

Constrained Training of Neural Networks via Theorem Proving

Mark Chevallier¹, Matthew Whyte¹ and Jacques D. Fleuriot¹

¹Artificial Intelligence and its Applications Institute, School of Informatics, University of Edinburgh, UK

Abstract

We introduce a theorem proving approach to the specification and generation of temporal logical constraints for training neural networks. The distinctive benefit of our approach is the fully rigorous implementation of generalised constrained training, that guarantees correctness of implementation at every step.

1. Introduction

Neural networks are powerful tools that can be used for low-level action and perception in complicated environments, e.g. autonomous driving, but whether they subsequently implement intended behaviour is difficult to assert, let alone guarantee [1]. We want a neural network that can be taught constraints such as “never go past this threshold” or “always avoid this region” and actually obey them as intended when deployed. Learning that incorporates such constraints has been the subject of recent efforts, where many approaches have been used to express and inject knowledge into model training. The latter encompass logical encoding directly in PyTorch [2] and TensorFlow [3] and a range of other bespoke encoding and integration [4, 5, 6, 7, 8, 9]. Given the ad-hoc nature of these implementation mechanisms, it is difficult to provide strong guarantees about the correctness of logical constraints and hence about the actual training.


Our work introduces a neurosymbolic pipeline that injects a loss function, which measures the breach of a constraint expressed in linear temporal logic over finite traces [10] (LTL_f), into the training of any neural network, with formal guarantees of correctness.


To achieve this, we use the proof-assistant Isabelle/HOL [11] to formally verify our model of the logical system, the soundness of the loss function \mathcal{L} and the loss function derivative $d\mathcal{L}$, which is used for backpropagation in training neural networks. The formal proof that \mathcal{L} is sound with respect to the semantics of LTL_f guarantees that it will enforce the intended behaviour during training. We then automatically generate OCaml code from our provably correct formal specifications [12], and integrate it into a PyTorch neural network via a library that provides OCaml bindings for Python [13]. The use of code generation here, rather than an ad-hoc, manually written implementation, guarantees that the implementation will match the specification.

OVERLAY 2022: 4th Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis, November 28, 2022, Udine, Italy

✉ m.chevallier@sms.ed.ac.uk (M. Chevallier); m.j.whyte@sms.ed.ac.uk (M. Whyte); jdf@ed.ac.uk (J. D. Fleuriot)

ORCID 0000-0001-5307-7018 (M. Chevallier); 0000-0002-6867-9836 (J. D. Fleuriot)

 © 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

Lastly, to demonstrate the approach’s guarantees, we build an example implementation, and perform some experiments confirming that it behaves as expected. The results show that the method yields a practical implementation with formal guarantees.

2. Formalising linear temporal logic

We define a `state` as a function from the integers to the reals that tracks several values in a learning problem that might be compared: a constant, or some measurement that a neural network uses in its training e.g. how far a robotic hand is from a barrier at a given moment in time. The integers passed to a state function are simply the indices to those values of interest. As we are working with LTL_f (see below) we allow the state functions to vary over time. With each state function encoding information about a system at a specific time-step, we define a path of length N as a set of state functions encoding the evolution of a system over N time-steps.

Our formalisation uses a variant of linear temporal logic [14] known as LTL_f [10], which is interpreted over finite paths and is often viewed as a more natural choice for applications in AI, e.g. planning [15], where processes are usually of finite duration.

Following Fischer et al. we take comparisons of a state function’s values as our atomic propositions ρ , namely $t < t'$, $t \leq t'$, $t = t'$, $t \neq t'$ [5]. Thus we build constraints in LTL_f from these comparisons (`comp`) and the constraints (`constraint`) arising from LTL_f ’s operators.

The LTL_f formulae are thus: ρ , $\rho_1 \wedge \rho_2$, $\rho_1 \vee \rho_2$, $\mathcal{N}\rho$ (Next), $\Box\rho$ (Always), $\Diamond\rho$ (Eventually), $\rho_1\mathcal{U}\rho_2$ (Weak Until), $\rho_1\mathcal{R}\rho_2$ (Strong Release). In LTL_f , these have the usual semantics of LTL [14], except at the end of a path. In particular, $\neg(\mathcal{N}\rho)$ holds for all ρ at the final time-step i [10].

