GENERATING TEMPORAL LOGICAL FORMULAS WITH TRANSFORMER GANS

Anonymous authors
Paper under double-blind review

ABSTRACT

Training neural networks requires large amounts of training data, often not readily available in symbolic reasoning domains. In this extended abstract, we consider the scarcity of training data for temporal logics. We summarize a recently performed study on the capabilities of GANs and Wasserstein GANs equipped with Transformer encoders to generate sensible and challenging formulas in the prototypical temporal logic LTL. The approach produces novel and unique formula instances without the need for autoregression. The generated data can be used as substitute for real training data when training a classifier, and training data can be generated from a dataset that is too small to be trained on directly.

1 INTRODUCTION

Deep learning is increasingly applied to more untraditional domains that involve complex symbolic reasoning. Examples include the application of deep neural network architectures to SAT Selsam et al. (2019); Selsam & Bjørner (2019); Ozolins et al. (2021), SMT Balunovic et al. (2018), temporal specifications in verification Hahn et al. (2021); Schmitt et al. (2021), symbolic mathematics Lample & Charton (2020), mathematical problems with program synthesis Drori et al. (2021), or theorem proving Loos et al. (2017); Bansal et al. (2019); Huang et al. (2019); Urban & Jakubuv (2020); Paliwal et al. (2020).

The acquisition of training data for symbolic reasoning domains, however, is a challenge. Existing instances, such as benchmarks in competitions Biere & Claessen (2010); Froleyks et al. (2021); Jacobs et al. (2017), are too few to be trained on directly such that training data is laboriously generated synthetically, for example in Selsam et al. (2019); Lample & Charton (2020); Kaliszyk et al. (2017); Saxton et al. (2019); Schmitt et al. (2021); Evans et al. (2018).

In this extended abstract, we report on a recent study on the automatic generation of training data for Linear-time Temporal Logic (LTL) Pnueli (1977a) with Generative Adversarial Networks (GANs) Goodfellow et al. (2014). LTL is a prototypical hardware specification language of high importance for the formal methods and verification community. It forms the basis of many industrial specification languages, such as PSL IEEE-Commission et al. (2005a) or System Verilog Assertions (SVA). The model can be used to construct large amounts of meaningful training data from a significantly smaller data source.

We summarize our approach to tackle the challenges of applying GANs to this setup, as they can not immediately be applied: Specifications in LTL are symbolic and thereby discontinuous by nature as well as of variable length. Additionally, we recap on the experiments performed to show the use of our GAN approach for the generation of LTL formulas. We provide details on how to achieve a stable training of a standard GAN and a Wasserstein GAN Arjovsky et al. (2017) both equipped with Transformer encoders. We summarize the particularities of their training behavior, such as the effects of adding different amounts of noise to the one hot embeddings. Secondly, we show for an LTL satisfiability classifier that the generated data can be used as a substitute for real training data, and, especially, that training data can be generated from a dataset that is too small to be trained on directly. In particular, out of 10K training instances, a dataset consisting of 400K instances can be generated, on which a classifier can successfully be trained on.

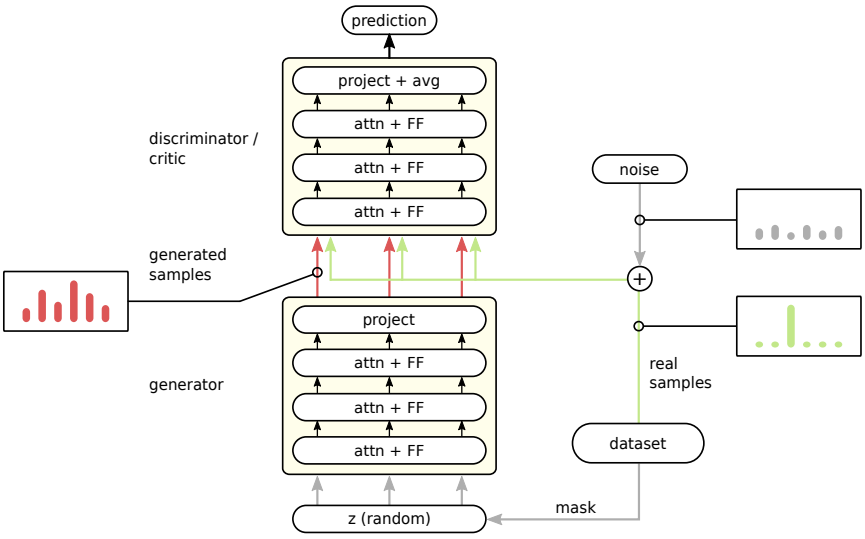

Figure 1: Transformer GAN for generating LTL formulas with visualizations of the per-position one-hot space.

2 Linear-time Temporal Logic (LTL)

Linear-time Temporal Logic (LTL) Pnueli (1977b) is a temporal logic that forms the basis of many practical specification languages, such as PSL IEEE-Commission et al. (2005b), STL Maler & Nickovic (2004), or SVA Vijayaraghavan & Ramanathan (2005). LTL extends propositional logic with temporal modalities $\mathcal{U}$ (until) and $\bigcirc$ (next). There are several derived operators, such as $\Diamond\varphi \equiv \mathit{true}\,\mathcal{U}\,\varphi$ and $\Box\varphi \equiv \neg\Diamond\neg\varphi$. $\Diamond\varphi$ states that φ will *eventually* hold in the future and $\Box\varphi$ states that φ holds *globally*. Operators can be nested: $\Box\Diamond\varphi$, for example, states that φ has to occur infinitely often. LTL specifications describe a system's behavior and its interaction with an environment over time. For example, given a process 0 and a process 1 and a shared ressource, the formula $\Box(r_0 \rightarrow \Diamond g_0) \wedge \Box(r_1 \rightarrow \Diamond g_1) \wedge \Box\neg(g_0 \wedge g_1)$ describes that whenever a process requests (r) access to the shared ressource, it will eventually be granted (g). The formula $\Box\neg(g_0 \wedge g_1)$ ensures that grants are only given mutually exclusive.

So far, the construction of datasets for LTL formulas has been done in two ways Hahn et al. (2021): Either by obtaining LTL formulas from a fully random generation process, which likely results in unrealistic formulas, or by sampling conjunctions of LTL specification patterns occuring in practice Dwyer et al. (1999). We slightly refined the pattern-based method with two operations, to obtain a healthy amount of unsatisfiable and satisfiable instances. Since the formula length correlates to unsatisfiability, we filter for equal proportions of classes per formula length. The tree size of the formulas is restricted to 50. We call the resulting dataset with around $380K$ instances `LTLbase`. We will refer the interested reader to the full version for more details after the double-blind reviewing ends.

3 Architecture

The Transformer GAN architecture for generating LTL formulas is depicted in Figure 1. It consists of two Transformer encoders as discriminator/critic and generator, respectively. The inner layers of the encoders are largely identical to standard transformers Vaswani et al. (2017), but their input and output processing is adjusted to the GAN setting. We use an embedding dimension of $d_{emb} = 128$, $n_h = 8$ attention heads, and a feed-forward network dimension of $d_{FF} = 1024$ for both encoders as default.

The generator's input is a real scalar random value with uniform distribution $[0, 1]$ for each position in the sequence. It is mapped to d_{emb} by an affine transformation before being processed by the first layer. The position-wise padding mask is copied from the real data during training, so the lengths of

real and generated instances at the same position in a batch are always identical. During inference, the lengths can either be sampled randomly or copied from an existing dataset similar to training. Either way, the generator encoder's padding mask is predetermined so it has to adequately populate the unmasked positions. With V being the vocabulary, and $|V|$ being the size of the vocabulary, an affine transformation to dimensionality $|V|$ and a softmax is applied after the last layer. The generator's output lies, thus, in the same space as one-hot encoded tokens. We use $n_{lG} = 6$ layers for our default model's generator.

A GAN discriminator and WGAN critic are virtually identical in terms of their architecture. The only difference is that a critic outputs a real scalar value where a discriminator is limited to the range $[0, 1]$, which we achieve by applying an additional logistic sigmoid in the end. To honor their differences regarding the training scheme, we use both terms when referring to exchangeable properties and make no further distinctions between them. For input processing, their $|V|$-dimensional (per position) input is mapped to d_{emb} by an affine transformation. After the last layer, the final embeddings are aggregated over the sequence by averaging and a linear projection to a scalar value (the prediction logit) is applied. Our default model uses $n_{lD} = 4$ layers. We achieved best results with slightly more generator than discriminator/critic layers.

Working in the $|V|$-sized one-hot domain poses harsh constraints on the generator's output. Contrary to continuous domains where GANs are usually employed, each component of a real one-hot vector is, by definition, either 0 or 1. If the generator were to identify this distribution and use it as criterion to tell real and generated instances apart, this would pose a serious difficulty for training. We therefore sample a $|V|$-sized vector of Gaussian noise $N(0, \sigma^2_{\text{real}})$ for each position (see Figure 1). We add it to the real samples' one-hot encoding and re-normalize it to sum 1 before handing them to the discriminator/critic. We use a value of $\sigma_{\text{real}} = 0.1$ for all models to get comparable results.

4 Summary of Experimental Results

In the following, we summarize the results of our experiments. For details on the training settings, we will refer the interested reader to the full version after the double-blind reviewing process ends.

4.1 Producing Syntactically Correct Formulas

Generating valid LTL formulas. During training we periodically sample several generated instances and convert them to their text representation, which involves taking the $argmax$ at every position. We then try to parse a prefix-encoded tree from the resulting tokens. If the parsing of a formula is successful and no tokens remain in the sequence, we note this instance as *fully correct*. Both GAN and WGAN variants increase the measure relatively continuously, but eventually reach their limit around 30K training steps with roughly 30% for WGAN and 15% for GAN. Both generators are able to produce a large fraction of fully correct temporal specifications, despite the length of the instances (up to 50 tokens) and the non-autoregressive, fixed-step architecture. We list two random examples below:

$$\neg(h \to h)\,\mathcal{W}\bigcirc(g \vee h) \wedge (g \wedge g) \wedge \diamondsuit\square\neg\square j \wedge \neg\square j\,\mathcal{W}\,\neg b \wedge \square(\square h \wedge \bigcirc j \to \square j) \wedge \diamondsuit\diamondsuit j \quad ,$$
$$\bigcirc\bigcirc\bigcirc(c \vee i) \wedge \neg d \wedge \neg\diamondsuit\bigcirc c\,\mathcal{W}\,\neg c \wedge \square(\square d \wedge \neg((b \leftrightarrow c) \hookleftarrow$$
$$\leftrightarrow \square c) \to \bigcirc(c \leftrightarrow \square d)) \wedge \square(b \wedge d \to \diamondsuit\diamondsuit\square\square d) \wedge c \quad .$$

Differences in homogeneity. Comparing the correct instances generated by the WGAN and GAN variants, we find that often, the latter would produces formulas in the likes of

$$\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\diamondsuit i \wedge \bigcirc\bigcirc i \wedge \bigcirc\bigcirc\bigcirc\bigcirc\neg\bigcirc\square\neg\neg\diamondsuit i \wedge \neg\neg(g \wedge g \wedge i) \quad ,$$

which contains repetitions (of the $\bigcirc$-operator) or easily stringed together sequences. In fact, some GAN runs achieved fully correct fractions above 30% (higher than WGAN), but these exclusively produced formulas with such low internal variety. To quantify this, we calculated a *sequence entropy* which treats the number of occurrences of the same token in the sequence relative to the sequences length as probability. This metric decreases for the GAN variant during training but remains stable for WGANs. Further experiments were thus carried out on the WGAN variant.

Table 1: Accuracies of Transformer classifiers trained on different datasets (5-run average with standard deviations in parentheses); all are validated on the `LTLbase` dataset.

trained on	bs	train acc @ 30K	val acc @ 30K	train acc @ 50K	val acc @ 50K
`LTLbase`	1024	96.6% (0.5)	95.5% (0.4)	98.1% (0.3)	**96.1%** (0.3)
	512	92.4% (0.7)	93.0% (0.8)	95.4% (0.5)	95.0% (0.8)
`LTLbase` 100K	512	95.3% (0.7)	88.3% (0.9)	98.1% (0.3)	87.8% (1.0)
`LTLbase` 10K	512	100% (0.1)	76.4% (1.7)	100% (0.0)	75.5% (1.5)
`Generated`	1024	95.4% (0.2)	93.6% (1.0)	97.1% (0.1)	**93.9%** (0.3)

Effects of additive noise on one-hot representation. The effect of adding different amounts of noise to the one-hot representation of real instances strongly affected the performance of the GAN scheme, which is unable to work without added noise. Stronger noise however improved this variant's performance. WGAN models on the other hand were not significantly influenced by added noise and are able to be trained without it. Adding uniform noise has no benefit over Gaussian noise.

4.2 Substituting Training Data with Generated Instances

To obtain a dataset of generated instances, we first train a WGAN with default parameters but smaller batch size of 512 on a set of 10K instances from the `LTLbase` dataset. After training for 15K steps, we collect 800K generated formulas from it and call this dataset `Generated-raw`. This set is processed similar to the original base dataset: Duplicates are removed and satisfiable and unsatisfiable instances are balanced to equal amounts per formula size. We randomly keep 400K instances and call the resulting dataset `Generated`.

We train several deep neural classifiers for LTL formula satisfiability on different training sets and compare their performance in Table 1. The validation accuracy is computed on the `LTLbase` dataset. Training on differently-sized subsets of `LTLbase` shows that a reduced number of training samples strongly decreases performance. 10K instances lead to immense overfitting and poor accuracy. We were not able to train a classifier on this few formulas with significantly higher accuracy.

A classifier trained on the `Generated` set however achieves almost the identical validation accuracy on the base set as the classifier that was actually trained on it. Note that the WGAN that created this set was trained on only 10K instances. We therefore find that the data produced by the WGAN is highly valuable as it can serve as full substitute for the complete original training data even when provided with much fewer examples.

Two instances of `LTLbase` ($(\square\neg a) \wedge (\square\bigcirc a)$ and $(\square\square e) \wedge (\square\neg e)$), i.e., only 0.02%, reappear in the 800K large data set `Generated-raw`. Additionally, in `Generated-raw`, only 2.3K of the 800K (0.28%) generated formulas were duplicates, which displays an enormous degree of variety.

5 Conclusion

In this extended abstract, we summarized a recent study on the capabilities of (Wasserstein) GANs equipped with two Transformer encoders to generate sensible training data for Linear-time Temporal Logic (LTL). Results show that both can be trained directly on the one-hot encoding space when adding Gaussian noise. Additionally, training data can be succesfully generated and the data can be used as a meaningful substitute when training a classifier. After the double-blind review ends, we refer the interested reader to the full version, which contains additional tables, plots and experiments. In particular, it contains an additional experiment on adding an uncertainty measure to the generator's output, where the models then generated instances on which a classifier was harder to train on. A limitation of this approach is the fixed maximum length of the generated sequences. In general, logical and mathematical reasoning with neural networks requires large amounts of sensible training data. Better datasets will lead to powerful neural heuristics and end-to-end approaches for many symbolic application domains, such as mathematics, search, verification, synthesis and computer-aided design. This novel, neural perspective on the generation of symbolic reasoning instances is also of interest to generate data for tool competitions, such as SAT, SMT, code and hardware synthesis, or model checking competitions.

REFERENCES

Martín Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein GAN. *arXiv*, 2017.

Mislav Balunovic, Pavol Bielik, and Martin T. Vechev. Learning to solve SMT formulas. In *NeurIPS*, 2018.

Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher order logic theorem proving. In *ICML*, 2019.

Armin Biere and K Claessen. Hardware model checking competition. In *Hardware Verification Workshop*, 2010.

Iddo Drori, Sunny Tran, Roman Wang, Newman Cheng, Kevin Liu, Leonard Tang, Elizabeth Ke, Nikhil Singh, Taylor L. Patti, Jayson Lynch, Avi Shporer, Nakul Verma, Eugene Wu, and Gilbert Strang. A neural network solves and generates mathematics problems by program synthesis: Calculus, differential equations, linear algebra, and more. *arXiv*, 2021.

Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ISCE*, 1999.

Richard Evans, David Saxton, David Amos, Pushmeet Kohli, and Edward Grefenstette. Can neural networks understand logical entailment? In *ICLR*, 2018.

Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Sat competition 2020. *Artificial Intelligence*, 2021.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *NeurIPS*, 2014.

Christopher Hahn, Frederik Schmitt, Jens U. Kreber, Markus Norman Rabe, and Bernd Finkbeiner. Teaching temporal logics to neural networks. In *ICLR*, 2021.

Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. Gamepad: A learning environment for theorem proving. In *ICLR*, 2019.

IEEE-Commission et al. Ieee standard for property specification language (psl). *IEEE Std 1850-2005*, 2005a.

IEEE-Commission et al. IEEE standard for property specification language (PSL). *IEEE Std 1850-2005*, 2005b.

Swen Jacobs, Roderick Bloem, Romain Brenguier, Rüdiger Ehlers, Timotheus Hell, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup, and Adam Walker. The first reactive synthesis competition (SYNTCOMP 2014). *STTT*, 2017.

Cezary Kaliszyk, François Chollet, and Christian Szegedy. Holstep: A machine learning dataset for higher-order logic theorem proving. In *ICLR*, 2017.

Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In *ICLR*, 2020.

Sarah M. Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. In *LPAR*, 2017.

Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *FORMATS*, 2004.

Emils Ozolins, Karlis Freivalds, Andis Draguns, Eliza Gaile, Ronalds Zakovskis, and Sergejs Kozlovics. Goal-aware neural SAT solver. *arXiv*, 2021.

Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving. In *AAAI*, 2020. URL `https://aaai.org/ojs/index.php/AAAI/article/view/5689`.

Amir Pnueli. The temporal logic of programs. In *FOCS*, 1977a.

Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pp. 46–57. ieee, 1977b.

David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. In *ICLR*, 2019.

Frederik Schmitt, Christopher Hahn, Markus N Rabe, and Bernd Finkbeiner. Neural circuit synthesis from specification patterns. *arXiv*, 2021.

Daniel Selsam and Nikolaj Bjørner. Guiding high-performance SAT solvers with unsat-core predictions. In *SAT*, 2019.

Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. In *ICLR*, 2019.

Josef Urban and Jan Jakubuv. First neural conjecturing datasets and experiments. In *CICM*, 2020.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 2017.

Srikanth Vijayaraghavan and Meyyappan Ramanathan. *A practical guide for SystemVerilog assertions*. Springer Science & Business Media, 2005.