TOP-DOWN NEURAL MODEL FOR FORMULAE

Anonymous authors
Paper under double-blind review

ABSTRACT

We present a simple neural model that given a formula and a property tries to answer the question whether the formula has the given property, for example whether a propositional formula is always true. A structure of formula is captured by a feedforward neural network build recursively for the given formula in a top-down manner. The results of this network are then processed by two recurrent neural networks. One of the interesting aspects of our model is how propositional atoms are treated. The model is insensitive to the renaming of atoms and to the number of distinct atoms occurring in the formula.

1 INTRODUCTION

In real-world situations a very successful approach, popularized in Kahneman (2011), to problem solving is based on a clever combination of fast instinctive (heuristic) reasoning and slow logical reasoning. The latter is exemplified by abstract logical formulae where only structural properties matter. If computers are involved, a logical formula is traditionally a syntactic object which is a subject to simple but very fast syntactic manipulations Robinson & Voronkov (2001). Hence all but very basic decisions are postponed if possible. However, this viewpoint is rapidly changing as various AI methods are tested in the field of automated reasoning, in particular machine learning methods.

A fundamental problem in using machine learning in automated reasoning is a suitable representation of logical formulae. A formula as a solely syntactic object is no longer sufficient and we have to exploit its semantic properties. Various approaches have been proposed for different logical systems. In this paper we will concentrate on the simplest yet very powerful standard logical system—classical (Boolean) propositional logic.

This paper presents, as far as we know, a novel neural representation of propositional formulae that makes it possible to test whether a given formula has a given property, e.g., whether the formula is always true or not. Clearly, we try to solve a well-known CONP-complete problem. However, the fact that the problem is generally hard and requires a non-trivial search does not rule out the possibility that a decent heuristic can be learned, moreover, if only a specific subset of formulae is involved. In particular, our general goal is to obtain a useful heuristic that can help us in guiding a proof search, where we typically face numerous choice points.

Unlike in natural language processing, a parse tree for a formula is available for free. Although some approaches do not exploit this feature and try to learn the structure of a formula on their own, using usually various recurrent neural networks (RNN), it is more common to take advantage of this knowledge. Moreover, it seems that the later approach has a significant edge, see Evans et al. (2018). Usually propositional atoms, basic building blocks of propositional formulae, are learned as embeddings and each logical connective is treated as a unique neural network that given the vector representation of its arguments produces a vector that represents an application of the connective on these arguments, e.g., a binary connective takes two vectors of length d, and produces a new one of length d, see Allamanis et al. (2017). This clearly leads to tree recursive neural networks Socher et al. (2012) where the structure of the network follows the parse tree. Such models are built bottom-up and the meaning of the formula is usually the vector produced in the root of the tree.

Our model also uses the parse tree of formula, but the knowledge is propagated in the opposite direction. We start with a vector (random or learned), representing a property we want to test, and we propagate it from the root to leaves (propositional atoms). The knowledge propagated to atoms is then processed by recurrent neural networks and a decision is produced. This makes it possible to

ignore completely the names of propositional atoms and concentrate more on structural properties of formulae.

The experimental results suggest that the model is more than competitive and beats other known approaches on some benchmarks. More importantly, our model seems to suffer less if bigger formulae are involved and could be more useful in real world scenarios.

The structure of this paper is as follows. In Section 2 we discuss the architecture of our model in full details and also a dataset on which we will experiment is introduced there. In Section 3 we discuss an implementation of building blocks of our network and present experimental data. Some possible future modifications are briefly mentioned in Section 4. Few relevant models are mentioned in Section 5 and the paper concludes with Section 6.

2 MODEL

Our model tries to mimic the following approach sometimes used by humans to decide whether a propositional formula is always true (tautology). If we want to know whether, e.g., $(p \to q) \vee (q \to p)$ is a tautology[1] we can suppose the contrary and try to produce a truth-value assignment such that the whole formula is false. It means both $p \to q$ and $q \to p$ are false under such an assignment[2]. It follows from the former that p is true and q is false, but from the latter we obtain that q is true and p is false. Because such an assignment is impossible, we showed that the formula cannot be false and hence it is always true.