We define Isabelle datatypes `comp` and `constraint` for the real-valued comparisons and LTL_f constraints respectively. This approach to specifying the language in higher-order logic is known as a deep embedding [16]. Doing so will enable us to prove that the loss function is sound and, importantly, generate fully self-contained code for defining LTL_f constraints that will be used as part of the training of a neural network.

Next, we formalise the `eval` function, which characterises the semantics of LTL_f by recursively evaluating the truth-value of a constraint over a path. Given the complexity of LTL_f , we also prove a number of LTL_f equivalences, which confirms that our `eval` behaves as expected and matches LTL_f semantics.

3. A LTL-based loss function and its derivative

The loss function \mathcal{L} takes an LTL_f constraint ρ , a path P and a relaxation factor γ , and returns a real value. Its purpose is to return a proportional positive value if ρ is breached under path P , and 0 otherwise. It needs to satisfy several important properties:

1. $\mathcal{L}(\rho, P, \gamma) \geq 0$;
2. \mathcal{L} is differentiable w.r.t. any of the terms that the constraint compares, for $\gamma > 0$;
3. (Soundness) $\lim_{\gamma \rightarrow 0} \mathcal{L}(\rho, P, \gamma) = 0 \iff \rho(P)$, where $\rho(P)$ is the truth value of ρ on P .

When formalising \mathcal{L} , it must be differentiable for the neural network to be able to learn from it via backpropagation (see Section 4). We therefore use *soft*, differentiable versions of various

functions to quantify the satisfaction of constraints. Based on the work by Cuturi and Blondel [17], we define a binary softmax (and softmax analogously) function, \max_γ , as:

$$\max_\gamma(a_1, a_2) = \begin{cases} \max(a_1, a_2) & \gamma \leq 0 \\ \gamma \log(e^{a_1/\gamma} + e^{a_2/\gamma}) & \text{otherwise} \end{cases}$$

Every soft function takes an additional parameter γ , the relaxation factor used for \mathcal{L} . The intention behind this parameter is that as $\gamma \rightarrow 0$, $\max_\gamma \rightarrow \max$, and that it is differentiable when $\gamma > 0$. We formalise this in Isabelle, and prove it has the desired properties, going on to formalise its derivative and proving that it is correct. Different soft functions capture losses from the $t \neq t'$ comparison, again using γ as a parameter in the same way.

We proceed to define \mathcal{L} recursively, over the constraint and path, with γ as a parameter. For all the LTL_f operators, the \mathcal{L} function, in common with our `eval` function, recurses over the constraint from the outside in, and recurses down the path as required for temporal operators. Once we have formulated \mathcal{L} , we show via a series of lemmas and an inductive proof on the constraint datatype that it has the expected soundness property with respect to the `eval` function.

We next construct a derivative $d\mathcal{L}$ for the \mathcal{L} function, to be used for gradient-based methods in PyTorch (see Section 4). The derivative must be defined with respect to each time-step i and state-value index j at that time-step along the finite trace.

The $d\mathcal{L}$ function formalisation is structured similarly to our formalisation of the \mathcal{L} function, defined recursively over the components of the LTL_f constraint passed to it and essentially follows from repeated applications of the chain rule. In defining it, we make extensive use of the derivatives of the soft-functions we defined earlier. We formally prove that $d\mathcal{L}$ is the correct derivative for our loss function and thus guarantee that when used for backpropagation it will achieve the desired results.

4. A PyTorch-compatible LTL loss function

With our formalisation and proofs complete, what remains is to integrate them into the PyTorch environment. Unfortunately, there does not exist a mechanism for generating Isabelle functions as Python code. Instead, we choose to generate intermediate representations of \mathcal{L} and $d\mathcal{L}$ in OCaml since our recursive Isabelle functions can be straightforwardly translated to type-safe OCaml ones using the code generation machinery of Isabelle. Moreover, there exists a Python library that can be used to call OCaml functions from within Python code [13].

In order to produce computable code, we need to map the real numbers of Isabelle to floating points [18]. As this is an approximation, it naturally has some scope for machine arithmetic errors, although the code generated for the various functions is fully faithful to their definitions in Isabelle. We generate an OCaml module `LTL_Loss`.

Python bindings for the OCaml definitions of \mathcal{L} and $d\mathcal{L}$ are incorporated into a PyTorch `autograd.Function` object through the `forward` and `backward` methods, respectively [19]. These methods are required for the loss function to form part of a computational graph in PyTorch and enable training based on gradient descent. Consequently, our `LTL_Loss` module, implemented as a subclass of `autograd.Function`, is functionally identical to a differentiable

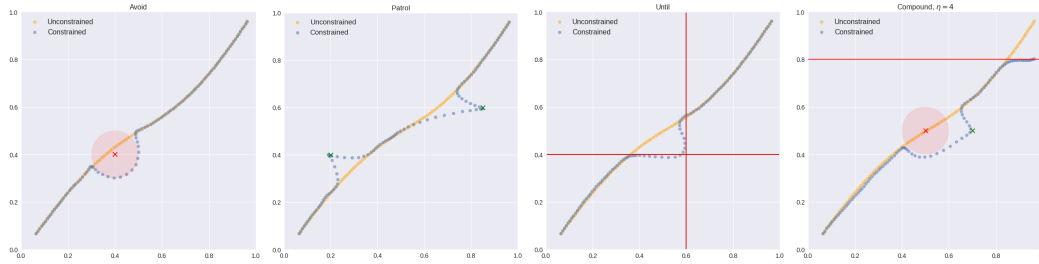


Figure 1: Avoid, Patrol, Until and Compound tests. The unconstrained output trajectory is in yellow, and the constrained output trajectory is in blue.

PyTorch operation on tensors. Significantly, we know exactly how LTL_Loss should behave when computing gradients with autograd, as it is solely characterised by its formalisation in Isabelle.

5. Experiments and Discussion

We base our experiments on those of Innes and Ramamoorthy [2] where they assess LTL_f constraint training in a spatial environment. Each of the tests takes place in a 2-dimensional planar environment with Cartesian co-ordinates. The training data follow a spline-shaped curve consisting of $N=100$ sequenced points in the plane following the curve with small random perturbations, simulating a demonstrator moving to some destination. We train a feed-forward neural network to learn a Dynamic Movement Primitive (DMP) [20] to follow this trajectory.

We lay out 4 different constraints to test. **Avoid:** The trajectory (always) avoids an open ball of radius 0.1 around the point (0.4, 0.4). **Patrol:** The trajectory eventually reaches (0.2, 0.4) and (0.85, 0.6) in the plane. **Until:** The y co-ordinate of the trajectory cannot exceed 0.4 until its x co-ordinate is at least 0.6. **Compound:** A more complicated test combining several conditions. The trajectory should avoid an open ball of radius 0.1 around the point (0.5, 0.5), while eventually touching the point (0.7, 0.5). Further, the y co-ordinate of the trajectory should not exceed 0.8.

When we run our tests, we first train the neural network ignoring the logical constraint, then we use the latter as part of its loss calculations. The differences between the two sets of results demonstrates the effectiveness of the logical constraint as used in the loss function for the training. Visual depictions of the results are given in Fig. 1.

This demonstrates that our approach has the desired intent: a theorem-proving based approach (with associated code-generation) can be used to rigorously inject constraints into neural network training to guarantee the expected behaviour. Our approach is generic, so in principle a different formalism, e.g. a continuous-time logic, could be used instead of LTL_f , and applied to any applicable neural network. Moreover, by formalising the derivative of the loss function, we unlock the potential to reason formally about the traversal of the loss surface during gradient descent. Given our results, we believe this work opens the way to a tighter integration between fully-formal symbolic reasoning in a theorem prover and machine learning.

