TenPy: Dependent Types in Python for Tensor Shapes

ARNAV JAIN, University of Kansas, USA
SANKHA NARAYAN GURIA, University of Kansas, USA

Tensor libraries like NumPy are essential tools for programmers, enabling the efficient representation and manipulation of massive datasets. Although NumPy streamlines array processing, it abstracts away the implementation details of tensor operations, leaving the user responsible for ensuring that tensor shapes are compatible. This dependency on shape compatibility makes NumPy code highly susceptible to run-time errors that standard type checkers, such as MyPy or PyRight, are unable to detect because they are shape-unaware. In this work, we introduce a novel dependent type system for tensor operations, designed to detect shape errors statically. The key design decision in our system is to use dependent types to check application code but not verify the NumPy implementation itself. This allows us to write expressive type signatures that can do symbolic manipulation to verify arbitrary tensor shapes. We implement our system as MyPy plugin, giving it access to type check the entire Python ecosystem. We evaluate our system on an open-source machine learning project that relies on NumPy operations, including broadcasting, matrix multiplication, reshape, and slicing, demonstrating its effectiveness in a real-world context. In summary, we believe this shape-aware type system represents an important step forward in ensuring the robustness and correctness of code using tensor libraries like NumPy.

ACM Reference Format:
Arnav Jain and Sankha Narayan Guria. 2025. TenPy: Dependent Types in Python for Tensor Shapes. *Proc. ACM Program. Lang.* 1, 1 (December 2025), 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Tensor libraries are the cornerstone of the deep learning revolution. Tensors are representations of a sequence of numbers grouped into a specific shape. For example, a group of 12 numbers can be represented with 2 rows and 6 columns, giving it the shape 2x6. Python has emerged as the mainstream language to write tensor manipulating programs, with popular libraries like NumPy [9], PyTorch [20], and TensorFlow [1]. These libraries abstract away the implementation details of the tensors and define optimized implementations of commonly used tensor operations.

Since Python is dynamically typed, tensor computations written in Python are vulnerable to run time errors. Recent work on Python's gradual type systems [15, 23, 24] have improved this issue with many industrial grade implementations such as MyPy, PyRight, and PyreFly providing implementations that can be retrofitted on existing Python programs. However, each tensor operation often imposes a precondition on the shapes of the tensors it receives as input. For example, an operation might require that the larger first dimension of the two input tensors be divisible by the smaller first dimension of the second tensor. If this precondition is met, the operation outputs a manipulated tensor whose dimension is a function of the input tensors. When the precondition is not met, a shape incompatibility error is encountered, which generally causes a program crash. In

Authors' Contact Information: Arnav Jain, University of Kansas, Lawrence, USA, arnav.jain@ku.edu; Sankha Narayan Guria, University of Kansas, Lawrence, USA, sankha@ku.edu.

ACM 2475-1421/2025/12-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

fact, a majority of errors in deep learning programs are shape errors [12] yet none of the aforementioned gradual type systems can find these errors. Shape incompatibility errors occurring in the later stages of these programs, which often run for days or weeks, result in a significant loss of compute and time.

In this paper, we introduce TenPy, a dependent type system for Python programs for checking tensor shape errors. In TenPy, tensor shapes are defined as polymorphic types with the base type as the tensor types (`ndarray` for numpy and `tensor` for PyTorch), parameterized by the dimensions of the tensors. For example, a 2x6 tensor will have the type `ndarray<2, 6>`. TenPy allows type parameters to be symbolic (like `ndarray<2, a>`) to denote cases where the shapes are not fixed or known ahead of time. All types of library methods of numpy and PyTorch are defined as dependent types in the form of Python programs rather than static type signatures. This enables the type checker to compute expected types based on the arguments to an operation. For symbolic shapes, TenPy does local type inference to provide human-readable types for every program variable. Crucially, TenPy applies the dependent type checker to only the application code, but gracefully falls back to type checking without dependent types when checking library code. These choices allow TenPy to be used across practical Python programs to apply the expressive power of dependent types to catch shape errors. We discuss a few illustrative examples in § 2.