In a nutshell, we try to propagate a truth value assigned to the whole formula, in our case the formula being false, to its subformulae and we repeat this process until we reach propositional atoms. This gives us a set of constraints on possible truth-value assignments to atoms. At this point we check whether a truth-value assignment satisfying these constraints exists and thus we answer the original question.

However, it is pretty clear that usually this approach is not so straightforward. For example, if we want a formula of the form $A \to B$ to be true, it means that the subformula A is false or the subformula B is true. Moreover, such choices easily accumulate with the increasing depth[3] of formula.

Still we will show that a neural model based on the above mentioned idea is competitive. Similarly to a standard recursive approach we also represent a formula by a recursively built model based on the parse tree of formula. However, unlike there we use a top-down approach. In our case, we start with a vector $\boldsymbol{w} \in \mathbb{R}^d$ (random or learned), where d is a parameter of the model, that roughly represents a property of formulae we want to check.[4] We continue by propagating this knowledge to subformulae in a top-down manner, e.g., a binary connective $\circ$ takes a vector $\boldsymbol{v} \in \mathbb{R}^d$ (input) and produces two new vectors $\boldsymbol{v}_1 \in \mathbb{R}^d$ and $\boldsymbol{v}_2 \in \mathbb{R}^d$ (outputs), hence $\circ\colon \mathbb{R}^n \to \mathbb{R}^n \times \mathbb{R}^n$. Since all the vector have the same length d, it is possible to connect individual building blocks together and produce a recursive neural network.

The basic building blocks of the tree part of our model are

$\boldsymbol{w} \in \mathbb{R}^d$	an initial vector,
$c_i\colon \mathbb{R}^d \to \underbrace{\mathbb{R}^d \times \cdots \times \mathbb{R}^d}_{k\text{-times}}$	a neural network that represents a k-ary logical connective i,

for all logical connectives i in our language[5].

From these components, which are shared among all formulae in a given language, we dynamically build a recursive network for a particular formula that corresponds to the parse tree of formula—the

[1] A formula that is true under any assignment of truth values to propositional atoms.

[2] For simplicity, from now on we assume that all the claims are under a given assignment.

[3] The depth of formula is the depth of its parse tree.

[4] However, it does not mean that by simply changing $\boldsymbol{w}$ our model starts to test a different property, other parts of the model have to be trained accordingly, see also footnote 8.

[5] Note that even nullary connectives called constants are allowed.

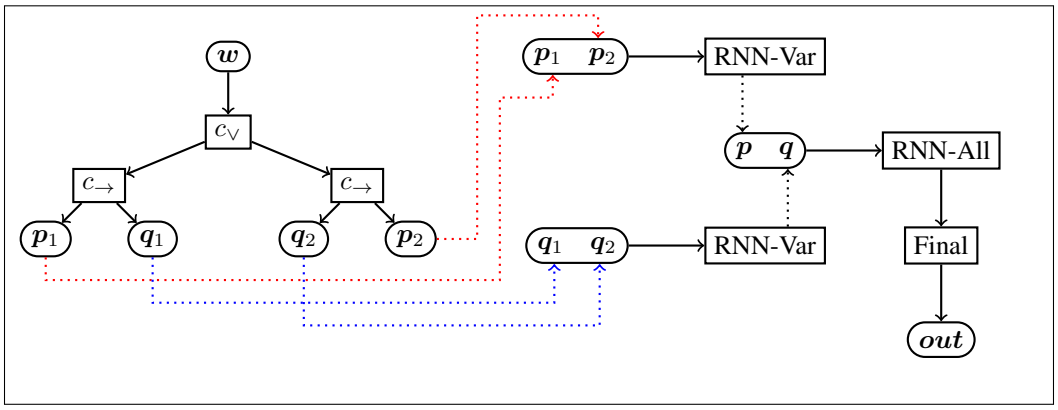

Figure 1: An example of our model for formula $F = (p \rightarrow q) \vee (q \rightarrow p)$. The initial vector $\boldsymbol{w}$ is propagated through the tree recursive network part producing vectors $\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{q}_1$, and $\boldsymbol{q}_2$, where $\boldsymbol{p}_1$ corresponds to the first occurrence of atom p in F. Vectors corresponding to the same atom are processed by RNN-Var and these results are then processed by RNN-All. A final output $\boldsymbol{out}$ is produced by Final using the output of RNN-All.

parse tree tells us how to connect individual components together. For example in the left part of Figure 1 we see a model for $F = (p \rightarrow q) \vee (q \rightarrow p)$, which is directly obtained from the parse tree. The vector $\boldsymbol{w}$ is an input to the root of the tree and all the connectives in it are replaced by neural nets c_i that represent them. Atoms contain propagated outputs (vectors) of this tree recursive model, here we obtain vectors $\boldsymbol{p}_1, \boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{p}_2 \in \mathbb{R}^d$, where $\boldsymbol{p}_1$ is at the position of the first occurrence of p in F.

Note that each occurrence of the same atom in our tree model produces a unique vector. The fact that they correspond to the same atom is exploited by another level of our model. We take a recurrent neural network RNN-Var that has a sequence of all vectors corresponding to the same atom in the formula as an input and outputs a vector in $\mathbb{R}^d$. In Figure 1 RNN-Var takes $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ and outputs $\boldsymbol{p} \in \mathbb{R}^d$ and also the same RNN-Var takes $\boldsymbol{q}_1$ and $\boldsymbol{q}_2$ and outputs $\boldsymbol{q} \in \mathbb{R}^d$.

One more recurrent neural network RNN-All takes the sequence of all output vectors produced by RNN-Var and outputs a vector in $\mathbb{R}^d$ that is input to a component Final: $\mathbb{R}^d \rightarrow \mathbb{R}^2$ which ends with a softmax layer and produces $\boldsymbol{out} \in \mathbb{R}^2$ that makes it easy to decide whether the output of whole network is true or false. It should be emphasized that our model completely ignores the names of atoms, we only check whether they are the same or distinct.

The basic building blocks of the recurrent part of our model are

RNNVar: $\mathbb{R}^d \times \cdots \times \mathbb{R}^d \rightarrow \mathbb{R}^d$	a RNN aggregating vectors corresponding to the same atom,
RNNAll: $\mathbb{R}^d \times \cdots \times \mathbb{R}^d \rightarrow \mathbb{R}^d$	a RNN aggregating the outputs of RNN-Var components,
Final: $\mathbb{R}^d \rightarrow \mathbb{R}^2$	a final decision layer.

It should be emphasized again how the model is built. Given a property we want to test, e.g., whether a formula is always true, we train representations of $\boldsymbol{w}$, c_i, RNN-Var, RNN-All, and Final. These components are shared among all the formulae. For a single formula we produce a model (neural network) recursively from them as described above. For implementation details see Section 3.

2.1 Dataset

To provide comparable results in Section 3 we use the dataset[6] presented in Evans et al. (2018) and thoroughly described therein. For our purposes here it is essential that it contains triples of the form $(A, B, A \models B)$ where A and B are propositinal formulae and $A \models B$ indicates whether B follows

[6]It can be obtained from `https://github.com/deepmind/logical-entailment-dataset`.

from A. The dataset contains train (99876 triples), validation (5000), and various test sets with a different level of difficulty given by a number of atoms and connectives[7] occurring in them.

