Evaluating Object-Oriented Best Practices in Deep Learning and Neuroimaging

Sierra Reschke, Nolan Brady

[1]University of Colorado Boulder
Boulder, CO

Abstract

The rapid development of deep learning continues to push the boundaries of model complexity and performance, yet this progress often comes with challenges in software sustainability and reproducibility. As models grow more sophisticated, the lack of structured, modular code becomes a significant barrier to collaboration, code reuse, and long-term maintenance. This paper presents object-oriented programming (OOP) as a robust solution for organizing deep learning code in a way that promotes clarity, scalability, and efficiency. By analyzing OOP implementations in frameworks like PyTorch, we highlight how principles such as inheritance, encapsulation, and design patterns contribute to code organization and usability. We further discuss how the adoption of these principles aligns with the need for testing frameworks in research, ultimately bridging the gap between advanced model design and accessible, reproducible code. Our findings encourage deep learning practitioners to leverage OOP practices as a foundation for creating sustainable, flexible, and collaborative research environments.

Introduction

Deep learning research increasingly faces a critical challenge: as models grow in complexity, their implementations often become unwieldy and difficult to reproduce. While groundbreaking architectures continue to advance the field, the supporting code bases frequently fall short of software engineering best practices, creating barriers to collaboration, code reuse, and future development (Liao and Poggio 2017). Traditional approaches of publishing single-file implementations alongside academic papers are no longer sufficient for today's sophisticated models, which often involve intricate data pre-processing, complex training pipelines, and elaborate evaluation procedures.

Object-oriented programming patterns (OOP) offer a systematic solution to these challenges. By leveraging core principles researchers can create more maintainable and extensible implementations. These patterns are particularly valuable in deep learning, where clear separation of concerns between model architecture, data handling, and training logic is essential. Furthermore, OOP's natural alignment with test-driven development addresses another critical gap in deep learning research: the widespread lack of comprehensive testing frameworks.

The deep learning community stands to gain significant benefits from adopting OOP best practices. Modern frameworks like PyTorch and TensorFlow already incorporate object-oriented principles in their core design, providing a foundation for researchers to build upon (Abadi et al. 2016; Subramanian 2018). This paper examines how systematic application of OOP patterns can enhance code quality, reproducibility, and collaboration in deep learning research, while maintaining the flexibility needed for rapid experimentation.

Object-oriented programming in Deep Learning Frameworks

Deep learning in both academia and industry has been largely carried out using one of two frameworks: the Google developed package TensorFlow, and PyTorch which came out of Facebook's R&D program. These two frameworks, while having been developed in isolated environments, settled in very similar architectures and implementation decisions. Outside of mathematical or hardware considerations, the programming interface for both frameworks is firmly based in object oriented methods. As both projects have incredible levels of complexity it seems the object-oriented paradigm has helped minimize user frustration while maintaining the highest degree of code quality and readability possible. In this section we will take a deeper look into the decisions made specifically by PyTorch in their package implementation to try to understand how various object-oriented practices have helped the package functionality expand. In this evaluation we will focus primarily on: inheritance, encapsulation, polymorphism, Builder patterns and Factory patterns. These patterns are at the core of object-oriented programmings mission to minimize complexity and therefore perfect for analyzing the structure of a complicated software product.

Inheritance

Inheritance plays a large role in how PyTorch seems to have organized its code. The top level class, called nn.Module handles a majority of the logistic information that would be involved in all deep learning processes. This includes managing interaction with hardware, toggling various training

modes and tuning parameters. Other classes inherit from the nn.Module class in order to build on it's fundamental capabilities, this includes parts of the code such as the forward method implemented in all deep learning projects as well as other pre-built modules such as convolutional layers and recurrent layers (PyTorch 2024d). Another example of inheritance that stands out in the PyTorch implementation is the further inheritance between similar classes. Take for instance the Convolution layer, this layer is inherited from all other convolutional layers such as Conv1d, Conv2d and Conv3d (PyTorch 2024c).

The implementation of inheritance within the PyTorch code base could have many potential benefits, these range from code reuse, simpler user interfacing, and avoiding package maintenance bloat. In terms of user interfacing, the inheritance patterns allow for certain niceties when using the package. One such form of inheritance that was previously mentioned above is the forward pass method. A common behavior and root to that method prevents implementation issues between various class implementations. This also improves the developer experience by allowing for a more modular construction of the deep learning pipelines needed. Since larger deep learning implementations are notoriously challenging from an architectural perspective, this modularity reduces a lot of friction that could be experienced otherwise.

