers: Davis–Putnam–Logemann–Loveland algorithm ("DPLL") [11] is a SAT solver employs a systematic backtracking search procedure to explore the space of variable assignments that satisfy the given formula. The basic backtracking algorithm works by selecting a literal, giving it a truth value, then simplifying the formula. If the simplified formula is satisfiable, the original formula is satisfiable; otherwise, the same recursive check is performed assuming the opposite truth value. The problem is divided into two easier sub-problems using this technique, which is known as the splitting rule. The simplification step essentially eliminates from the formula all literals that become false and all sentences that become true under the assignment.

The DPLL algorithm enhances over the backtracking algorithm by the eager use of the rules Unit propagation (If a clause has only a single literal it is set to True always) and Pure literal elimination (If a propositional variable occurs with only one polarity in the formula, it is called pure. A pure literal can always be allocated in a fashion that ensures the truth of every clause that includes it. As a result, when it is assigned in this manner, these clauses no longer restrict the search and can be removed.) at each step. A partial

assignment is considered unsatisfiable if one of its clauses becomes empty, or if all of its variables have been assigned in a way that renders their corresponding literals untrue. When all variables are assigned without producing the empty clause or, in more recent implementations, when all clauses are satisfied, the formula is said to be satisfied. The full formula's unsatisfiability can only be found after a thorough search.

CDCL Algorithm: Conflict-Driven clause learning (CDCL) [12] is an algorithm which has it's inner workings inspired by DPLL Solvers but in one major area. The back jumping in CDCL is non-chronological. There are two main rules followed in this algorithm. The Unit clause rule which says that one literal or variable must be True for a sentence to be satisfied if all but one of its literals or variables are evaluated as False. The Resolution logic is a rule of inference leading to a refutation complete theorem-proving technique for sentences in propositional logic and first-order logic. The logical steps of this algorithm are similar to the DPLL algorithm.

3.2 Significant work on SAT solvers using NN

NeuroComb: Improving SAT Solving using Graph Neural Networks [13]: Mainstream modern SAT solvers are based on the Conflict-Driven Clause Learning (CDCL) algorithm. Recent research sought to improve CDCL SAT solvers by the use of predictions produced by Graph Neural Networks (GNNs) to improve their variable branching heuristics. However, so far this approach either has not made solving more effective, or has required online access to substantial GPU resources. This paper suggests a method called NeuroComb that builds on two insights in order to make GNN improvements practical: (1) predictions of significant variables and clauses can be combined with dynamic branching to create a more efficient hybrid branching strategy; and (2) it is sufficient to query the neural model only once for the predictions prior to the SAT solving. NeuroComb is implemented as an enhancement to a classic CDCL solver called MiniSat and a more recent CDCL solver called Glucose. As a result, with the computational resource requirement of only one GPU, it allowed MiniSat to solve 11% and Glucose 5% more problems on the SATCOMP-2021 competition problem set.

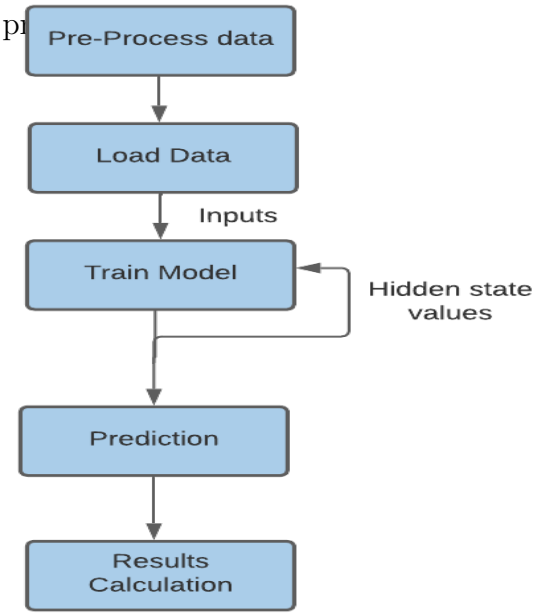
Learning a SAT solver from Single-bit supervision [7]: This paper presented NeuroSAT, a message-passing neural network that simply needs to be trained as a classifier to predict satisfiability before training to solve SAT problems. NeuroSAT can solve problems that are far larger and more challenging than those it encountered during training, despite the fact that it is not competitive with state-of-the-art SAT solvers. It achieves this by simply running for more iterations. In addition, NeuroSAT generalizes to novel distributions; after training exclusively on random SAT problems, it can solve SAT problems encoding graph coloring, clique detection, dominating set, and vertex cover problems during testing, all of which are on a wide range of distributions over small random graphs.

