
NeuroTrace: Spec-Aware Neural Network Runtime

Robert Joseph George
California Institute of Technology

With
Apart Research

Abstract

AI-written neural-network code can run, train, export, and even receive a verifier result while silently drifting from the specification the user intended. This creates a false-assurance failure mode: a proof may apply to the wrong runtime, output head, input domain, training behavior, or optimized implementation. We introduce **NeuroTrace**, a spec-driven observability framework for AI-written neural models. To our knowledge, NeuroTrace is the first framework that turns neural architecture and verification specs into live runtime contracts for PyTorch execution, training, optimized implementations, and verifier applicability. It generates TorchLean-style formal architecture artifacts, PyTorch runtimes, hook and torch.fx traces, training-time SpecTrace logs, robustness evidence, and SpecFuzz mutations. During execution, it records layer paths, tensor shapes, dtypes, activation summaries, gradient flow, mode assumptions, and artifact provenance. If the runtime, export, optimized path, or property binding departs from the accepted spec, NeuroTrace emits a replayable violation artifact and rejects verifier applicability. We evaluate the prototype on MLP, CNN, and attention-style models, plus controlled reproductions of documented bug classes such as ONNX/export drift, logits/probability confusion, input-contract violations, dropout/mode drift, and broken gradient flow. The long-term goal is to scale this into a practical observability layer for larger ML systems, where formal specs help detect real implementation, training, compiler, and deployment bugs before verification evidence is trusted.

1. Introduction

Neural-network software is increasingly assembled through chains of generated and transformed artifacts. A user may describe an intended architecture; an AI coding agent may generate PyTorch code; a verifier may consume an ONNX export and a VNN-LIB property; and a developer may later replace part of the model with a faster fused kernel, compiler graph, or optimized runtime. Each artifact can look correct in isolation while the full chain no longer describes the same

computation. This creates a false-assurance problem: a verifier result may be attached to the wrong runtime, output head, input domain, training behavior, or optimized implementation.

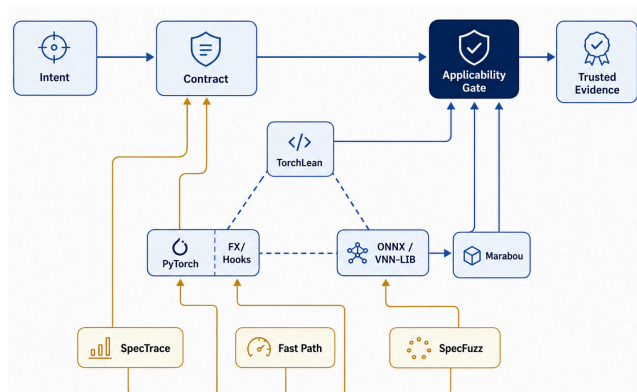


Figure 1: NeuroTrace Framework.

To me, this problem matters because verification evidence is only meaningful if it still applies to the system actually being executed. Existing neural-network verifiers are powerful, but they usually answer a downstream question: given this network and this property, does the property hold? [1-5] NeuroTrace focuses on the question that comes before that: is this still the

network-property pair the user meant to verify? This is not a rare edge case. Deep-learning systems often fail through data bugs, logic bugs, tensor-shape mistakes, model-conversion errors, stale exports, logits/probability confusion, accidental detach(), or train/eval mode drift [16-27]. I discuss the closest prior work and the remaining gap in Section 2.

The threat model is ordinary but consequential artifact drift. A runtime may match the specification at first, then diverge during training, export, optimization, or verifier use. A fast implementation may be introduced for performance, a VNN-LIB property may target an outdated output dimension, gradients may silently stop flowing, or a verifier may prove a property over an export that no longer corresponds to the current PyTorch model. The danger is not only that the model is wrong, but that evidence about the model remains trusted after its link to the intended specification has broken. To address this, I introduce **NeuroTrace**, a spec-driven observability framework for AI-written neural models. It treats a neural specification as a live runtime contract rather than a static document. Starting from an architecture and property contract, it triangulates across TorchLean-style formal artifacts, PyTorch runtimes, hook traces, torch.fx graphs, ONNX/VNN-LIB artifacts, Marabou outputs, SpecTrace training logs, optimized runtime candidates, and SpecFuzz mutations. If any channel departs from the accepted contract, NeuroTrace emits a replayable violation artifact. My main contributions are:

1. **Spec-driven neural observability:** neural architecture and verification specs become runtime contracts for PyTorch execution, training, optimized implementations, and verifier applicability.
2. **A multi-artifact applicability gate:** verifier evidence is accepted only when the surrounding artifacts still agree with the accepted contract.
3. **Controlled demonstrations on real bug classes:** the prototype checks MLP, CNN, and attention-style examples, plus controlled reproductions of output-shape drift, ONNX/export drift, logits/probability confusion, input-contract violations, dropout/mode drift, and broken gradient flow.

2. Related Work

In reviewing prior work, I found that many systems address important parts of neural verification, but not the full false-assurance problem targeted here. Neural-network verifiers such as Marabou and CROWN-style methods check whether a fixed network satisfies a fixed property, while DNNV, VNN-LIB, and VNN-COMP standardize verifier-facing networks and property formats [1-5]. Vehicle is closest on the specification side, since it provides a high-level language for neural-network specifications and connects them to verifiers and theorem provers [4]. My recent work on TorchLean is closest on the formal-semantics side: it gives neural networks a Lean-based substrate with typed tensors, a shared graph IR, verified differentiation, and certificate checking [5]. These systems are valuable, but they mostly assume that the network and property being checked are already the right artifacts. NeuroTrace instead asks the prior question: does the artifact chain still describe the computation the user intended?

NeuroTrace also builds on ideas from verified synthesis and testing. Clover introduced a closed-loop view of verifiable code generation, where code, documentation, and formal annotations are checked together rather than trusted independently [6]. In my work on BRIDGE, we extend this idea by treating synthesis as a multi-artifact problem across code, specifications, theorem statements, and proofs [9]. NeuroTrace applies the same triangulation principle to neural systems: the artifacts are architecture contracts, TorchLean specs, PyTorch runtimes, hook traces, compiler graphs, optimized fast paths, training traces, ONNX/VNN-LIB files, and verifier outputs. Related testing and monitoring systems such as DeepXplore, TensorFuzz, DeepHunter, NNSmith, ML observability tools, and runtime verification also expose failures or monitor execution [8,9,10,11,12]. The difference is that NeuroTrace monitors the **neural specification boundary** itself. It is useful when an AI coding agent, developer, compiler, or optimized runtime changes the model’s meaning without causing an immediate crash. The new information it provides is not only whether a property was proved, but whether that proof remains applicable, where the artifact chain drifted, and which replayable trace or mutation exposed the mismatch. **Table 1 summarizes the closest related work and the gap NeuroTrace addresses.**

<i>Prior work / system</i>	<i>Fixed NN property</i>	<i>Formal spec</i>	<i>Intermediate triangulation</i>	<i>Runtime trace</i>	<i>Training / gradients</i>	<i>Export / property binding</i>	<i>Fast path checks</i>	<i>Applicability gate</i>
<i>Marabou / CROWN verifiers [1,2]</i>	✓	—	—	—	—	◐	—	—
<i>DNNV / VNN-LIB / VNN-COMP [3-5]</i>	✓	◐	◐	—	—	✓	—	—
<i>Vehicle [6]</i>	✓	✓	◐	—	—	◐	—	—
<i>TorchLean [7]</i>	✓	✓	✓	◐	◐	◐	◐	◐
<i>Clover [8]</i>	—	✓	✓	—	—	—	—	◐
<i>BRIDGE [9]</i>	—	✓	✓	—	—	—	—	◐
<i>DeepXplore / TensorFuzz /</i>	—	—	◐	◐	—	—	—	—

<i>DeepHunter</i> [12-15]								
<i>NNSmith / DL</i> <i>compiler testing</i> [10]	—	—	•	—	—	•	✓	—
<i>ML observability /</i> <i>runtime verification</i> [28-31]	•	•	—	✓	•	—	•	•
<i>NeuroTrace</i>	✓	✓	✓	✓	✓	✓	✓	✓

3. Methods

I built NeuroTrace around a practical issue I kept running into while thinking about verified ML systems: the model that is easiest to specify is rarely the model that is eventually executed. A clean PyTorch implementation is useful as a reference, but real ML systems at frontier labs often replace it with faster paths: fused blocks, compiler graphs, quantized exports, optimized attention kernels, or deployment-specific runtimes. I wanted to test whether a system could keep checking that these later implementations, exports, verifier artifacts, and training runs still implement the same underlying specification.