Encapsulation

In PyTorch, encapsulation is achieved through the use of classes and modules, primarily leveraging torch.nn.Module. This base class allows developers to encapsulate complex model architectures, methods, and operations into cohesive, reusable units. When building a neural network, for instance, layers and their respective weights are encapsulated as internal attributes within the model class, shielding them from direct access. Instead, methods such as model.parameters() allow the user to access the weights while keeping the internal implementation hidden (PyTorch 2024d).

Encapsulation in PyTorch is also evident in how it handles data loading through the Dataset and DataLoader classes. The Dataset class provides an abstraction for loading and accessing data, while DataLoader efficiently handles the creation of subsets and shuffling of this data during training (PyTorch 2024a). By encapsulating data-related operations, these classes separate the model architecture from data handling concerns, promoting cleaner, more modular code. This abstraction allows users to focus on defining their neural networks without needing to manage low-level data retrieval, which enhances code readability and reusability.

Polymorphism

In PyTorch, polymorphism is a fundamental concept facilitated through the torch.nn.Module class. Every model created in PyTorch is an instance of a class that inherits from torch.nn.Module, which provides a standardized interface for defining and training neural networks. To implement specific behaviors in custom models or layers, developers override the forward() method inherited from the base class (PyTorch 2024d; GeeksforGeeks 2024). This allows each derived class to define its unique computation while maintaining compatibility with the overall model and code. By adhering to this polymorphic structure, users can integrate custom layers into existing frameworks, ensuring that models can be trained and evaluated consistently without modifying the underlying code architecture.

Builder Pattern

The Builder pattern is conceptually reflected in PyTorch's torch.nn module, emphasizing modularity and flexibility in constructing neural network architectures. In PyTorch, layers are defined as classes that inherit from torch.nn.Module, allowing users to compose complex models through a structured approach (PyTorch 2024b). For instance, the torch.nn.Sequential class enables users to stack layers when constructing a model(PyTorch 2024e). This design allows for the construction of specialized layers while maintaining a separation between model building and its implementation.

Factory Pattern

Factory patterns are another place where PyTorch makes use of object-oriented best practices. This takes many forms, but it becomes most apparent in the device creation logic and in the functional module. The device creation method is used in almost all deep learning projects. Since there is a large difference between running and accelerating code on GPU, CPU and TPU there needs to be a way to differentiate between the various hardware. The device method handles this large amount of complexity with a simple call that takes in hardware specifications and configures the PyTorch implementation accordingly to work on the specified hardware. Within the functional module of PyTorch the torch.nn.functional package allows for the easy generation of several different functional layers without the need for importing their torch classes manually. The same is done to declare the optimizer that is being used. To specify the optimizer you use the torch.nn.optim module and call the optimizer you need (PyTorch 2024f).

The use of factory patterns within PyTorch allows for a more pleasant user experience from the perspective of the developers. In the case of the device method, it allows for someone to reliably and reproducibly declare various machine types without needing to worry about class implementations. This becomes especially important when the hardware used for testing and/or training a model is different from the hardware being used for deployment or fine tuning.

Object-oriented Programming in Research

Deep learning research, particularly at its intersection with neuroimaging, faces numerous challenges. In this section, we will first outline the key issues that hinder progress in this field and then discuss how object-oriented design patterns can provide solutions.

Current Issues

Data Processing One of the primary challenges in both deep learning and neuroscience research is data processing.

Both fields rely heavily on large, well-structured datasets to train models.

From a traditional deep learning perspective, datasets must align with the specific parameters of the model. For example, if a model expects an input image of size (512, 512, 3) but receives an improperly formatted image, the model will fail. Even more problematic is when the dataset meets technical requirements but violates implicit assumptions about its structure. Consider a (268, 268) adjacency matrix: transposing the matrix would not cause the model to fail but would misalign the orientation, leading to edges falling in incorrect positions. Such errors can result in hours of wasted training time, especially with large models.

Data processing in neuroimaging is even more complex. Neuroimaging data is typically provided in raw format directly from the scanner, often amounting to several terabytes of 3D images paired with behavioral or physiological measurements. Subtle issues, such as barely noticeable inversions in images, can complicate analysis. These challenges necessitate highly specialized preprocessing pipelines for each study. Common problems include off-by-one errors between brain images and phenotypic data, or subtle, imperceptible data mutations. Moreover, neuroimaging datasets are frequently processed for multiple phenotypes and analysis methods, often leading to version control issues down the line.