We formalize TenPy as a core object-oriented language λ_{py}. We formally define method type signatures separately for application code and library code, and define a separate type checking pass for these. Our core calculus defines TenPy dependent types as programs that uses no loops or recursion, thus making the type signatures termination-safe. This ensures that type checking stays decidable. We define the type checking rules formally for library methods only using base types but for application code using dependent types. We prove a soundness theorem for our system using standard progress and preservation lemmas. § 3 describes TenPy formally.

We implement TenPy as a plugin for mypy. Mypy is a popular type checker for Python officially supported by the Python Software Foundation that has support for user-defined plugins to extend mypy's behavior. Mypy supports type checking with nominal, literal (i.e., singletons), and generic (i.e., polymorphic) types over the entire Python programming language. Moreoever, mypy supports gradual typing semantics, that allows typed and untyped code to interact. TenPy benefits from this by extending only tensor shape checking semantics in the plugin, while interoperating with the broader Python typing ecosystem. § 4 describes our implementation in more detail.

We evaluate TenPy on SNG:♣ ... need eval?♣ § 5

Our results gives strong evidence that TenPy is a promising step forward to build a dependent type system that can interact with the broader Python ecosystem to prevent more classes of type errors.

2 Overview

This section provides an informal description of the type system and demonstrates its capabilities through examples, abstracting away the underlying formalisms.

The type system assigns every tensor created in NumPy a type representing its shape. As shown in Figure 1, the tensor x has the type `[2, 3]` and the tensor y has the type `[3, 4]`. The `@` operator denotes matrix multiplication. For an operation between a tensor of shape $M \times N$ and a tensor of shape $K \times P$ to be valid, the inner dimensions must match (i.e., N must equal K). The resulting tensor will have the shape $M \times P$. In the example, z is the result of multiplying x and y; since the inner dimensions match ($3 = 3$), the type of z is inferred as `[2, 4]`. Conversely, the variable *err* attempts to multiply y (a tensor with the shape 3×4) and x (a tensor with the shape 2×3). Since $4 \neq 2$, the shapes are incompatible. The type system correctly identifies this mismatch and reports a static error.

```
x = np.array([[1,2,3], [4,5,6]])
y = np.zeros((3,4))
reveal_type(x) # np.ndarray[Tuple[Literal[2], Literal[3]]]
reveal_type(y) # np.ndarray[Tuple[Literal[3], Literal[4]]]
z = x @ y
reveal_type(z) # np.ndarray[Tuple[Literal[2], Literal[4]]]
err = y @ x
reveal_type(err) # Error : shape mismatch
```

Fig. 1. Python code showing type-checking for matrix multiplication.

The complexity of the type system increases when introducing functions and operations such as broadcasting. Functions accept arbitrary arguments, meaning input variables often have unknown or partially known shapes. Furthermore, broadcasting allows operations between tensors of different ranks by aligning shapes to their rightmost dimension; corresponding dimensions must either be equal or one of them must be 1.

In Figure 2, the input tensor x in `func` has a rank of 2, but the dimension sizes are unknown. The `*` operator denotes broadcasting multiplication. Based on broadcasting rules between x (unknown) and y (3×4), the system infers that for the operation to be valid, the first dimension of x must be compatible with 3 (i.e., 1 or 3) and the second compatible with 4 (i.e., 1 or 4). While MyPy limitations prevent narrowing the type of the operands x and y in place, these constraints are tracked internally. Consequently, the subsequent operation `z @ x` fails because the second dimension of z (4) cannot align with the first dimension of x (which is constrained to 1 or 3).

Additionally, the system validates function arguments against shape signatures. In `func2`, the signature requires the second dimension of x to be exactly 4. Passing `invalid_arg` (which has a second dimension of 9) correctly triggers an error.