From the deduction theorem we know that $A \models B$ is equivalent to $A \rightarrow B$ being a tautology. We prefer the later form, because it fits nicely into our model and instead of learning the meaning of entailment ($\models$), we can use the connective $\rightarrow$ directly.

Note that although our model ignores the names of atoms this cannot be exploited directly, because validation and test sets do not contain pairs of formulae that would result from pairs in the training set after renaming atoms.

3 EXPERIMENTS

Although our model is conceptually simple, each building block can be implemented in various ways. In this section we briefly discuss some possible variants and parameters. However, we have not tried to optimize over all the possible parameters discussed later and it is very likely that our results can be improved.

The following implementation of the model introduced in Section 2 is our standard experimental model, called TopDownNet:

- $d = 128$,
- $\boldsymbol{w} \in \mathbb{R}^d$ is a learned vector,
- every c_i is a sequence of linear, ReLU, linear, and ReLU layers, where the input size and output size is always the same with the exception of binary connectives, where the last linear layer is $\mathbb{R}^d \rightarrow \mathbb{R}^{2d}$,
- RNN-Var is a gated recurrent unit (GRU) with 2 recurrent layers and the size of input and hidden state is d,
- RNN-All is a GRU with 1 recurrent layer and the size of input and hidden state is d,
- Final is a sequence of linear ($\mathbb{R}^d \rightarrow \mathbb{R}^{d/2}$), ReLU, linear ($\mathbb{R}^{d/2} \rightarrow \mathbb{R}^2$), and a log softmax,

and we use the mean square error as a loss function and Adam algorithm as an optimizer with the learning rate 10^{-4}.

A key parameter of our model is the dimension d, which is the length of the initial vector $\boldsymbol{w}$ and also the length of most other vectors occurring in our model. Even $d = 8$ produces reasonable results, as Figure 2 shows, but with increasing d the quality of the model grows, see Table 1.

Our various experiments suggest that whether $\boldsymbol{w}$ is chosen randomly or learned makes little to no difference.[8] Similarly and more importantly, connectives c_i composed of a single linear layer seem to perform almost equally, and sometimes even better, to more complicated versions containing non-linearities as in our standard TopDownNet.

In RNN-Var and RNN-All we use gated recurrent units (GRUs) which in our experiments perform similarly to long short-term memory (LSTM) RNNs, Similarly to results obtained in Evans et al. (2018), we have seen no real advantage of using their bidirectional variants for our model and in many cases they produce worse results. The order in which RNN-Var and RNN-All consume their inputs is random.

Although it is conceptually nicer and fits to our original intuition, our combination of two RNNs and their functions clearly suggest a question whether they are in fact necessary. Either we can use the same RNN for RNN-Var and RNN-All, or we can use padding to separate the sequences of different atoms. Our preliminary experiments suggest that both new variants and our original approach perform quite similarly, see Figure 3.

[7] The dataset contains negations ($\neg$), implications ($\rightarrow$), conjunctions ($\wedge$), and disjunctions ($\vee$).

[8] An obvious question is whether it makes sense to even consider a learned $\boldsymbol{w}$. Generally, it could be useful, because we can be in a situation where we want to use more such vectors which are in relations that we want to learn.