Data Analysis As mentioned earlier, both deep learning and neuroimaging often involve multiple analyses on a single dataset. In deep learning, this typically involves evaluating multiple models on various datasets in a pairwise manner. This approach is common for testing the limitations of models and their ability to solve different problems. In neuroimaging, this often takes the form of applying several preprocessing pipelines to a single dataset.

In both cases, the iterative nature of these workflows can lead to complications. Quick fixes to adapt to different paradigms can create unmanageable and error-prone codebases. This becomes especially problematic when workflows involve remote servers. Both deep learning and neuroimaging require significant computational resources, which are often accessed on high-performance computing clusters. Complex or poorly structured code can obscure potential issues, making debugging on remote systems difficult and expensive.

Research Collaboration Perhaps the most significant barrier lies in collaboration. Much of the progress in deep learning stems from building on previous models or datasets. This workflow often involves downloading code repositories from GitHub, replicating results on the same dataset, and adapting the model for new tasks or datasets.

While straightforward in theory, this process frequently devolves into weeks of debugging due to incomplete or poorly documented code, unresolved dependency conflicts, or missing files. In some cases, the effort to use a codebase proves fruitless, with the model rendered unusable. These issues waste valuable time and hinder progress.

Solutions to Common Issues

While object-oriented programming (OOP) is not a drop in solution for all problems, it can serve as an effective buffer against these challenges. Simply adopting an object-oriented mindset at the outset of a project can prevent many issues from arising later on. In the following sections, we will explore how the design patterns discussed earlier can be leveraged to address the problems outlined above.

Builder pattern The builder pattern provides a structured way to construct complex objects such as data preprocessing pipelines and more complicated machine learning models, breaking the process into clear, manageable steps. For data preprocessing, the builder pattern allows users to sequentially add or modify steps such as .removeOutliers(), .normalize(), or .augment(). For model generation, this pattern ensures clarity and flexibility by allowing researchers to iteratively add layers, activation functions, and other components.

This approach is especially valuable when dealing with varying requirements across datasets or experiments. By encapsulating the construction logic, the builder pattern makes it easy to debug, extend, and adapt pipelines to different scenarios. It also promotes collaboration, as builders can be shared or modified independently without affecting other components.

Factory pattern The factory pattern provides a systematic way to standardize and simplify the creation of preprocessing pipelines and models in deep learning workflows. This approach is particularly beneficial for ensuring reproducibility and flexibility during experimentation. By centralizing the logic for creating preprocessing steps and models, the factory pattern reduces redundancy, facilitates testing of different combinations, and ensures consistency across multiple datasets or model configurations.

For example, a Data Factory can generate preprocessing pipelines for different datasets, enabling rapid switching between configurations. Similarly, a Model Factory can produce architectures tailored to specific tasks, allowing researchers to test combinations of hyperparameters and layers systematically. This modularity improves the efficiency of experimentation and collaboration, as standardized configurations can be easily shared and reused.

Inheritance Inheritance is a cornerstone of object-oriented programming that promotes code reuse and simplifies the extension of existing structures. This principle can address many of the challenges in deep learning and neuroimaging workflows and collaboration by establishing hierarchical relationships between classes, enabling the creation of specialized versions of existing components without exposing the underlying complexity.

Inheritance allows developers to create a base class for data preprocessing pipelines that includes common functionality, such as data cleaning, normalization, and augmentation. Subclasses can extend the base class to address dataset-specific requirements. For instance, a NeuroimagingPipeline class can inherit from a general DataPipeline

class, adding specialized steps like alignment of brain images or voxel intensity normalization. This approach avoids duplicating shared logic while enabling flexibility for handling domain-specific tasks.

For iterative workflows that involve multiple analyses, inheritance simplifies the management of diverse preprocessing pipelines or model configurations. A base class, such as BaseAnalysis, can encapsulate common analysis methods, while subclasses like ModelComparisonAnalysis or PipelineEvaluationAnalysis implement unique behaviors. This structure ensures consistency across different analysis types while providing room for tailored functionality.

In collaborative projects, inheritance can facilitate modular and extendable codebases, reducing the likelihood of dependency conflicts or missing files. For example, a shared ModelBase class can define foundational behaviors for deep learning models, such as loading weights or evaluating performance, while specialized subclasses like ImageClassificationModel or SegmentationModel define task-specific logic. This hierarchical organization makes it easier for collaborators to adapt the codebase to new datasets or tasks without introducing breaking changes.

Encapsulation Encapsulation enhances modularity and robustness in codebases by bundling data and methods within a class and restricting access to implementation details. Furthermore, by defining clear interfaces and hiding internal complexities, encapsulation helps mitigate common issues related to debugging, extensibility, and code clarity.

Encapsulation ensures that preprocessing steps and data transformations are contained within well-defined classes or methods. For example, a DataPreprocessor class might expose public methods such as .fit() and .transform(), while internal details, like the exact algorithms used for data augmentation or normalization, are hidden. This abstraction prevents accidental modifications to critical processing logic, reducing errors such as off-by-one misalignments or subtle data mutations.

Encapsulation is particularly valuable in iterative workflows that involve running multiple models or pipelines. Encapsulating each analysis step within a class ensures that intermediate results, configurations, and metadata are securely stored and accessible through a controlled interface. For instance, an AnalysisRunner class could encapsulate methods for logging, error handling, and resource management, preventing issues related to unstructured or duplicated code when adapting workflows for high-performance computing environments.

Encapsulation facilitates collaboration by defining clear boundaries between components. A shared ModelFactory class, for example, can encapsulate the logic for generating model architectures, exposing a simple interface for collaborators to specify hyperparameters or layer configurations. Internally, the factory ensures that all generated models conform to predefined standards, eliminating the risk of inconsistencies when sharing codebases across teams.

Testing in Deep Learning Testing is an often underutilized tool in deep learning, and it is even less commonly applied in neuroimaging codebases. Although not inherently a feature of object-oriented programming, testing and test-driven development (TDD) provide similar benefits to the object-oriented patterns discussed earlier. Testing can address many, if not all, of the issues highlighted in this work (Wang et al. 2024).

In data processing, testing can ensure that no unintended mutations occur during preprocessing steps. For instance, as mentioned earlier, matrices often need to be transposed to meet the requirements of different software packages. Without careful attention, this can result in an incorrectly transposed matrix being passed into a model. By incorporating tests, researchers can verify matrix orientation both before and after the preprocessing pipeline, preventing such errors.

Similarly, testing can ensure the proper alignment of imaging and phenotypic data when they are processed in parallel. This prevents critical issues such as off-by-one errors during model training, which can compromise results.

Perhaps most importantly, widespread adoption of testing could significantly enhance the efficiency of research collaborations. Instead of spending weeks replicating results, researchers could use a test suite to quickly validate that code and data are correctly set up. This would allow collaborators to focus their time on evaluating promising models, rather than troubleshooting misaligned datasets or flawed pipelines.

Explanation of Code

We designed our code to demonstrate how various design patterns can streamline the research process for deep learning and neuroimaging. This project focuses on the patterns we considered most relevant: Builder, Factory, Inheritance, and Strategy.

Builder

We implemented the Builder pattern to showcase two distinct use cases.

The first use case involves real signal preprocessing, addressing a critical issue mentioned earlier. In this implementation, the Builder pattern facilitates a clean and efficient preprocessing pipeline for time-series data. A key advantage of this pattern is the modularity of its components. Since the methods in the Builder pattern are not interdependent, preprocessing steps can be easily added or removed. This flexibility is crucial for fine-tuning preprocessing strategies, as balancing noise removal with signal preservation often requires iterative testing. By allowing rapid experimentation with different combinations of preprocessing methods, this implementation ensures an improved signal-to-noise ratio through effective interventions.

The second use case involves using the Builder pattern to simulate synthetic data, which is vital for benchmarking complex models. Controlling data properties in synthetic datasets enables researchers to explore causal relationships between signal features and model performance. By offering easy manipulation of the data, this approach significantly reduces the time required between experiments. These two examples highlight the potential of the Builder pattern to enhance both deep learning and neuroimaging research workflows.

Factory

We used the Factory pattern to demonstrate its utility in projects requiring multiple models. While our implementation is relatively simple, it illustrates the benefits of this pattern in larger projects with more parameters. The Factory pattern allows models to be initialized outside the main analysis script, reducing clutter and improving clarity.

In our code, we demonstrate the ease of initializing a new model with a single line of code using the Factory pattern. This approach is particularly beneficial when multiple datasets are involved, as datasets can be called in a similarly streamlined manner. By avoiding complex logic in the main analysis script, this method minimizes confusion during experiments and reduces the likelihood of errors. Additionally, even though the models in this example are fundamentally different, their initialization processes are similar, improving the overall workflow for researchers.