A key feature of this plugin, which distinguishes it from tools like Pythia [14], is support for slicing. Slicing selects tensor elements and alters the resulting shape. In `func2`, the value of a is unknown statically, making the size of the resulting slice unknown. The type system represents this as an generic integer type making the resulting shape `(int, 4)` since `:` means keep that dimension the same.

A current limitation is observed on line 13: even if the slice indices are concrete (e.g., `0:4`), applying them to a dimension of unknown size results in an unknown output dimension. However, when the input dimension is known (as with the second dimension of x), the system correctly calculates the new size. Finally, slicing with a single index reduces the rank of the tensor; if no dimensions remain, the result is inferred as a scalar type.

3 Technical details

Technical details

4 Implementation

The type system is implemented as a plugin for MyPy. MyPy was chosen because other major static analysis tools (Pylance, Pyre, Pyright) currently lack robust plugin support. During development, we identified a soundness issue within MyPy's handling of binary operators. Specifically, when a magic method (e.g., `__mul__`) failed, MyPy implicitly attempted the reverse operation (e.g., `__rmul__`) while still using the original method hook (the hook without the r), effectively treating all operators

```python
def func(x: np.ndarray[Tuple[int, int]]):
    y = np.zeros((3,4))
    reveal_type(x) # np.ndarray[Tuple[int, int]]
    reveal_type(y) # np.ndarray[Tuple[Literal[3], Literal[4]]]
    z = x * y
    reveal_type(z) # np.ndarray[Tuple[Union[1, 3], Union[1,4]]]
    err = z @ x # Error : Shape mismatch
    return z

def func2(x: np.ndarray[Tuple[int, Literal[4]]], a : int):
    y = x[0:a, :]
    reveal_type(y) # np.ndarray[Tuple[int, Literal[4]]]
    z = x[0:4, 1:3]
    reveal_type(z) # np.ndarray[Tuple[int, Literal[2]]]
    number = x[a, 0]
    reveal_type(number) # builtins.int

valid_arg = np.zeros((3,4))
invalid_arg = np.zeros((3,9))

a = func2(valid_arg)
b = func2(invalid_arg) # Error : Invalid arguement
```

Fig. 2. Python code showing type-checking for functions, broadcasting, and slicing.

as commutative. This behavior is unsound for non-commutative linear algebra operations. We submitted a fix which was merged into the MyPy repository (Pull Request #18995 [1]).

Type Representation

The default NumPy type signature is generic: `np.ndarray[Tuple[Any, ...], dtype(Any)]`. Our plugin refines the first type argument to represent shape explicitly. We map tensor dimensions to Python types as follows:

- **Unknown Dimension:** Represented by the built-in `int` type.
- **Static Dimension:** Represented by `Literal[k]` where k is the dimension size.
- **Constrained Dimension:** Represented by `Union[Literal[k], ...]` for dimensions that must be one of a finite set of values.

The second part of the default NumPy type signature stores the data type of the elements in the tensor. Our plugin does not currently track this; however, it can be updated in the future to provide better data type support.

Constraint Solving with Z3

To validate operations and infer output shapes, we employ the Z3 SMT solver [5]. The plugin does not solve constraints globally; rather, it solves constraints incrementally for each operation. This design choice leverages MyPy's existing infrastructure and provides localized error messages. The verification process follows these steps:

[1]https://github.com/python/mypy/pull/18995

(1) **Symbolic Encoding:** The input tensor shapes are converted into symbolic integers. A `z3.Int` variable is instantiated for every dimension of the left-hand side (LHS) and right-hand side (RHS) tensors.
(2) **Constraint Injection:** Known shape information is asserted as constraints. A concrete dimension D asserts `var == D`. A union of dimensions (D_1, D_2) asserts `Or(var == D1, var == D2)`. Unconstrained dimensions (`int`) imply no constraints.
(3) **Semantic Rule Application:** The semantics of the operation are asserted:
   - **Broadcasting:** Tensors are logically padded to equal rank. For each dimension pair (l, r), we assert the broadcasting invariant: `Or(l == 1, r == 1, l == r)`.
   - **Matrix Multiplication:** We assert broadcasting constraints on the batch dimensions. For the inner contracting dimensions, we assert `lhs[-1] == rhs[-2]`.
   - **Reshape:** We assert conservation of volume: `Product(lhs_dims) == Product(rhs_dims)`.
(4) **Solution Extraction (All-SAT):** We utilize an iterative blocking strategy to find all valid output shapes. When the solver returns `sat`, the model is recorded, and a blocking clause (negating that specific solution) is added to the solver. This repeats until `unsat` is returned. To prevent state explosion with dynamic dimensions, we employ a `SOLUTION_LIMIT`; if solutions exceed this threshold, the dimension falls back to `int`.
(5) **Type Reification:** The set of valid models is converted back into a MyPy-compatible `Union` or `Literal` type. If no models exist, a shape mismatch error is reported.

4.1 Slicing and Instantiation

Slicing is implemented via a syntactic heuristic rather than SMT solving. Ellipses are expanded, and static slices (e.g., `0:4`) on known dimensions result in exact shapes. Dynamic slices or slices on unknown dimensions decay to `int`. While less rigorous than the Z3 approach, this covers the majority of real-world use cases. The only case this approach does not yet cover is when a new dimension is added using `np.newaxis`.

We also provide shape-aware overrides for common creation functions (e.g., `np.zeros`, `np.ones`) by parsing their arguments into types. Shape-agnostic functions (e.g., `np.maximum`) are supported to ensure compatibility with existing open-source codebases.

4.2 Limitations

A primary trade-off of our incremental approach is the lack of inter-procedural context propagation. As demonstrated in Figure 3, the plugin checks `func` in isolation. The signature `Tuple[int, int]` implies dimensions of unknown size. Inside the function, the operation is structurally valid for *some* integers, so no error is raised. The caller passes incompatible concrete shapes (10×4 and 5×4), but MyPy verifies the arguments against the generic signature, effectively "forgetting" the concrete shapes upon function entry. This limitation could be addressed by adding flow-sensitivity or stricter annotation requirements, but we opted to adhere to standard MyPy behavior.

Design Decisions

We deliberately omitted two potential features to maintain consistency with MyPy's design philosophy.

First, we do not override user-provided type annotations, even when the plugin infers a stricter type. For example, in Figure 2, the user annotates x generically as an $N \times M$ matrix. Although the solver infers the stricter shape constraints $((1, 3), (1, 4))$, the plugin respects the user's broader annotation. We avoided reporting "strictness warnings" to prevent alerting the user to information they likely already know.

```
def func(x: np.ndarray[Tuple[int, int]],
         y: np.ndarray[Tuple[int, int]]):
    # Valid for some ints, so no local error
    err = x @ y
    return err

x = np.zeros((10,4))
y = np.zeros((5,4))

# Valid against signature (int, int), so no call-site error.
# but if the code is ran, it will throw an error
func(x, y)
```

Fig. 3. Example of an undetectable error due to lack of inter-procedural shape propagation.

Second, we limited the scope of flow-sensitivity. While we discovered a way to track variable shapes across state changes more aggressively, we restricted our analysis to the immediate operation context. This prevents the plugin's behavior from diverging too far from MyPy's standard inference logic.

5 Evaluation

To evaluate the effectiveness and performance of our type system, we utilized `numpy-ml` [4], an open-source repository containing pure NumPy implementations of standard machine learning algorithms. We selected the Reinforcement Learning (RL) module for our case study, as it provides a concise yet computationally dense usage of NumPy operations, including broadcasting, reductions, and element-wise manipulation.

Methodology and Code Modifications

The current implementation of the plugin requires code to modified to facilitate shape inference. We applied two primary transformations to the benchmark code:

(1) **Type Annotation:** We added shape annotations to function signatures, as the original code was untyped.
(2) **Operation Decomposition:** Complex one-liners were decomposed into intermediate assignments. As shown in Figure 4, compound operations like `x @ y.T` (matrix multiplication with an implicit transpose) were split so that the transpose occurs on a separate line. Additionally, property accessors (e.g., `.T`) were replaced with explicit function calls (e.g., `np.transpose()`) to trigger the plugin's hooks.

5.1 Inference Accuracy

After modifying the code, the plugin successfully tracked tensor shapes throughout the RL module without producing false positives. Figure 5 demonstrates the plugin's capability on a Deep Q-Network agent action selection method.

The system correctly propagates shapes through complex transformations. The input is transposed and multiplied by weights W, then broadcasted with bias b. Despite the ambiguous inputs, Z is correctly inferred as a rank-1 tensor. Notably, the system handles reduction operations accurately.

```
x = np.zeros((10,4))
y = np.zeros((5,4))

# Original compound operation (unsupported)
# z = x @ y.T

# Modified operations (supported)
y_temp = np.transpose(y)
z = x @ y_temp
```

Fig. 4. Example of code modifications. Compound operations are split into atomic function calls.

np.max and np.sum reduce the tensor dimension, which is correctly reflected in the types of the subsequent operations.

We observed a minor limitation with np.exp. For whatever reason, this function did not trigger the function hook but instead the dynamic class hook. Dynamic class hooks are handled before function hooks, so the type of the input is not yet known so the type of the output is unknown as well. We temporarily addressed this by manually annotating e_Z, allowing the subsequent summation and division operations to proceed, but this issue is being investigated further. The final action a is correctly identified as a scalar integer. This case study highlights the tool's ability to clarify unintuitive tensor transformations that often confuse developers.

Performance

To quantify the overhead introduced by the constraint solver, we benchmarked the execution time of MyPy under three configurations: (1) Default MyPy, (2) MyPy with a "Dummy" plugin (hooks registered but returning immediately), and (3) MyPy with our Shape Inference plugin.

Tests were conducted on an Intel Core i7-8550U CPU running Ubuntu. We averaged the runtime over 10 iterations, clearing the MyPy cache between runs to measure cold-start performance.

Configuration	Avg Time (s)	Overhead
Default MyPy	4.0905	-
Dummy Plugin	4.0982	+0.19%
Shape Inference	4.6061	+12.6%

Table 1. Performance benchmarks comparing standard MyPy against our plugin.

As shown in Table 1, the overhead of the plugin infrastructure itself is negligible (0.19%). The full shape inference system introduces a 12.6% overhead. While this increase is statistically significant, we argue it is acceptable for the development workflow. MyPy employs an incremental architecture (daemon mode), meaning subsequent checks only re-analyze modified files. Therefore, this overhead is primarily a one-time cost during the initial cold start or CI/CD pipelines.

6 Related work

The challenge of shape errors when working with tensors is a common issue across all major deep learning ecosystems. Researcher is most often related to static analysis techniques to detect these errors before execution. There are three main approaches to finding shape incompatibilities using

```python
def act(self, obs: np.ndarray[Tuple[int]]):
    E, P = self.env_info, self.parameters
    W : np.ndarray[Tuple[int, int]] = P["W"]
    b : np.ndarray[Tuple[int]] = P["b"]

    s = self._obs2num[obs]
    s = np.array([s]) if E["obs_dim"] == 1 else s

    # compute softmax
    s_transpose = np.transpose(s)
    temp = s_transpose @W
    Z = temp + b
    reveal_type(Z) # np.ndarray[Tuple[int]]

    temp = np.max(Z, axis=-1, keepdims=True)
    Z_temp = Z-temp

    # Manual annotation required for np.exp
    e_Z : np.ndarray[Tuple[int]] = np.exp(Z_temp)
    e_Z_temp = np.sum(e_Z, axis=-1, keepdims=True)

    action_probs = e_Z / e_Z_temp

    # sample action
    a = np.random.multinomial(1, action_probs).argmax()
    reveal_type(a) # Type: int (Scalar)
    return self._num2action[a]
```

Fig. 5. Shape inference on the numpy-ml reinforcement learning agent.

static analysis: relational dataflow analysis, constraint solving using SMT solvers, and programming language functionality.