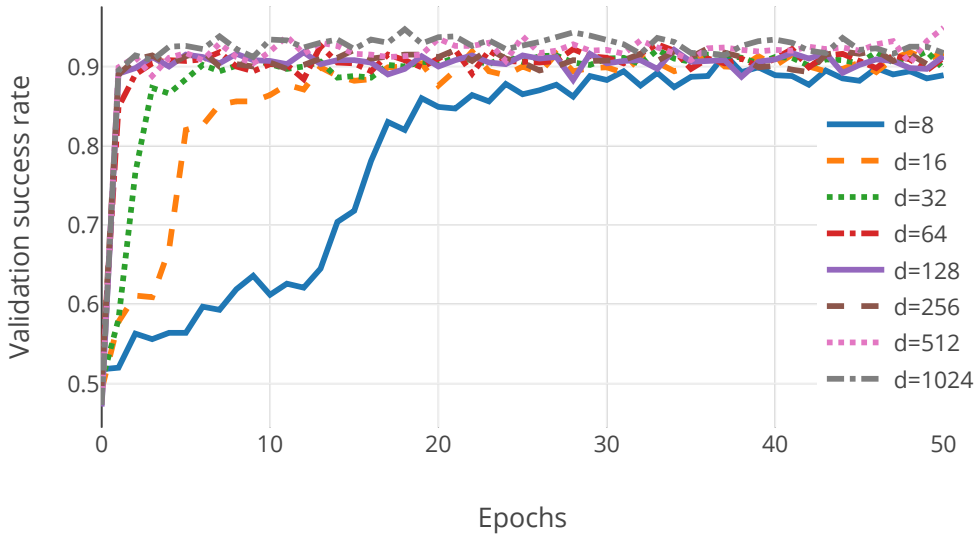

Figure 2: Various choices of d used on a very simple variant of our network with a long short-term memory (LSTM)—every c_i is a linear layer, RNN-Var and RNN-All are LSTMs with one recurrent layer, and Final is a linear layer combined with log softmax.

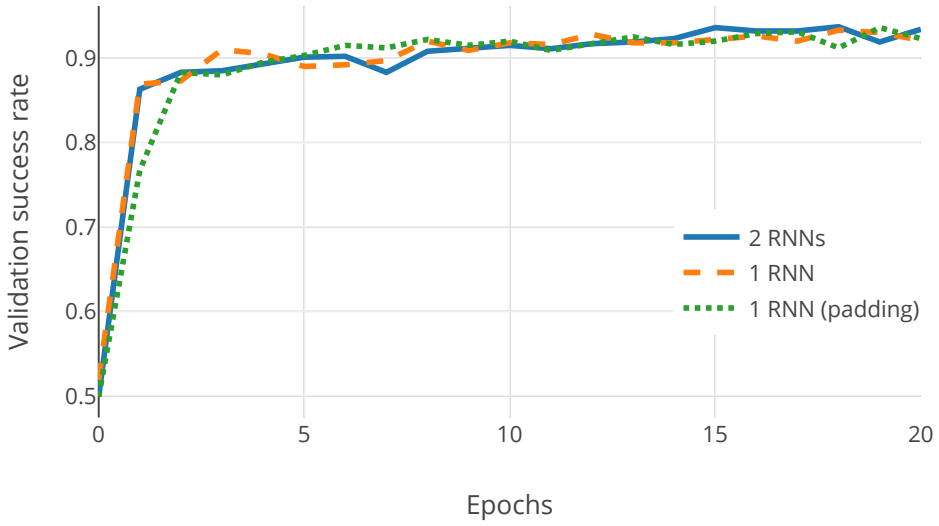

Figure 3: RNN variants where 2 RNNs is TopDownNet ($\boldsymbol{w}$ is random and fixed). 1 RNN uses RNN-Var also as RNN-All and 1 RNN (padding) uses only RNN-Var where the sequences of distinct atoms are separated by padding.

It seems that if the final layer is just a combination of linear and log softmax the performance is slighly better than the one used in TopDownNet with an added non-linearity. However, the best

Table 1: TopDownNet models against other approaches (accuracy)

model	valid	easy	hard	big	massive	exam
TreeNet Encoders	72.7	72.2	69.7	67.9	56.6	85.0
TreeLSTM Encoders	79.1	77.8	74.2	74.2	59.3	75.0
PossibleWorldNet	**98.7**	**98.6**	**96.7**	**93.9**	73.4	**96.0**
TopDownNet ($d = 128$)	94.0	92.8	81.0	80.7	79.7	95.0
TopDownNet ($d = 256$)	95.1	95.2	82.3	80.3	82.4	95.0
TopDownNet ($d = 512$)	95.1	95.3	84.2	83.6	**83.6**	**96.0**
TopDownNet ($d = 1024$)	95.5	95.9	83.2	81.6	**83.6**	**96.0**

performing models that we present in Table 1 use our standard parameters with different values of dimension d.