Inheritance

We showcase the use of inheritance in this project by demonstrating how it can reduce redundant code in similar model architectures. We demonstrated this by implementing a base 'AutoEncoder' class and extended it to create a 'VariationalAutoEncoder' class. Since these architectures share significant overlap, using inheritance avoids duplicating code. This approach also reduces the risk of errors, such as updating parameters in one architecture but forgetting to update them in another. This logic can be taken even one step further, and custom layers can be inherited for various model architectures as drop in solutions. For instance if you need an auto-regressive self-attention layer you can build a base layer that can then be used in any new model. This not only ensures dryer code but also increases development time.

Conclusion

As problems become more complex and collaboration among individuals increases, the need for greater cohesion in organizing these efforts becomes paramount. Object-oriented programming emerged to address this need for order within large codebases. The field of deep learning is currently at an interesting inflection point, where the complexity of problems and the number of people working on them have grown exponentially. Despite this, general software engineering best practices have yet to become commonplace in the deep learning community.

This paper explores the various ways object-oriented principles have already been applied to the organization of large deep learning frameworks, such as PyTorch. Understanding these implementation decisions is crucial to maximizing the utility of these frameworks. Additionally, we discussed how object-oriented design can be leveraged in deep learning and related fields, such as neuroimaging, to streamline workflows in both research and industry.

Our provided code examples illustrate these suggested implementations using a realistic yet simplified codebase, demonstrating how each individual pattern contributes to cleaner, more maintainable, and more readable code. Overall, we hope to see increased adoption of object-oriented patterns in deep learning and neuroimaging projects moving forward.

References

Abadi, M.; Abadi, M.; Barham, P.; Barham, P.; Chen, J.; Chen, J.; Chen, Z.; Chen, Z.; Davis, A.; Davis, A.; Dean, J. M.; Dean, J.; Devin, M.; Devin, M.; Ghemawat, S.; Ghemawat, S.; Irving, G.; Irving, G.; Irving, G.; Isard, M.; Isard, M.; Kudlur, M.; Kudlur, M.; Levenberg, J.; Levenberg, J.; Monga, R.; Monga, R.; Moore, S.; Moore, S.; Murray, D. G.; Murray, D. G.; Steiner, B.; Steiner, B.; Tucker, P. A.; Tucker, P. A.; Vasudevan, V.; Vasudevan, V. K.; Warden, P.; Warden, P.; Wicke, M.; Wicke, M.; Yuan, Y.; Yu, Y.; Yu, Y.; Yu, Y.; Zheng, X.; and Zheng, X. 2016. TensorFlow: a system for large-scale machine learning. *USENIX Symposium on Operating Systems Design and Implementation*, 265–283. ARXIV$_ID$: 1605.08695$MAGID$: 240214481$1S2ID$: 4954fa180728932959997a4768411ff9136aac81.

GeeksforGeeks. 2024. Understanding the Forward Function Output in PyTorch. Accessed: 2024-10-25.

Liao, Q.; and Poggio, T. 2017. *Object-Oriented Deep Learning*. Accepted: 2017-10-31T23:45:35Z.

PyTorch. 2024a. *Datasets & DataLoaders Tutorial*. PyTorch. Accessed: 2024-10-25.

PyTorch. 2024b. *torch.nn Documentation*. PyTorch. Accessed: 2024-10-25.

PyTorch. 2024c. *torch.nn.Conv1d Documentation*. PyTorch. Accessed: 2024-10-25.

PyTorch. 2024d. *torch.nn.Module Documentation*. PyTorch. Accessed: 2024-10-25.

PyTorch. 2024e. *torch.nn.Sequential Documentation*. PyTorch. Accessed: 2024-10-25.

PyTorch. 2024f. *torch.optim — PyTorch Documentation*. PyTorch. Accessed: 2024-10-25.

Subramanian, V. 2018. *Deep Learning with PyTorch: A practical approach to building neural network models using PyTorch*. Packt Publishing Ltd. ISBN 978-1-78862-607-1. Google-Books-ID: DOlODwAAQBAJ.

Wang, H.; Yu, S.; Chen, C.; Turhan, B.; and Zhu, X. 2024. Beyond Accuracy: An Empirical Study on Unit Testing in Open-source Deep Learning Projects. *ACM Transactions on Software Engineering and Methodology*, 33(4): 1–22. ArXiv:2402.16546 [cs].