Robustness as a Refinement Type
Verifying Neural Networks in Liquid Haskell and F*

Wen Kokke[1,2], Ekaterina Komendantskaya[2], Daniel Kienitz[2], and David Aspinall[1*]

[1] School of Informatics, University of Edinburgh, Edinburgh, UK
{wen.kokke, david.aspinall}@ed.ac.uk
[2] Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, UK
{dk50, e.komendantskaya}@hw.ac.uk

Abstract. We introduce StarChild and Lazuli, two proof-of-concept libraries which leverage the type system and theorem proving capabilities of F* and Liquid Haskell, respectively, to verify properties of pre-trained neural networks. We largely focus on StarChild, as the F* syntax is slightly more concise, but Lazuli implements the same functionality. Currently, both libraries are capable of verifying small models. Performance issues arise for larger models. Optimising the libraries is future work.
We make two novel contributions. We demonstrate that (a) it is possible to leverage a sufficiently advanced type system to model properties of neural networks such as robustness as types, and check them without any proof burden; and in service of that, we demonstrate that (b) it is possible to approximately translate neural network models to SMT logic.

Introduction Neural networks are widely used for classification and pattern-recognition tasks in computer vision, signal processing, data mining, and many other domains. They have always been valued for their ability to work with noisy data, yet only recently [7], it was discovered that they are prone to adversarial attacks—specially crafted inputs that lead to unexpected outputs. Verifying properties of neural networks, such as, *e.g.*, robustness against adversarial attacks, is a recognised research challenge [4]. Several current approaches involve: (a) encoding properties as satisfiability problems [2,3]; (b) proving properties via abstract interpretation [5]; (c) or using an interactive theorem prover [1].

F* [6] and Liquid Haskell [8] are functional languages with refinement types, *i.e.*, types can be refined with SMT-checkable constraints. For instance, the type of positive reals (`x:ℝ{x > 0}`), or booleans which are true (`b:bool{b ≡ true}`), or a type of neural networks which are robust against adversarial attacks. Unlike, *e.g.*, Python, F* and Liquid Haskell are referentially transparent, which means the semantics of pure programs in these languages can be directly encoded in the SMT logic. This tight integration allows users to specify neural network models and their properties in the same language, while leveraging the powerful automated verification offered by SMT solvers!

* The work was funded by the National Cyber Security Center, UK.

StarChild: Verifying Neural Networks in F* StarChild leverages the type system of F* to verify properties of pre-trained neural networks. Users can either write their models directly in F*, or export them from Python. To illustrate, we train a model to mimic the AND gate, and export it:

```
val m : network (*n_inputs*) 2 (*n_outputs*) 1 (*n_layers*) 1
let m = NLast { weights    = [[17561.5R]; [17561.5R]]
              ; biases     = [−25993.1R]
              ; activation = Sigmoid }
```

We can verify properties of models using either refinement types or assertions. For instance, we can check that the model `m` correctly implements the AND gate:

```
let _ = assert(run m [1.0R;1.0R] ≡ [1.0R]) // true  AND true  ≡ true
let _ = assert(run m [0.0R;1.0R] ≡ [0.0R]) // false AND true  ≡ false
let _ = assert(run m [1.0R;0.0R] ≡ [0.0R]) // true  AND false ≡ false
let _ = assert(run m [0.0R;0.0R] ≡ [0.0R]) // false AND false ≡ false
```

Assertions in F* have no significance at runtime. They are checked statically, as part of type checking. You can think of **`assert`** as a function with type:

```
val assert : b:bool{b ≡ true} → ()
```

Its argument is a `bool` which must be `true`, which F* checks using an SMT solver. We are not limited to assertions we can run, but can also check assertions using quantifiers, which are infeasible or impossible to run. For instance, we can check that the model `m` is robust for inputs within an ε-interval:

```
let epsilon  = 0.2R
let truthy x = dist x 1.0R ≤ epsilon
let falsy  x = dist x 0.0R ≤ epsilon
let _ = assert(∀(x₁:ℝ{truthy x₁})(x₂:ℝ{truthy x₂}).run m [x₁;x₂] ≡ [1.0R])
let _ = assert(∀(x₁:ℝ{falsy  x₁})(x₂:ℝ{truthy x₂}).run m [x₁;x₂] ≡ [0.0R])
let _ = assert(∀(x₁:ℝ{truthy x₁})(x₂:ℝ{falsy  x₂}).run m [x₁;x₂] ≡ [0.0R])
let _ = assert(∀(x₁:ℝ{falsy  x₁})(x₂:ℝ{falsy  x₂}).run m [x₁;x₂] ≡ [0.0R])
```

The assertions cover the entire ε-interval around 1.0 and 0.0, which we could not have achieved by executing them. The program type checks, and hence we know the model `m` is, in fact, robust for $\varepsilon = 0.2$.

All models specified using StarChild are usable in type refinements and assertions. Better yet, F* takes care of the translation to the SMT logic for us! F* translates programs to the SMT logic by normalising it, translating constructs to their SMT equivalents where possible, and translating the rest as uninterpreted functions. For instance, the expression `run m [x₁;x₂]` normalises to

```
sigmoid(x₁ × 17561.5R + x₂ × 17561.5R − 25993.1R)
```

When translating this term, F* maps $\times$, $+$, and $-$ to their equivalent in the SMT logic, and maps maps `sigmoid` to an uninterpreted function. Its definition uses the exponential function, which most SMT solvers do not support. However, the SMT solver cannot reason about uninterpreted functions. To circumvent this, we use linear approximations, *e.g.*, `lsigmoid`, during verification:

```
let lsigmoid x = 0.0R `min` (0.25R × x + 0.5R) `max` 1.0R
```

The use of approximations introduces an error, which impacts the accuracy of the verification. Investigating the bounds on these errors is future work.

References

1. Bagnall, A., Stewart, G.: Certifying true error: Machine learning in Coq with verified generalisation guarantees. AAAI (2019)
2. Katz, G., et al.: The Marabou framework for verification and analysis of deep neural networks. In: CAV 2019, Part I. LNCS, vol. 11561, pp. 443–452. Springer (2019)
3. Kwiatkowska, M.Z.: Safety verification for deep neural networks with provable guarantees (invited paper). In: Fokkink, W., van Glabbeek, R. (eds.) CONCUR 2019,. LIPIcs, vol. 140, pp. 1:1–1:5. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
4. Pertigkiozoglou, S., Maragos, P.: Detecting adversarial examples in convolutional neural networks. CoRR **abs/1812.03303** (2018), http://arxiv.org/abs/1812.03303
5. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: An abstract domain for certifying neural networks. PACMPL **3**(POPL), 41:1–41:30 (2019), https://doi.org/10.1145/3290354
6. Swamy, N., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y.: Dependent types and multi-monadic effects in F^*. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016. ACM Press (2016). https://doi.org/10.1145/2837614.2837655, https://doi.org/10.1145/2837614.2837655
7. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., Fergus, R.: Intriguing properties of neural networks. CoRR **abs/1312.6199** (2014), https://arxiv.org/abs/1312.6199
8. Vazou, N.: Liquid Haskell: Haskell as a Theorem Prover. Ph.D. thesis, University of California, San Diego, USA (2016), http://www.escholarship.org/uc/item/8dm057ws