Tools like Pythia [14] for TensorFlow utilize a relational approach. This method first converts the program to an Intermediate Representation (IR), which is then converted into relational tables (e.g., using Doop/Datalog) to infer the flow of possible tensor shapes. Similarly, Ariadne [6], another static tool for TensorFlow, uses the WALA program analysis framework to convert Python code into its own IR for dataflow analysis. While powerful, these methods require using specialized programs outside of the standard developer environment.

In contrast, other significant works rely on constraint solvers like Z3. These works include the type-based analysis for Python, PyTea [13], a similar system for Swift [21], work by UCLA [18], and Refty [7]. These tools model program operations as logical constraints and use the solver to check for satisfiability. The aforementioned systems collect all constraints across the entire program (or along specific execution paths, as in PyTea) and solve them once. This can answer the question of whether a valid shape exists for the whole program, but it often struggles to provide precise, localized error feedback. Our work checks the satisfiability at each operation, making it much better at giving the programmer feedback

A separate category of solutions, avoids the problem of solving constraints entirely by integrating shape analysis directly into the language design. In the Futhark programming language [11], shape checking is a 'built-in' feature of the compiler. A similar system has been created for Scala, but hasn't been widely adopted [3]. An even more powerful example is found in languages with dependent types, such as Idris [2], where tensor dimensions can be encoded directly into a variable's type. This allows the compiler to statically guarantee the absence of any shape errors, making it the most formal solution.

This body of static work is contrasted by dynamic approaches, such as ShapeFlow [25], which interprets TensorFlow code at runtime to discover shape inconsistencies and optimize the code accordingly. ShapeIt [26] also works at runtime and it provides annotations about the shapes of tensors.

There is also an approach which is a hybrid between static and dynamic checking for OCaml-Torch called GraTen [10] which adds refinement types and implicit assertions to catch potential errors.

A significant and more recent category of static analysis for Python is gradual type checking. The most prominent tools in this space are Mypy [19], Pylance [17], and Pyre [16]. All three of these are designed to find errors due to type inconsistencies. However, their analysis is limited due to Python's loose type system.

Python Taint [22] is a much more specific static analysis tool that uses dataflow techniques to find vulnerabilities. Gorbovitski et al [8] developed a deep, context-sensitive alias analysis for program optimization, but no public implementation is available

7 Conclusion

We introduced a novel shape-aware type system for NumPy to eliminate a broad class of runtime errors that plague tensor programming. By modeling the shapes directly in MyPy and leveraging the Z3 SMT solver, we demonstrated that it is possible to statically verify complex tensor operations (broadcasting, slicing, matrix multiplication, and reduction operations) without requiring extensive changes to existing codebases.

Looking forward, there are several ways to extend this work. First, we want to expand the plugin's coverage to include as many NumPy functions as possible. Second, we plan to investigate the dynamic class hooks being used for certain functions and find a workaround. Third, we plan to implement data type tracking for the numbers in the tensor. This will make the plugin more similar to how MyPy currently works and will give us the ability to find data type related issues (ie multiplying a 32-bit float with a 64-bit float). Finally, we want to adapt this implementation to PyTorch or TensorFlow.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] EDWIN BRADY. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. doi:10.1017/S095679681300018X

[3] Tongfei Chen. 2017. Typesafe abstractions for tensor operations (short paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala* (Vancouver, BC, Canada) *(SCALA 2017)*. Association for Computing Machinery, New York, NY, USA, 45–50. doi:10.1145/3136000.3136001