The central design choice is **triangulation**. No single framework is trusted on its own. NeuroTrace treats the architecture contract as the source of truth, then checks it through independent views: TorchLean-style formal artifacts, PyTorch runtimes, forward hooks, torch.fx, ONNX/VNN-LIB, Marabou, SpecTrace training logs, Fast Path candidates, and SpecFuzz mutations. At least my goal was not to make every backend fully formal in one weekend; the goal is to make disagreement visible and to reject evidence when the artifact chain has drifted. The pipeline starts by converting a user architecture prompt into a structured contract. The contract records the model family, input shape, layer dimensions, activation functions, output dimension, output semantics, training assumptions, and optional verification property. For example, an MLP contract records whether outputs are raw logits or probabilities, which matters for classifier margins and robustness properties. This contract becomes the accepted basis specification. Later stages are not allowed to silently rewrite it; if the runtime changes the final layer, if an export is stale, or if a property targets the wrong output index, it reports drift.

Now my framework generates a TorchLean-style specification artifact from the contract, giving a typed view of the architecture: layer composition, tensor shapes, output arity, activation choices, and expected graph structure. TorchLean is useful because it is designed around typed neural semantics and a shared graph representation for execution, differentiation, and verification. In NeuroTrace, TorchLean is the reference for what the model is supposed to mean; everything else checks whether the implementation still agrees.

The PyTorch runtime is generated from the same contract because PyTorch is where most neural code is written, edited, trained, and optimized [10]. NeuroTrace instruments the runtime with hooks that record module names, operator types, shapes, dtypes, train/eval mode, activation

summaries, and nonfinite values. It also extracts a torch.fx graph when possible. Hooks answer “what ran on this input?” while torch.fx answers “what graph does PyTorch expose to tracing or compiler tooling?” This gives the first triangulation:

Contract $\leftrightarrow$ TorchLean-style spec $\leftrightarrow$ PyTorch hooks $\leftrightarrow$ torch.fx graph

Spec Verify checks operator order, layer-boundary shapes, activation choices, output dimension, dtype, runtime mode, and output semantics. Executable probes are included, but they are gated: a mutated runtime can still execute and agree with itself on sampled inputs even if it no longer matches the accepted specification. Probe agreement therefore supports alignment only after structural and spec-level checks pass.

I also wanted NeuroTrace to connect to actual certified robustness, not only runtime checking. For supported small models, NeuroTrace exports the runtime to ONNX, generates a VNN-LIB property, and invokes Marabou on a simple robustness or logit-margin query [1,4,27]. This gives the second triangulation:

Contract $\leftrightarrow$ PyTorch runtime $\leftrightarrow$ ONNX export $\leftrightarrow$ VNN-LIB property $\leftrightarrow$ Marabou result

Marabou’s output is treated as conditional evidence. A solver may prove a property of the ONNX/VNN-LIB pair it received, while that pair may be stale, attached to the wrong output head, or inconsistent with the current PyTorch runtime. NeuroTrace records hashes and metadata for the contract, runtime, export, property, and verifier result, then accepts verifier evidence only if the surrounding artifacts remain aligned. I also include three stress checks. **SpecTrace** turns the contract into a training-time monitor, recording loss, layer execution, activation summaries, gradient norms, dtype, runtime mode, and nonfinite values during a controlled training run. **Fast Path refinement** checks whether an optimized implementation preserves the accepted contract under a declared tolerance, even if its internal trace differs. **SpecFuzz** mutates the contract, runtime, or property with changes such as activation swaps, output-width edits, malformed input boxes, changed robustness radii, and wrong output indices.

The final applicability gate accepts evidence only when the relevant independent channels agree. The output is not simply “verified” or “not verified.” it also emits an inspectable evidence trail: architecture contract, formal artifact, PyTorch runtime, hook trace, torch.fx summary, verifier manifest, training trace, fast-path report, SpecFuzz report, and violation JSON files. My whole goal to conclude was that verification evidence should be trusted only while the artifact chain remains observably tied to the accepted neural specification.

4. Results

So my framework was evaluated on a small but concrete set of models and failure modes, with a focus on whether NeuroTrace could detect when verification evidence stopped applying to the intended neural specification. Due to limited Codex credits and the hackathon timeframe, I did not

run a large-scale benchmark or statistical study. Instead, I focused on end-to-end demonstrations that exercise the main artifact boundaries: MLP, CNN, and attention-style models; TorchLean-style spec generation; PyTorch runtime execution; hook and torch.fx triangulation; ONNX/VNN-LIB export; Marabou robustness queries; training-time SpecTrace; Fast Path refinement; and SpecFuzz mutation testing. Check my live demo:) that walks through these stages interactively and shows the generated evidence artifacts, including hook traces, verifier manifests, training traces, fast-path reports, mutation reports, and violation JSON files.

The main observation to me is that NeuroTrace catches several realistic classes of neural artifact drift before verifier evidence is accepted. In controlled examples, the system rejects output-shape edits, wrong output-head/property bindings, logits-vs-probabilities drift, input-domain violations, dropout or train/eval mode drift, stale export-style mismatches, and broken gradient flow from accidental detach(). These are not artificial-only concerns: the tested bug classes line up with documented deep-learning failure modes, including tensor-shape faults, data and logic bugs, ONNX converter “wrong model” failures, PyTorch logits/probability confusion, dropout and BatchNorm train/eval semantics, detach() gradient behavior, and torch.fx limitations around input-dependent dynamic control flow [16-27]. The honest claim is that NeuroTrace did not reproduce each named public bug end-to-end; rather, it exercises the same bug classes in controlled settings and shows that they produce explicit spec-drift artifacts instead of silent acceptance. I really would like to scale this further on real life systems and large scale studies.

The robustness and observability results support the same conclusion. For supported small models, NeuroTrace can export ONNX, generate VNN-LIB properties, and invoke Marabou on simple robustness or logit-margin queries; however, the solver result is accepted only if the contract, PyTorch runtime, export, property binding, and verifier metadata remain aligned. SpecTrace similarly shows that initial spec/runtime agreement is not enough: during training, NeuroTrace records loss, activations, gradients, dtype, runtime mode, and nonfinite events, so failures such as missing gradients or mode drift become observable. Fast Path refinement checks whether an optimized implementation preserves the accepted contract under a declared tolerance, and SpecFuzz tests whether mutations to the contract/runtime/property are detected. Overall, the strongest result is not that every model is verified, but that NeuroTrace makes false assurance visible: proof and optimization evidence are rejected when the surrounding artifact chain no longer matches the accepted neural specification! I am very excited to see how this will work at scale:) I am tired of debugging bugs all the time, and hope now during training it can catch bugs and let me know as soon as it violates a property!

5. Discussion and Limitations

I am particularly interested in AI safety from the formal verification side: not just making models perform well, but making sure that any claim about a model is actually attached to the system being run. NeuroTrace is motivated by that gap. The main lesson from this project is that many

safety failures are not simply verifier failures. They happen earlier or around the verifier: the PyTorch runtime changes, the export becomes stale, the property points to the wrong output, training breaks gradient flow, or a fast implementation no longer matches the intended contract. NeuroTrace tries to make this drift observable before verifier evidence is trusted.

So what I believe is that neural verification needs an applicability layer. Existing tools such as Marabou, DNNV, VNN-LIB workflows, CROWN-style methods, and TorchLean each address important parts of the problem, but none of them fully solves the end-to-end engineering reality alone. TorchLean gives a strong formal semantics, but it does not yet cover arbitrary PyTorch programs or all optimized runtimes. Marabou and related verifiers can prove useful properties, but they work best on supported model/property fragments and do not by themselves guarantee that the exported artifact is the current runtime. ONNX/VNN-LIB standardize interfaces, but they also introduce boundaries where stale exports, wrong output bindings, or conversion drift can enter. NeuroTrace is meant to sit around these tools and ask: does this evidence still apply?

The main obvious limitation is scope. This is a hackathon prototype, tested on small MLP, CNN, and attention-style examples plus controlled reproductions of real bug classes. It is not a large-scale benchmark, and it does not prove that all instances of these bugs are caught. It also assumes the architecture contract is meaningful; if the contract is wrong, the system may preserve the wrong intent. The current checks are strongest for supported structural, runtime, training, export, and property-binding failures. They are not full semantic equivalence proofs between arbitrary PyTorch, TorchLean, ONNX, optimized kernels, and hardware execution.

Future Work

The natural next step is to scale NeuroTrace beyond small examples. This means supporting larger transformer blocks, LoRA/adapters, quantized models, fused kernels, neural operators, and scientific ML systems. It also means building a larger corpus of real ML-system bugs: stale exports, wrong output bindings, preprocessing drift, train/eval mistakes, broken gradient flow, compiler drift, and optimization bugs.

Another important direction is strengthening the formal link to TorchLean. In the current prototype, TorchLean provides a formal specification channel; future versions could use stronger translation validation between PyTorch graphs, TorchLean IR, ONNX artifacts, and optimized runtimes. Longer term, I think NeuroTrace could become a practical spec-observability layer for AI-written ML code: formal tools provide the contracts, verifiers provide property evidence, and NeuroTrace checks that the runtime, training process, optimized implementation, and exported artifacts remain tied to the same specification. Interested in hearing what all I should do from you all since this is not a live hackathon:) feedback would be appreciated.

6. Conclusion

To conclude, NeuroTrace is built around one idea I haven't seen addressed directly in the community: verification evidence should only be trusted while it remains attached to the system actually being run. A proof over a stale export, a property targeting the wrong output head, or a training run with broken gradient flow can all look fine in isolation while the underlying specification has already drifted. This prototype treats the neural spec as a live contract rather than a static document, and uses multiple independent artifact channels to make that drift visible before any verifier result is accepted.

The main finding is not that NeuroTrace solves neural verification end to end. It shows that an applicability layer around existing tools is both feasible and useful. Many failures in AI-written ML systems aren't caused by an unsound verifier — they happen earlier, when the artifact chain quietly stops describing the intended computation. Catching that gap is what NeuroTrace is for.

Longer term, I think this points toward something the community needs more of: formal specs, runtime observability, and verification tools treated as a connected system rather than independent deliverables. A verifier result is only as trustworthy as the chain of artifacts it was applied to. Getting that chain right — and keeping it right through training, optimization, and deployment — is an engineering problem that formal methods alone don't yet solve, but one they're well-positioned to help with once the observability layer exists.

Code and Data

- **Code repository:** <https://github.com/Robertboy18/NeuroTrace>
- **Other artifacts** (optional): Demo:  `video2495599136.mp4`

LLM Usage Statement

I used Codex and GPT-5.5 Pro as development assistants and helped with figure creations during this project. They helped with brainstorming the NeuroTrace design, iterating on implementation ideas and experimenting with the user-facing workflow (I am so bad at frontend design, I have to admit almost all of it was done with codex help). LLM assistance is also part of the NeuroTrace framework itself: the system begins from user prompts and turns them into architecture/specification artifacts that are then checked by the rest of the pipeline. All technical claims, implementation behavior (as in the actual verifiers, fuzzy and checking alignment), and reported results were manually reviewed and verified by me against the generated artifacts, traces, code outputs, and experiments. The final interpretation is my own, informed by my prior work on TorchLean and BRIDGE and my familiarity with the related neural verification and verified synthesis literature. Sorry for the long document:) I just had to explain all the points well!

References

- [1] Katz, G., Huang, D. A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., Dill, D. L., Kochenderfer, M. J., and Barrett, C. (2019). **The Marabou Framework for Verification and Analysis of Deep Neural Networks.** *Computer Aided Verification (CAV 2019)*, Lecture Notes in Computer Science, vol. 11561, pp. 443–452. Springer. DOI: 10.1007/978-3-030-25540-4_26. URL: https://link.springer.com/chapter/10.1007/978-3-030-25540-4_26
- [2] Zhang, H., Weng, T.-W., Chen, P.-Y., Hsieh, C.-J., and Daniel, L. (2018). **Efficient Neural Network Robustness Certification with General Activation Functions.** *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*. URL: <https://proceedings.neurips.cc/paper/2018/hash/d04863f100d59b3eb688a11f95b0ae60-Abstract.html>
- [3] Shriver, D., Elbaum, S., and Dwyer, M. B. (2021). **DNNV: A Framework for Deep Neural Network Verification.** *Computer Aided Verification (CAV 2021)*, Lecture Notes in Computer Science, pp. 137–150. Springer. DOI: 10.1007/978-3-030-81685-8_6. URL: https://link.springer.com/chapter/10.1007/978-3-030-81685-8_6
- [4] VNN-LIB Developers. (2021). **VNN-LIB: A Standard for Neural Network Verification Specifications.** URL: <https://www.vnnlib.org/>
- [5] Brix, C., Bak, S., Johnson, T. T., and Wu, H. (2024). **The Fifth International Verification of Neural Networks Competition (VNN-COMP 2024): Summary and Results.** arXiv:2412.19985. URL: <https://arxiv.org/abs/2412.19985>
- [6] Daggitt, M. L., Kokke, W., Atkey, R., Arnaboldi, L., and Komendantskaya, E. (2022). **Vehicle: Interfacing Neural Network Verifiers with Interactive Theorem Provers.** arXiv:2202.05207. DOI: 10.48550/arXiv.2202.05207. URL: <https://arxiv.org/abs/2202.05207>
- [7] George, R. J., Cruden, J., Zhong, X., Zhang, H., and Anandkumar, A. (2026). **TorchLean: Formalizing Neural Networks in Lean.** arXiv:2602.22631. URL: <https://arxiv.org/abs/2602.22631>
- [8] Sun, C., Sheng, Y., Padon, O., and Barrett, C. (2024). **Clover: Closed-Loop Verifiable Code Generation.** *AI Verification: First International Symposium (SAIV 2024)*, Lecture Notes in Computer Science, pp. 134–155. Springer. DOI: 10.1007/978-3-031-65112-0_7. URL: <https://arxiv.org/abs/2310.17807>

- [9] George, R. J., Eisenach, C., Ghai, U., Perrault-Joncas, D., Anandkumar, A., and Foster, D. (2025). **BRIDGE: Building Representations in Domain-Guided Program Verification**. arXiv:2511.21104. URL: <https://arxiv.org/abs/2511.21104>
- [10] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). **PyTorch: An Imperative Style, High-Performance Deep Learning Library**. *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*. URL: <https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html>
- [11] Reed, J. K., DeVito, Z., He, H., Ussery, A., and Ansel, J. (2022). **torch.fx: Practical Program Capture and Transformation for Deep Learning in Python**. *Proceedings of Machine Learning and Systems (MLSys 2022)*, vol. 4, pp. 638–651. URL: <https://proceedings.mlsys.org/paper/2022/hash/7c98f9c7ab2df90911da23f9ce72ed6e-Abstract.html>
- [12] Pei, K., Cao, Y., Yang, J., and Jana, S. (2017). **DeepXplore: Automated Whitebox Testing of Deep Learning Systems**. *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017)*. DOI: 10.1145/3132747.3132785. URL: <https://dl.acm.org/doi/10.1145/3132747.3132785>
- [13] Odena, A., Olsson, C., Andersen, D., and Goodfellow, I. (2019). **TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing**. *Proceedings of the 36th International Conference on Machine Learning (ICML 2019)*, Proceedings of Machine Learning Research, vol. 97, pp. 4901–4911. URL: <https://proceedings.mlr.press/v97/odena19a.html>
- [14] Xie, X., Ma, L., Juefei-Xu, F., Xue, M., Chen, H., Liu, Y., Zhao, J., Li, B., Yin, J., and See, S. (2019). **DeepHunter: A Coverage-Guided Fuzz Testing Framework for Deep Neural Networks**. *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*, pp. 146–157. DOI: 10.1145/3293882.3330579. URL: <https://dl.acm.org/doi/10.1145/3293882.3330579>
- [15] Liu, J., Lin, J., Ruffy, F., Tan, C., Li, J., Panda, A., and Zhang, L. (2023). **NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers**. *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2023), Volume 2*. DOI: 10.1145/3575693.3575707. URL: <https://dl.acm.org/doi/10.1145/3575693.3575707>
- [16] Islam, M. J., Nguyen, G., Pan, R., and Rajan, H. (2019). **A Comprehensive Study on Deep Learning Bug Characteristics**. *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*

(ESEC/FSE 2019). DOI: 10.1145/3338906.3338955. URL: <https://dl.acm.org/doi/10.1145/3338906.3338955>

[17] Wu, D., Shen, B., and Chen, Y. (2021). **An Empirical Study on Tensor Shape Faults in Deep Learning Systems**. arXiv:2106.02887. URL: <https://arxiv.org/abs/2106.02887>

[18] Jajal, P., Jiang, W., Tewari, A., Kocinare, E., Woo, J., Sarraf, A., Lu, Y.-H., Thiruvathukal, G. K., and Davis, J. C. (2024). **Interoperability in Deep Learning: A User Survey and Failure Analysis of ONNX Model Converters**. *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. DOI: 10.1145/3650212.3680374. URL: <https://dl.acm.org/doi/10.1145/3650212.3680374>

[19] PyTorch Contributors. (2026). **CrossEntropyLoss — PyTorch Documentation**. Accessed May 24, 2026. URL: <https://docs.pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html>

[20] PyTorch Contributors. (2022). **Could be clearer that Cross Entropy takes logits as input**. GitHub issue #82081. URL: <https://github.com/pytorch/pytorch/issues/82081>

[21] PyTorch Contributors. (2026). **Dropout — PyTorch Documentation**. Accessed May 24, 2026. URL: <https://docs.pytorch.org/docs/stable/generated/torch.nn.Dropout.html>

[22] PyTorch Contributors. (2026). **BatchNorm1d — PyTorch Documentation**. Accessed May 24, 2026. URL: <https://docs.pytorch.org/docs/stable/generated/torch.nn.BatchNorm1d.html>

[23] PyTorch Contributors. (2026). **torch.Tensor.detach — PyTorch Documentation**. Accessed May 24, 2026. URL: <https://docs.pytorch.org/docs/stable/generated/torch.Tensor.detach.html>

[24] PyTorch Contributors. (2026). **torch.fx — PyTorch Documentation**. Accessed May 24, 2026. URL: <https://docs.pytorch.org/docs/stable/fx.html>

[25] PyTorch Contributors. (2026). **torch.nn.functional.scaled_dot_product_attention — PyTorch Documentation**. Accessed May 24, 2026. URL: https://docs.pytorch.org/docs/stable/generated/torch.nn.functional.scaled_dot_product_attention.html

[26] PyTorch Contributors. (2024). **SDPA does not switch off dropout during evaluation**. GitHub issue #124464. URL: <https://github.com/pytorch/pytorch/issues/124464>

[27] ONNX Contributors. (2026). **ONNX Versioning Documentation: IR and Operator Set Versioning**. Accessed May 24, 2026. URL: <https://onnx.ai/onnx/repo-docs/Versioning.html>

- [28] Leucker, M., and Schallhart, C. (2009). **A Brief Account of Runtime Verification.** *Journal of Logic and Algebraic Programming*, 78(5), pp. 293–303. DOI: 10.1016/j.jlap.2008.08.004. URL: <https://www.sciencedirect.com/science/article/pii/S1567832608000775>
- [29] OpenTelemetry Authors. (2026). **What is OpenTelemetry?** OpenTelemetry Documentation. Accessed May 24, 2026. URL: <https://opentelemetry.io/docs/what-is-opentelemetry/>
- [30] Evidently AI. (2026). **Data Drift — Evidently Documentation.** Accessed May 24, 2026. URL: https://docs.evidentlyai.com/metrics/preset_data_drift
- [31] Naveed, H., Filho, S. A. S., Tahir, A., Knauss, E., and Grundy, J. (2024). **Towards Runtime Monitoring for Responsible Machine Learning.** *MODELS Companion 2024*. DOI: 10.1145/3640310.3674092. URL: <https://dl.acm.org/doi/10.1145/3640310.3674092>
- [32] Necula, G. C. (2000). **Translation Validation for an Optimizing Compiler.** *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI 2000)*, pp. 83–94. DOI: 10.1145/349299.349314. URL: <https://dl.acm.org/doi/10.1145/349299.349314>