In Table 1 is a comparison of our models and other approaches mentioned in the benchmark in Evans et al. (2018). The values for the first three models are taken from that paper. TreeNN follows Allamanis et al. (2017) and TreeLSTM follows Tai et al. (2015). PossibleWorldNet is developed in Evans et al. (2018). TopDownNets are our standard models from the beginning of Section 3 with the lowest losses on the validation set for different values of d.

The results in Table 1 show that our model is competitive with other approaches. It beats standard tree recursive models (TreeNet Encoders and TreeLSTM Encoders) and PossibleWorldNet, which performs similar to testing random truth-values, on the massive set and keeps pace on examples from textbooks. The good results on the massive set suggest that our model can do better than just random testing of truth-values. Moreover, it is interesting that the increase of d does not have such a significant role, as already mentioned before.

3.1 Interpretation of model

We present our model as an alternative to bottom-up approaches, where a possible interpretation of such models is that they combine (samples of) random assignments and produce the value of the whole formula. Similarly, we could argue that our model propagates a truth value top-down and makes (sampled) random choices when multiple options are available, e.g., when we want to make $A \rightarrow B$ true. However, this clearly does not explain better results of our model, because for example on the massive set from Table 1 random assignments are much more successful than random top-down propagations.

Our model therefore has to take advantage of something else. Clearly, it uses training data more efficiently, because it is insensitive to the names of atoms, but this is not enough to explain our results. However, it is possible that the top-down propagation captures the overall structure of the formula better and enables a more efficient transfer of knowledge through formulae.

All formulae in the dataset are of the form $A \rightarrow B$, see Section 2.1, hence propagating $\boldsymbol{w}$, which is supposed to mean false, through $\rightarrow$, denoted $c_{\rightarrow}(\boldsymbol{w})$, produces two vectors $\boldsymbol{v}_{\rightarrow 1}$ and $\boldsymbol{v}_{\rightarrow 2}$ that correspond to A and B, respectively. These two vectors are commonly very distinct and applying negation ($c_{\neg}$) on one of them almost exactly produces the other one. Moreover, this remains true even if we apply $c_{\neg}$ repeatedly, see Figure 4. If we now apply $c_{\wedge}$ on $\boldsymbol{v}_{\rightarrow 1}$, then we not so surprisingly obtain two vectors very similar to each other and also similar to $\boldsymbol{v}_{\rightarrow 1}$. More interesting is to apply $c_{\wedge}$ on $\boldsymbol{v}_{\rightarrow 2}$, because we obtain two vectors that are similar, but not as similar as in the previous case and commonly even less similar to $\boldsymbol{v}_{\rightarrow 2}$. This suggests that something more involved than simple splitting of truth values according to possible choices takes place in the model.

Unfortunately, it is very unclear what happens deeper in the model and how recursive neural networks combine the vectors in leaves; moreover, RNN-Var and RNN-All consume their inputs in random order and this slightly influences the results. However, it seems that our model is quite ro-

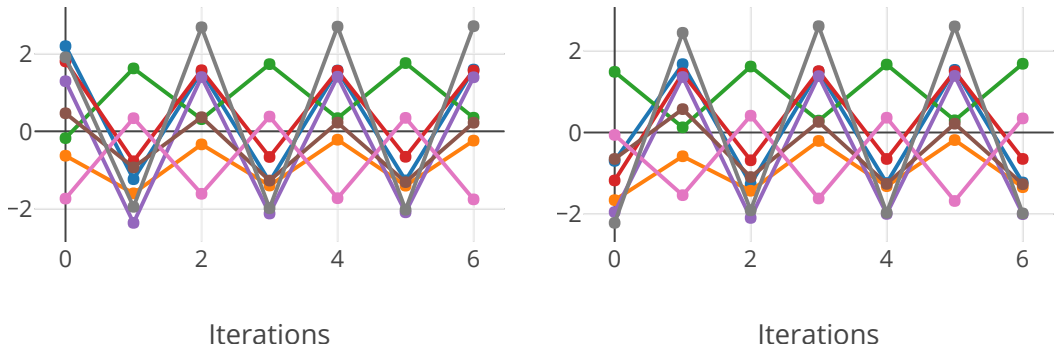

Figure 4: The repeated applications of negations ($c_\neg$) on $v_{\to 1}$ (left) and $v_{\to 2}$ (right), which are the results of $c_\to(\boldsymbol{w})$, for a model where every c_i is a linear layer with $d = 8$ and a learned $\boldsymbol{w}$.

bust to changes in the order of arguments in commutative connectives ($\wedge$ and $\vee$) and to other simple transformations of subformulae—replacing $A \to B$ by $\neg B \to \neg A$, $A \wedge B$ by $\neg(\neg A \vee \neg B)$ etc.

An explanation of very good performance could be that our model just exploits the given dataset in some unforeseen way. In fact, we want to learn regularities in the dataset. Although this is possible, we believe that this does not explain the performance of our model completely. We trained our model on BOOL8 and BOOL10 datasets from Allamanis et al. (2017) adapted[9] to our problem and tested it on unseen equivalence classes with the success rate 98.3% and 80.5%, respectively. These numbers are not directly comparable to the results presented in the paper, however, they suggest that our model is able to perform reasonably even on other datasets. Similarly, if we use the dataset from Evans et al. (2018) to test satisfiability[10], then we obtain results similar to the original dataset.

4 POSSIBLE VARIANTS

We aimed for a simple model and hence a plethora of modifications is possible; they can both improve the quality of produced models and their generality. Here we will discuss at least some of them.

Our model uses a feedforward neural network for representing a formula and only then two recurrent networks are used to process the results of the feedforward part. Clearly, more complicated versions can be produced that provide a better interplay between these two layers and/or the feedforward part can become more complicated, e.g., to allow a communication between different subformulae.[11] For example, it is possible to propagate information also back in the bottom-up direction through the tree part of our model, or use an attention mechanism.

It should be also noted that although we use our model for deciding whether a formula is a tautology, we can try to enforce other properties by a similar process, e.g., we can test whether a formula is satisfiable, i.e., is true under some assignment of truth-values. Also learning more related properties, more vectors like $\boldsymbol{w}$, with all the other components shared could be an interesting problem, and their interplay could even improve the performance on individual tasks.

[9] We produced randomly pairs of equivalent and non-equivalent formulae.

[10] For every formula $A \to B$ we take $\neg(A \to B)$ and propagate negation through the formula. Hence we obtain, e.g., $A \wedge \neg B$ and then continue by propagating negation through B.

[11] An example how a more complicated model can help is Peirce's law $((p \to q) \to p) \to p$, a well-known tautology. Similarly as in Section 2 we try to produce a truth-value assignment such that Peirce's law is false. It means that $(p \to q) \to p$ is true and p is false. Now using that p is false, we know that $p \to q$ is also false and hence p is true and q is false, a contradiction thanks to p. However, without using the fact that p is false (inaccessible to our model) in processing $(p \to q) \to p$ we have to deal with more choices along this line of reasoning.

Clearly, our presentation of the model suggests that it heavily relies on properties of classical (Boolean) propositional logic. Nevertheless, it can be used directly also in a non-classical (propositional) setting. Note that the semantics of a given logic is completely learned from examples and no a priory knowledge is provided, however, it is of course possible that our model is unable to reasonably capture the semantics of a given logic and some modifications, e.g., those mentioned above, are necessary. The question whether a more involved model based on our model can be successfully used also for more complex formulae (first-order or even higher-order) is a bit unclear, a clear goal for a future research.

5 Related work

A thorough recent comparison of related methods is in Evans et al. (2018) and therefore this section is only very brief. In Wang et al. (2017) authors also develop a model which is invariant to atoms (variables) renaming. However, a formula graph obtained from a parse tree is there translated into a vector using the embeddings of nodes given by their neighborhoods. Moreover, the problem they study is different (first-order and higher-order logics). In Zaremba et al. (2014), a pioneering paper on using recursive neural networks Socher et al. (2012) for mathematical expressions, they also do not deal with variables, because they allow at most one to occur in them.

In Allamanis et al. (2017) a bottom-up approach is presented, however, it also contains a restricted form of backward communication that proved to be useful. The problem studied there is learning vector representations of formulae and based on their similarity test their properties. In our case, we test the properties directly.

An approach based solely on recurrent neural networks has been used in natural language processing where textual entailment is an important topic. For LSTM models with an attention mechanism see Wang & Jiang (2016); Rocktäschel et al. (2016).

PossibleWorldNet developed in Evans et al. (2018) addresses the same problem as our model. It exploits a novel idea of learning various useful assignments and test them latter on. However, the results suggest that the method performs similarly to random assignments (a very powerful technique on its own). It could be because the standard bottom-up approach is used.

6 Conclusion

We have presented a novel top-down approach to represent formulae by neural networks and showed some preliminary experiments. They suggest that our approach is competitive with other known approaches and beats them on some benchmarks. More importantly, it seems that the presented model can deal better with the increasing size of formulae than other approaches. The model deals only with the structure of formulae and ignores completely for example names of individual atoms, only whether they are the same or distinct matters.

Acknowledgments

This work was supported by the European Regional Development Fund under the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

References

Miltiadis Allamanis, Pankajan Chanthirasegaran, Pushmeet Kohli, and Charles A. Sutton. Learning continuous semantic representations of symbolic expressions. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pp. 80–88, 2017. URL `http://proceedings.mlr.press/v70/allamanis17a.html`.

Richard Evans, David Saxton, David Amos, Pushmeet Kohli, and Edward Grefenstette. Can neural networks understand logical entailment? In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=SkZxCk-0Z`.

Daniel Kahneman. *Thinking, Fast and Slow*. Farrar, Straus and Giroux, 2011. ISBN 9781429969352.

John Alan Robinson and Andrei Voronkov (eds.). *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9.

Tim Rocktäschel, Edward Grefenstette, Karl Moritz Hermann, Tomas Kocisky, and Phil Blunsom. Reasoning about entailment with neural attention. In *International Conference on Learning Representations (ICLR)*, 2016.

Richard Socher, Brody Huval, Christopher D. Manning, and Andrew Y. Ng. Semantic compositionality through recursive matrix-vector spaces. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, EMNLP-CoNLL 2012, July 12-14, 2012, Jeju Island, Korea*, pp. 1201–1211, 2012. URL http://www.aclweb.org/anthology/D12-1110.

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1556–1566. Association for Computational Linguistics, 2015. doi: 10.3115/v1/P15-1150. URL http://www.aclweb.org/anthology/P15-1150.

Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 30*, pp. 2786–2796. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/6871-premise-selection-for-theorem-proving-by-deep-graph-embedding.pdf.

Shuohang Wang and Jing Jiang. Learning natural language inference with lstm. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 1442–1451. Association for Computational Linguistics, 2016. doi: 10.18653/v1/N16-1170. URL http://www.aclweb.org/anthology/N16-1170.

Wojciech Zaremba, Karol Kurach, and Rob Fergus. Learning to discover efficient mathematical identities. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pp. 1278–1286, 2014. URL http://papers.nips.cc/paper/5350-learning-to-discover-efficient-mathematical-identities.