[4] year = 2021 note = Accessed: 2025-11-22 David Bourgin, howpublished = https://github.com/ddbourgin/numpy-ml. [n. d.]. numpy-ml.
[5] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.
[6] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. 2018. Ariadne: analysis for machine learning programs. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (PLDI '18)*. ACM, 1–10. doi:10.1145/3211346.3211349
[7] Yanjie Gao, Zhengxian Li, Haoxiang Lin, Hongyu Zhang, Ming Wu, and Mao Yang. 2022. REFTY: Refinement Types for Valid Deep Learning Models. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 1843–1855. doi:10.1145/3510003.3510077
[8] Michael Gorbovitski, Yanhong A. Liu, Scott D. Stoller, Tom Rothamel, and Tuncay K. Tekle. 2010. Alias analysis for optimization of dynamic languages. *SIGPLAN Not.* 45, 12 (Oct. 2010), 27–42. doi:10.1145/1899661.1869635
[9] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. doi:10.1038/s41586-020-2649-2
[10] Momoko Hattori, Naoki Kobayashi, and Ryosuke Sato. 2023. Gradual Tensor Shape Checking. arXiv:2203.08402 [cs.PL] https://arxiv.org/abs/2203.08402
[11] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. *SIGPLAN Not.* 52, 6 (June 2017), 556–571. doi:10.1145/3140587.3062354
[12] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 510–520. doi:10.1145/3338906.3338955
[13] Ho Young Jhoo, Sehoon Kim, Woosung Song, Kyuyeon Park, DongKwon Lee, and Kwangkeun Yi. 2021. A Static Analyzer for Detecting Tensor Shape Errors in Deep Neural Network Training Code. arXiv:2112.09037 [cs.LG] https://arxiv.org/abs/2112.09037
[14] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static Analysis of Shape in TensorFlow Programs. doi:10.4230/LIPIcs.ECOOP.2020.15
[15] Michael Lee, Michael Sullivan, and Ivan Levkivskyi. 2019. PEP 586 – Literal Types. https://peps.python.org/pep-0586/. Accessed: 2025-12-04.
[16] Meta Platforms, Inc. 2025. Pyre. https://pyre-check.org/.
[17] Microsoft. 2025. Pylance. https://marketplace.visualstudio.com/items?itemName=ms-python.vscode-pylance. Accessed: 2025-11-22.
[18] Zeina Migeed, James Reed, Jason Ansel, and Jens Palsberg. 2024. Generalizing Shape Analysis with Gradual Types. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 29:1–29:28. doi:10.4230/LIPIcs.ECOOP.2024.29
[19] Mypy Team. 2025. Mypy. https://mypy-lang.org/. Accessed: 2025-11-22.
[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703 [cs.LG] https://arxiv.org/abs/1912.01703
[21] Adam Paszke and Brennan Saeta. 2021. Tensors Fitting Perfectly. arXiv:2102.13254 [cs.PL] https://arxiv.org/abs/2102.13254
[22] Python Security Team. 2019. Python Taint. https://github.com/python-security/pyt. Accessed: 2025-11-22.
[23] Matthew Rahtz, Pradeep Kumar Srinivasan, Mark Shannon, Alisdair McDiarmid, and Jelle Zijlstra. 2020. PEP 646 – Variadic Generics. https://peps.python.org/pep-0646/. Accessed: 2025-12-04.
[24] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. 2014. PEP 484 – Type Hints. https://peps.python.org/pep-0484/. Accessed: 2025-12-04.
[25] Sahil Verma and Zhendong Su. 2020. ShapeFlow: Dynamic Shape Interpreter for TensorFlow. arXiv:2011.13452 [cs.LG] https://arxiv.org/abs/2011.13452
[26] Dan Zheng and Koushik Sen. 2024. Dynamic Inference of Likely Symbolic Tensor Shapes in Python Machine Learning Programs. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering*

in Practice (Lisbon, Portugal) *(ICSE-SEIP '24)*. Association for Computing Machinery, New York, NY, USA, 147–156. doi:10.1145/3639477.3639718