4 Methodology

Key idea of this project is to apply a Recurrent Neural Network (RNN) to the CNF dataset and understand the issues, complexity, performance and feasibility of solving SATs using the NN.

The cnf files are loaded using a python script which are then converted into either CSVs or Avros or a preferred file format (the choice of the file format in this implementation of RNN is explained later sections) that contains the CNF in the text format.

The following figure explains the main flow of the pr



- 1: Pre-process the data and create the files in required file formats
- 2: Load the data using Pandas dataframes
- 3: Train the RNN model by passing in the inputs.
- 4: The training is continued by passing the current input along with the hidden state data from the previous input.
- 5: Once the model is trained, use it for prediction.
- 6: Calculate metrics like Accuracy, Precision, etc., using the predictions.

4.1 Data

For this project, We have found multiple resources for CNF files to be used for the Neural Network. In order to provide developers worldwide with an accessible set for their use, datasets are publicly run servers by institutions of the academic or business sort. They are specifically for SAT solver based engineering. There are different sections of CNF statements that were generated for different problem sets. The following are a few of them [14, 15]:

- AIM: Artificially generated Random-3-SAT
- JNH: Random SAT instances with variable length clauses
- Uniform Random-3-SAT, phase transition region

- Miroslav Velev's SAT Benchmarks: This dataset contains different CNFs that are problem statements derived from formal verification of VLIW processors.

All the datasets contains '.cnf' files that contain CNF formulae encoded in DIMACS CNF format.

4.1.1 DIMACS Encoding

The Center of Discrete Mathematics and Computer Science (DIMACS) [16] is a textual representation of a formula in conjunctive normal form. It uses positive integers to represent variables and their negation to represent the corresponding negated variable.

DIMACS CNF is a textual format. Any line that begins with the character c is considered a comment. Some other parsers require comments to start with c and/or support comments only at the beginning of a file. Varisat supports them anywhere in the file.

A DIMACS file begins with a header line of the form $p\ \ cnf < variables > < clauses >$. Where $< variables >$ and $< clauses >$ are replaced with decimal numbers indicating the number of variables and clauses in the formula. Following the header line are the clauses of the formula. The clauses are encoded as a sequence of decimal numbers separated by spaces and newlines. For each clause the contained literals are listed followed by a 0.

The following is an example:

```
p cnf 3 2
1 2 -3 0
-2 3 0
```

Figure 2: DIMACS format of the formula $(1 \vee 2 \vee \neg 3) \wedge (\neg 2 \vee 3)$

The satisfiability is listed in the documentation of the corresponding sources. This is used for generating labels and training the NN.

4.2 Implementation

4.2.1 Pre-processing

We have written a script that reads all the '.cnf' files by iterating over a list of directories and then converts each of the file into a CNF string and writes into the files. Each entry in the file would be the CNF string and the corresponding label of that CNF separated by a comma. We have considered 1 to be the label of Satisfiable CNF and 0 to be the label of Unsatisfiable CNF.

For example, the cnf file has the following content (Considering the CNF to be satisfiable)
Initially, we have started with the end-file format as CSVs, but since there is a data restriction that each cell of a CSV can contain a maximum of 32,767 characters, it could not accommodate larger CNFs which is common in our dataset. So we tried different

```
1   p cnf 3 2
2   1 2 -3 0
3   -2 3 0
```

Figure 3: Output: $(1 \vee 2 \vee \neg 3) \wedge (\neg 2 \vee 3)$, 1

file types like feather, parquet, and finally settled on using the Avro file format as it has faster read and write times [17] and this file type also has data compression which results in smaller end-data files.

4.2.2 Neural Network Implementation

Once the data is loaded into Dataframes of the Pandas library, the rows of the data are shuffled in place and then the dataframe is split into Training and Testing data. We am using the standard 80-20 split for this implementation.

4.2.2.1 Utility methods

We have also created utility method that are used through out the implementation.

1. $read\_data(dataframe)$: takes a dataframe as an input and converts the data into a list of tuples of the form $(cnf,\ label)$. This is called before the data is fed into the NN and also when the model is used for prediction.

2. $random\_training\_pair(pairs)$: which takes the list of tuples generated by the method $read\_data(dataframe)$, and return a random tuple (pair) picked from it.

3. $line\_to\_tensor(line)$: takes a character input and returns the one hot encoded vector of the character.

4. $label\_to\_tensor(label)$: takes a numeric label and returns a tensor array of it.

5. $category\_from\_output(output)$: takes the output tensor array and gives us the label from the output.

4.2.2.2 RNN class

We have created a class named $RNNClassify$ that methods associated with the neural network like forward propagation, hidden state initialization and the neural network initialization.

1. $\_init\_$: It is called when an instance of the $RNNClassify$ is created, which initializes the network with an input layer, a hidden layer and also assigns the activation function layer for the output.

2. $forward$: This method concatenates the input vector and the hidden state vector to be passed as an input for the input layer and the hidden layers.

3. $init\_hidden$: This method initializes the hidden state values with zeros at the beginning of the training.

4.2.2.3 Training methodology

We have created a method named $train_step$ that handles the training the neural network for a single iteration. This method initializes the hidden state values, trains the model on each character which generates and output and also gives back a hidden state. This hidden state is fed back to the model along with the current input character. Once the entire string is passed through the model, the loss is calculated and then propagated backwards using the '$.backward()$' method that gives us the gradients. Then the parameters of the optimizer are adjusted using the '$.step()$' and '$.zero\_grad()$' methods respectively.

We run the $train\_step$ for n_iters times and the accumulated loss is logged on to the console after every $print\_every$ iterations. Also the loss is stored in a list, to be plotted at the end of the training, for every $plot\_every$ iterations. This entire n_iters iterations of training is done for n_epochs times, where the learning rate of the optimizer is adjusted in each epoch using learning rate schedulers.

The following figure shows the flow of the neural network:

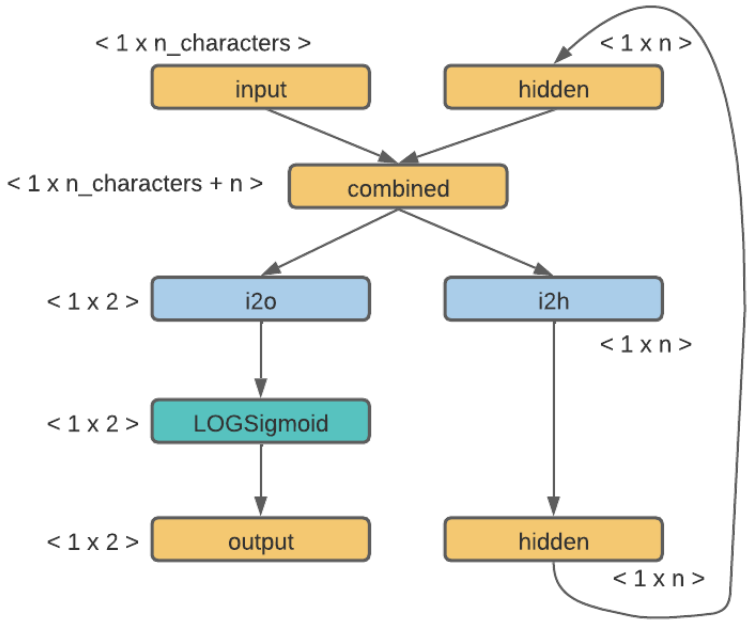


Figure 4: Flow of the input through the Network

Here the $i2o$ is the component where the input is converted to a midway output, which is then passed on to the activation function to get the output of the current input. $i2h$ is the component that converts the input into a hidden state value to be used in concatenation with the next character of that input.

Once the training is done, a plot of loss value through out the iterations are plotted on a graph and display after all the logs.

Here is a screenshot of how the log output looks like:

```
Epoch: 1 - 100 20% (1m 24s) 0.6925
Epoch: 1 - 200 40% (3m 11s) 0.6752
Epoch: 1 - 300 60% (4m 57s) 0.6740
Epoch: 1 - 400 80% (6m 40s) 0.7252
Epoch: 1 - 500 100% (8m 19s) 0.6417
Epoch: 2 - 100 20% (9m 56s) 0.7068
Epoch: 2 - 200 40% (11m 26s) 0.6231
Epoch: 2 - 300 60% (13m 13s) 0.6828
```

Figure 5: Output log during training

Here is a screenshot of the plot loss values collected over the training period.

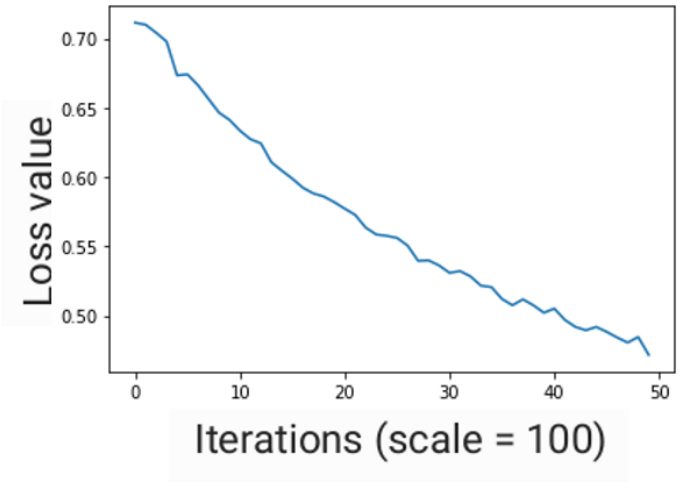


Figure 6: Plot of Loss value during training

4.2.2.4 Example using workflow

Consider a sample CNF statement ($1 \vee 2$) $\wedge$ ($\neg 2$). Our alphabet consists of the following letters "()0123456789 $- \vee \wedge$", so the first character "(" is one-hot encoded as $[1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]$. The length of this one-hot encoded vector will be 15 characters, as the number of characters in our vocabulary is 15. This is combined with the hidden state's initial vector $[0., 0., 0., \ ... \ 0., 0.]$ (the length of this vector is 64 in our case). Now these are concatenated and combined into a vector of length 79 (i.e., 64 + 15). Then this vector is passed onto the layers $i2o$ and $i2h$.

The layer $i2o$ converts this combined vector into a vector of the output dimensions i.e., 2. So the output vector will be converted into $[0., 1.]$. This output can be ignored and only the output at the final character of the CNF statement is considered. The layer $i2h$ converts the combined vector into a vector of the hidden layer size dimension (64 in our case). So hidden vector outputted by this would be $[-0.0094, -0.1123, -0.0028, 0.0120, \ ... \ , 0.0619,$

$-0.0542, 0.0050]$. Now this is used in combination with the one-hot encoded vector of the next character of the CNF i.e., 1. This combined vector is again passed through the neuron layers and the process goes on until all the characters in the CNF statement are exhausted. The final output returned by the layer $i2o$ will be then sent on to the activation function and that is considered as the label of the statement.

4.2.3 Parameters of the Neural network

All the parameters used for the neural network are from the *torch* component of the *PyTorch* library, a widely used open source machine learning framework.

1. **Optimizer:** We have used the Stochastic Gradient Descent (SGD) [18] Optimizer for this implementation. In each iteration of training this optimizer calculates the gradient descent on one data point that is chosen at random. This greatly reduces the computational power and achieves faster iterations in trade for lower convergence rate. For this implementation we have set the learning rate to be 0.0002.

2. **Loss Function:** We have used Negative Log Likelihood as the loss function for this implementation. It gives us the idea of how bad the model is performing and this loss needs to be reduced in order to get the best model parameters.

3. **Activation function:** We have used *LOGSigmoid* as the activation function here. It is a log of sigmoid function which is a very commonly used activation function in case of binary classification. The function is as follows:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

   The fact that the sigmoid function is monotonic, continuous and differentiable everywhere, coupled with the property that its derivative can be expressed in terms of itself, makes it easy to derive the update equations for learning the weights in a neural network when using back propagation algorithm.

4. **Learning rate Schedulers: [19]** We have chained tow learning rate schedulers for better approximation to the best possible model values. The first scheduler we have used is *ExponentialLR* which decays the learning rate of each parameter group by gamma every epoch. The formula for change in learning rate is as follows:

$$lr_{epoch} = Gamma * lr_{epoch\ -\ 1}$$

   where lr_{epoch} is the learning rate in the current epoch.

   Then we have chained this scheduler with $MultiStepLR$ scheduler which decays the learning rate of each parameter group by gamma once the number of epoch reaches one of the milestones. The formula for change in learning rate is as follows:

$$lr_{epoch} = \begin{cases} Gamma * lr_{epoch\ -\ 1} & if\ epoch\ in\ [milestones] \\ lr_{epoch} - 1 & otherwise \end{cases}$$

4.2.4 Prediction

Once the model is trained, the model state is saved using the *.save*() method of the torch component. This model is then used for prediction on the test dataset that was split from the original dataset earlier in the process. We have written a method named *evaluate* that takes in a CNF statement and the model, runs the model on the statement and gives us the label of the statement. The method *calculate_accuracy* calls the *evaluate* method on all the statements in the test dataset and calculates the accuracy, precision and recall of the predictions done by the model.

5 Results

We have run this code for multiple times altering the number of iterations, number of epochs, different activation functions etc., Here is a list of few of the training and prediction runs we have done as part of this implementation:

Activation function	n_epochs	n_iters	learning rate Schedulers	Accuracy
LOGSoftmax	10	1000	None (default value)	46.83
LOGSoftmax	15	1000	None (default value)	51.72
LOGSoftmax	10	500	$ExponentialLR$, $MultiStepLR$	53.96
LOGSigmoid	10	500	None (default value)	52.44
LOGSigmoid	10	500	$ExponentialLR$, $MultiStepLR$	54.62

Table 1: Results of RNN runs with different parameters

All the runs were performed on a dataset consisting of 5921 CNF statements, with the number of literals ranging from 20 to 250 and number of clauses ranging from 80 to 1065. Most of the CNF files contain variable length clauses. Each run takes around 60 - 90 minutes of time depending upon the number of epochs and the number of iterations in each epoch. Initially, as we was having a fixed learning rate for my SGD optimizer, the accuracy was not satisfactory. After adding the learning rate schedulers, the accuracy has increased considerably. Later we moved on to LOGSigmoid function, from LOGSigmoid, as it is a widely used activation function in case of binary classification. We have noticed that with the use of the said activation function and the learning rate schedulers, the accuracy of the model has increase considerably.

The Python script that contains the code for pre-processing the data, the dataset used for this implementation and the Jupyter notebook that contains the implementation of the neural network can be found at the following link: GitHub

6 Other Approaches and Limitations

Initially, we was referring to a text classification implementation created by Tensorflow [20], an end-to-end open source platform for machine learning. The example contained a sentiment analysis (a binary classification) of the IMDB large movie review dataset [21]. The data was directly loaded from the Tensorflow Datasets, and it was loaded as a Tensor element. It is then split into training and test datasets a vocabulary of the most frequent words of the train dataset was created. This was later used for encoding the input data into the neural network.

There were a couple of issues with this implementation. Initially, my main hurdle was to figure out a way to convert the CNF data into a neural network ready format. To adjust to the text classification network, we have decided to convert the CNF files into a CNF statement and store it along with it's label separated by a comma in a CSV file. The first issue with the Tensor flow example was the conversion of Pandas Dataframes to TensorSpec elements. It is a datatype specifically created to work with Tensor API functionalities. once we figured that out, the next hurdle was the encoding. The example had a dictionary of frequent words created for encoding. Although, this might work in language like English, where a common words that occur in a sentence determine the sentiment of the sentence. But this is not applicable in our dataset because, even though we have literal names (numbers in our case) that repeat across multiple CNF statements, we cannot consider them to have the same impact on each of the statements, hence we had to resort to the regular one-hot encoding. Doing the one-hot encoding made each input very long and as a result we had to change the dataset.

Working with datasets that had CNFs containing around 1000000 literals and 4000000 clauses was very computation heavy and My PC couldn't load that on to the RAM. So we created a custom CSV file with around 100 CNF statements and labels to make an RNN implementation work and then adjust the dataset to make it work. Later, We started working on the implementation of Vanilla RNN and came to the current implementation of the Neural Network and ran the code against my sample dataset. The code ran with a few errors, which were fixed on the go and then we had to figure out a way to run with actual CNF files.

We later came across the limitation of CSV, that it allowed a maximum of 32,767 characters per cell, which did not support the larger CNFs in our dataset. Then we tried different file formats as explained in the pre-processing section of the implementation and settled on Avro file format. Although, this file type could hold the larger data we are dealing with, We could not load the dataset onto my RAM to feed it to the neural network. So we have chosen the smaller CNF files among the data set we had and have grouped around 5900 of such files to be used for training and testing.

Later, we had to tweak the parameters of the NN, to achieve better accuracy, so, We tried across different activation functions, iterations, epochs and learning rates of the Optimizer and came to the final results we reported here.

7 Conclusion

This Implementation of SAT solver so far has shown considerable results. We would suggest researching a better way of encoding larger datasets in our case. A better encoding approach along with a better RAM capability would help us run the current implementation with bigger datasets. We feel like the number of samples in our dataset could be far more than what we currently worked on, to achieve better reliable models and an accurate model that we can use to conclude the effectiveness of RNN models to solve the satisfiability of the CNF statements.

We would also suggest implementation of the NN using Long Short-Term Memory (LSTM) [22], a special kind of recurrent neural network (RNN) which is formed by adding long and short term memory units in recurrent neural network(RNN). When training a vanilla RNN using back-propagation, the long-term gradients which are back-propagated can "vanish" (that is, they can tend to zero) or "explode" (that is, they can tend to infinity), because of the computations involved in the process, which use finite-precision numbers. RNNs using LSTM units partially solve the vanishing gradient problem, because LSTM units allow gradients to also flow unchanged.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012, p. 1097–1105. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[2] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[3] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, Eds., vol. 27. Curran Associates, Inc., 2014. [Online]. Available: https://proceedings.neurips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf

[4] I. Lenz, H. Lee, and A. Saxena, "Deep learning for detecting robotic grasps," *The International Journal of Robotics Research*, vol. 34, no. 4-5, pp. 705–724, 2015. [Online]. Available: https://doi.org/10.1177/0278364914549607

[5] J. Coelho and M. Vanhoucke, "Multi-mode resource-constrained project scheduling using rcpsp and sat solvers," *European Journal of Operational Research*, vol. 213, no. 1, pp. 73–82, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S037722171100230X

[6] "En.wikipedia.org. 2022. Recurrent neural network." [Online]. Available: https://en.wikipedia.org/wiki/Recurrent_neural_network

[7] D. Selsam, "Neural networks and the satisfiability problem," 2019. [Online]. Available: https://stacks.stanford.edu/file/druid:jt562cf4590/dselsam_dissertation_final-augmented.pdf

[8] "Recurrent Neural Network (RNN) — Simplilearn." [Online]. Available: https://www.youtube.com/watch?v=lWkFhVq9-nc

[9] "En.wikipedia.org. 2022. Conjunctive Normal Form." [Online]. Available: https://en.wikipedia.org/wiki/Conjunctive_normal_form

[10] "En.wikipedia.org. 2022. SAT Solvers." [Online]. Available: https://en.wikipedia.org/wiki/SAT_solver

[11] "En.wikipedia.org. 2022. DPLL Algorithm." [Online]. Available: https://en.wikipedia.org/wiki/DPLL_algorithm

[12] "En.wikipedia.org. 2022. CDCL Algorithm." [Online]. Available: https://en.wikipedia.org/wiki/Conflict-driven_clause_learning

[13] W. Wang, Y. Hu, M. Tiwari, S. Khurshid, K. L. McMillan, and R. Miikkulainen, "Neurocomb: Improving SAT solving with graph neural networks," *CoRR*, vol. abs/2110.14053, 2021. [Online]. Available: https://arxiv.org/abs/2110.14053

[14] "SATLIB - Benchmark Problems." [Online]. Available: https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

[15] "Miroslav Velev's SAT Benchmarks." [Online]. Available: http://www.miroslav-velev.com/sat_benchmarks.html

[16] "DIMACS CNF." [Online]. Available: https://jix.github.io/varisat/manual/0.2.0/formats/dimacs.html

[17] "Stop Using CSVs for Storage — Here Are the Top 5 Alternatives." [Online]. Available: https://towardsdatascience.com/stop-using-csvs-for-storage-here-are-the-top-5-alternatives-e3a7c9018de0

[18] "En.wikipedia.org. 2022. Stochastic Gradient Descent." [Online]. Available: https://en.wikipedia.org/wiki/Stochastic_gradient_descent

[19] "Guide to Pytorch Learning Rate Scheduling." [Online]. Available: https://www.kaggle.com/code/isbhargav/guide-to-pytorch-learning-rate-scheduling/notebook

[20] "Text classification with an RNN." [Online]. Available: https://www.tensorflow.org/text/tutorials/text_classification_rnn

[21] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, pp. 142–150. [Online]. Available: http://www.aclweb.org/anthology/P11-1015

[22] "En.wikipedia.org. 2022. Long short-term memory." [Online]. Available: https://en.wikipedia.org/wiki/Long_short-term_memory