References

- [1] N. Kalra, S. M. Paddock, Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?, *Transportation Research Part A: Policy and Practice* 94 (2016) 182–193. URL: <https://www.sciencedirect.com/science/article/pii/S0965856416302129>. doi:<https://doi.org/10.1016/j.tra.2016.09.010>.
- [2] C. Innes, R. Ramamoorthy, Elaborating on Learned Demonstrations with Temporal Logic Specifications, in: M. Toussaint, A. Bicchi, T. Hermans (Eds.), *Proceedings of Robotics: Science and System XVI*, 2020. doi:[10.15607/RSS.2020.XVI.004](https://doi.org/10.15607/RSS.2020.XVI.004).
- [3] G. Marra, F. Giannini, M. Diligenti, M. Gori, Lyrics: A general interface layer to integrate logic inference and deep learning, in: U. Brefeld, E. Fromont, A. Hotho, A. Knobbe, M. Maathuis, C. Robardet (Eds.), *Machine Learning and Knowledge Discovery in Databases*, Springer International Publishing, Cham, 2020, pp. 283–298.
- [4] S. Badreddine, A. d’Avila Garcez, L. Serafini, M. Spranger, Logic tensor networks, *Artificial Intelligence* 303 (2022) 103649. URL: <https://www.sciencedirect.com/science/article/pii/S0004370221002009>. doi:<https://doi.org/10.1016/j.artint.2021.103649>.
- [5] M. Fischer, M. Balunovic, D. Drachler-Cohen, T. Gehr, C. Zhang, M. Vechev, DL2: Training and querying neural networks with logic, in: *International Conference on Machine Learning*, 2019, pp. 1931–1941.
- [6] Z. Hu, X. Ma, Z. Liu, E. Hovy, E. Xing, Harnessing deep neural networks with logic rules, in: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Association for Computational Linguistics, Berlin, Germany, 2016, pp. 2410–2420. URL: <https://aclanthology.org/P16-1228>. doi:[10.18653/v1/P16-1228](https://doi.org/10.18653/v1/P16-1228).
- [7] T. Li, V. Srikumar, Augmenting neural networks with first-order logic, in: A. Korhonen, D. R. Traum, L. Màrquez (Eds.), *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, Association for Computational Linguistics, 2019, pp. 292–302. URL: <https://doi.org/10.18653/v1/p19-1028>. doi:[10.18653/v1/p19-1028](https://doi.org/10.18653/v1/p19-1028).
- [8] R. Stewart, S. Ermon, Label-free supervision of neural networks with physics and domain knowledge, in: *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.
- [9] J. Xu, Z. Zhang, T. Friedman, Y. Liang, G. Broeck, A Semantic Loss Function for Deep Learning with Symbolic Knowledge, in: J. Dy, A. Krause (Eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, PMLR, Stockholm, Sweden, 2018, pp. 5502–5511. URL: <http://proceedings.mlr.press/v80/xu18h.html>.
- [10] G. De Giacomo, M. Y. Vardi, Linear temporal logic and linear dynamic logic on finite traces, in: *IJCAI’13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, Association for Computing Machinery, 2013, pp. 854–860.
- [11] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL: a proof assistant for higher-order logic, volume 2283, Springer Science & Business Media, 2002.
- [12] F. Haftmann, L. Bulwahn, Code generation from Isabelle/HOL theories, Isabelle documentation (2021). URL: <https://isabelle.in.tum.de/doc/codegen.pdf>.
- [13] L. Mazare, Using Python and OCaml in the same Jupyter notebook (2019). URL: <https://blog.janestreet.com/using-python-and-ocaml-in-the-same-jupyter-notebook>.

- [14] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), 1977, pp. 46–57. doi:10.1109/SFCS.1977.32.
- [15] J. Baier, S. Mcilraith, Planning with First-Order Temporally Extended Goals using Heuristic Search, in: Proceedings of the National Conference on Artificial Intelligence, volume 1, 2006.
- [16] R. J. Boulton, A. D. Gordon, M. J. Gordon, J. Harrison, J. Herbert, J. Van Tassel, Experience with embedding hardware description languages in hol., in: TPCD, volume 10, Citeseer, 1992, pp. 129–156.
- [17] M. Cuturi, M. Blondel, Soft-DTW: a differentiable loss function for time-series, arXiv preprint arXiv:1703.01541 (2017).
- [18] F. Haftmann, J. Hölzl, T. Nipkow, Code_Real_Approx_By_Float, HOL_Library (2021). URL: https://isabelle.in.tum.de/library/HOL/HOL-Library/Code_Real_Approx_By_Float.html.
- [19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al., Pytorch: An imperative style, high-performance deep learning library, Advances in neural information processing systems 32 (2019) 8026–8037.
- [20] S. Schaal, J. Peters, J. Nakanishi, A. Ijspeert, Learning movement primitives, in: Robotics research. the eleventh international symposium, Springer, 2005, pp. 561–572.

Acknowledgements

The authors would like to express their gratitude to the Artificial Intelligence and its Applications Institute, the Informatics Graduate School at the University of Edinburgh, and the Engineering and Physical Sciences Research Council for funding this work.