The Journal of Open Source Software

PureML: a transparent NumPy-only deep learning framework for teaching and prototyping

Yehor Mishchyriak [1]

1 Department of Mathematics and Computer Science, Wesleyan University, Middletown, CT, United States

DOI: 10.21105/joss.09631

Software

- Review
- Repository
- Archive

Editor: Evan Spotte-Smith

Reviewers:

- @sampottinger
- @paquiteau
- @Manishms18
- @yewentao256

Submitted: 02 December 2025
Published: 21 January 2026

Summary

PureML is a compact deep-learning framework implemented entirely in NumPy. It provides a tensor type with reverse-mode automatic differentiation, core neural-network layers and losses, optimizers and learning-rate schedulers, activations, training utilities, and persistence for model and optimizer states. A packaged MNIST dataset makes it easy to benchmark or teach end-to-end (LeCun et al., 1998).

PureML is for learners and instructors who need a small, auditable codebase to illustrate end-to-end training; researchers prototyping or auditing algorithms without the overhead of multi-language stacks; and CPU-only or minimal-dependency environments where installing PyTorch, TensorFlow/Keras, or JAX is impractical.

Statement of need

Modern deep-learning libraries such as PyTorch, TensorFlow/Keras, and JAX provide rich ecosystems but are conceptually and operationally heavy for teaching low-level ML theory, code reading, or CPU-only environments (Abadi et al., 2016; Bradbury et al., 2018; Chollet & others, 2015; Paszke et al., 2019). Pedagogical materials (including standard texts like *Deep Learning* (Goodfellow et al., 2016)) often rely on pseudo-code or small snippets that omit practical details: broadcasting semantics, batching, parameter persistence, computational graph construction, vectorization, gradient accumulation, checkpointing, etc. As a result, learners struggle to bridge the gap to real systems. At the other end of the spectrum, educational projects like `micrograd` (Karpathy, 2020) purposefully keep the scope tiny, and minimalist systems like `tinygrad` (Hotz & contributors, 2024) assume a systems background: `tinygrad` is engineered like a compiler to optimize kernels and targets performance and low-level optimization over didactic readability, effectively teaching how to build PyTorch. Projects like `numpy-ml` (Bourgin, 2019) offer a broad catalog of algorithms implemented in NumPy but are not built around an autodiff engine and are not aimed at performance; they serve as references rather than frameworks for training deep networks.

PureML aims to sit between these extremes: small enough to audit end-to-end, but feature-complete enough for nontrivial models. The code remains transparent while supporting batch-vectorized computation, dynamic computational graphs, forward pass caching, persistence, and related functionality. It focuses on:

- Explicit reverse-mode autodiff with readable vector-Jacobian products (VJPs) for every operation, so gradient flow is inspectable.
- Minimal runtime dependencies (NumPy and zarr) suitable for laptops, classrooms, and CPU-only servers (Harris et al., 2020; Miles et al., 2025).

- Ready-to-run MNIST example to demonstrate end-to-end training without additional downloads (LeCun et al., 1998).
- Persistence utilities that round-trip models, optimizer slots, and data for reproducible exercises or small experiments.
- Intentional trade-offs: no GPU bindings, a single-file autodiff core with explicit VJPs, and an emphasis on readability and auditability. Despite the small surface area, operations are vectorized and efficient relying on NumPy.

Design and implementation

Core autograd. The `Tensor` type wraps NumPy arrays, records operations dynamically, and executes reverse-mode autodiff with explicit VJPs. Broadcasting-aware gradient reduction, slice/advanced indexing support, and graph teardown utilities (`zero_grad_graph`, `detach_graph`) mirror the behaviors found in larger frameworks while remaining short enough to audit. Safe exports (`Tensor.numpy`) discourage in-place mutation of parameter buffers.

Layers and losses. The library supplies `Affine`, `Dropout`, `BatchNorm1d`, and `Embedding` layers. Losses include mean squared error, binary cross-entropy (probabilities or logits), and categorical cross-entropy with optional label smoothing. Stable softmax and log-softmax implementations avoid overflow.

Optimization stack. Optimizers (SGD (Robbins & Monro, 1951) with momentum, AdaGrad (Duchi et al., 2011), RMSProp (Tieleman & Hinton, 2012), Adam/AdamW (Kingma & Ba, 2015)) share a common interface, support coupled or decoupled weight decay, and persist optimizer slots via `save_state`/`load_state`. Lightweight schedulers (step, exponential, cosine annealing (Loshchilov & Hutter, 2016)) operate in-place on optimizer learning rates.

Data utilities and models. A `Dataset` protocol, `TensorDataset`, and `DataLoader` (with slicing fast paths and optional shuffling) simplify input pipelines. The bundled `MnistDataset` streams compressed images/labels from a packaged zarr archive (LeCun et al., 1998). Example models include a small fully connected MNIST classifier (`MNIST_BEATER`) and a classical k-nearest neighbors classifier.

Persistence. The `ArrayStorage` abstraction wraps zarr v3 groups with Blosc compression and can compress to read-only zip archives (Miles et al., 2025). Model parameters, buffers, and top-level literals can be round-tripped to a single `.pureml.zip` file for reproducibility.

Ecosystem and dependencies. PureML requires only NumPy and zarr at runtime (Harris et al., 2020; Miles et al., 2025), targets Python 3.11+, and is distributed on PyPI as `ym-pure-ml`. Logging utilities configure rotating file/console handlers for experiments.

Project structure at a glance (code modules):

```
pureml/
  machinery.py       # Tensor core, autograd graph/VJPs
  layers.py          # Affine, BatchNorm1d, Dropout, Embedding
  losses.py          # CCE, BCE, MSE
  activations.py     # relu, softmax, log-softmax, etc.
  optimizers.py      # SGD, Adam/AdamW, RMSProp, AdaGrad + schedulers
  training_utils.py  # DataLoader, batching/loop helpers
  datasets/
    MNIST/dataset.py # packaged MNIST reader (zarr)
  models/
    neural_networks/mnist_beater.py
    classical/knn.py
  util.py            # ArrayStorage (zarr persistence), helpers
  base.py            # NN base class (save/load, train/eval)
  evaluation.py      # metrics (accuracy)
```

```
  general_math.py    # math helpers
  logging_util.py    # logging setup
```

Quality control

The GitHub repository contains a unit test suite (`tests/`) consisting of 106 tests that cover autograd correctness (elementwise ops, broadcasting, slicing, matmul, reshaping), activation stability, layers and buffers (including bias toggles and seeding), optimizer and scheduler behavior, persistence round-trips, data utilities, and the MNIST dataset/model flow. The suite runs with `python -m unittest discover tests`.

Example usage

```python
from pureml import Tensor
from pureml.activations import relu
from pureml.layers import Affine
from pureml.base import NN
from pureml.datasets import MnistDataset
from pureml.optimizers import Adam
from pureml.losses import CCE
from pureml.training_utils import DataLoader
from pureml.evaluation import accuracy
import time

class MNIST_BEATER(NN):

    def __init__(self) -> None:
        self.L1 = Affine(28*28, 256)
        self.L2 = Affine(256, 10)

    def predict(self, x: Tensor) -> Tensor:
        x = x.flatten(sample_ndim=2) # passing 2 because imgs in MNIST are 2D
        x = relu(self.L1(x))
        x = self.L2(x)
        if self.training:
            return x
        return x.argmax(axis=x.ndim-1) # argmax over the feature dim

with MnistDataset("train") as train, MnistDataset("test") as test:
    model = MNIST_BEATER().train()
    opt = Adam(model.parameters, lr=1e-3, weight_decay=1e-2)
    start_time = time.perf_counter()
    for _ in range(5):
        for X, Y in DataLoader(train, batch_size=128, shuffle=True):
            opt.zero_grad()
            logits = model(X)
            loss = CCE(Y, logits, from_logits=True)
            loss.backward()
            opt.step()
    end_time = time.perf_counter()
    model.eval()
    acc = accuracy(model, test, batch_size=1024)
print("Time taken: ", end_time - start_time, " sec.")
```

```
print(f"Test accuracy: {acc * 100}")
```

Example usage in computational biology

SAWNERGY project builds its skip-gram embedding pipeline for amino acid interaction networks using PureML (link).

Availability

Source code is available at https://github.com/Yehor-Mishchyriak/PureML.
The package is published on PyPI at https://pypi.org/project/ym-pure-ml/.
Documentation is available at https://ymishchyriak.com/docs/PUREML-DOCS.
The license is Apache-2.0.

Future directions

Planned extensions include convolutional, recurrent, and message-passing layers, attention mechanisms, additional activation and loss functions, richer evaluation metrics, and related tooling to support a broader range of deep-learning experiments.

Acknowledgements

I am grateful to Professor Kelly M. Thayer (Wesleyan University) for guidance and constructive feedback on this project. I also acknowledge the authors of the *Deep Learning* textbook, which informed the design and pedagogy of PureML (Goodfellow et al., 2016). This work was supported by the National Science Foundation under Grant No. CHE-2320718.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., & others. (2016). TensorFlow: A system for large-scale machine learning. *OSDI*. https://doi.org/10.48550/arXiv.1605.08695

Bourgin, D. (2019). *Numpy-ml*. https://github.com/ddbourgin/numpy-ml

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.3.13). http://github.com/jax-ml/jax

Chollet, F., & others. (2015). *Keras*. https://keras.io.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, *12*, 2121–2159. http://jmlr.org/papers/v12/duchi11a.html

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

Harris, C. R., Millman, K. J., Walt, S. J. van der, & others. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Hotz, G., & contributors. (2024). *Tinygrad*. https://github.com/tinygrad/tinygrad

Karpathy, A. (2020). *Micrograd*. https://github.com/karpathy/micrograd

Kingma, D. P., & Ba, J. (2015). *Adam: A method for stochastic optimization*. https://doi.org/10.48550/arXiv.1412.6980

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324. https://doi.org/10.1109/5.726791

Loshchilov, I., & Hutter, F. (2016). *SGDR: Stochastic gradient descent with warm restarts*. https://doi.org/10.48550/arXiv.1608.03983

Miles, A., Kirkham, J., Stansby, D., Papadopoulos Orfanos, D., Hamman, J., & others. (2025). *Zarr-developers/zarr-python: v3.1.5* (Version v3.1.5). https://doi.org/10.5281/zenodo.17672242

Paszke, A., Gross, S., Massa, F., & others. (2019). PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, *32*. https://doi.org/10.48550/arXiv.1912.01703

Robbins, H., & Monro, S. (1951). A stochastic approximation method. *The Annals of Mathematical Statistics*, *22*(3), 400–407. https://doi.org/10.1214/aoms/1177729586

Tieleman, T., & Hinton, G. (2012). *Lecture 6.5 - rmsprop: Divide the gradient by a running average of its recent magnitude*. Coursera: Neural Networks for Machine Learning. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf