Comparative Analysis of OOP Patterns in Modern ML Frameworks

Sarthak Paithankar, Sawyer McKenney
Sarthak.Paithankar@colorado.edu, Sawyer.McKenney@colorado.edu

Abstract

The rise of AI and Machine Learning models in the modern day is rapidly changing how the software landscape looks, and best practices must be established to maintain these systems. This paper examines how object-oriented programming (OOP) and functional programming (FP) influence the architecture of modern machine learning frameworks. We explore how object-oriented design allows models to be modular but obscure clarity and how functional programming makes some models more performant at the cost of modularity. Both approaches have their limitations, which often limit developers from creating scalable and performant models. Libraries such as Equinox attempt to solve this problem but more research and development in these libraries is needed before they can be used in production.

Introduction

As AI and machine learning rapidly evolve, design maintainability and modularity will become areas of great concern in these fields. Algorithmic innovations have advanced models to be more efficient and accurate, but the architecture of these systems remains a critical part in the maintainability of these complex code bases. Picking an approach that provides optimal speeds, clarity, and modularity are areas that should be more diligently researched. Object-oriented programming (OOP) and functional programming (FP) are still two dominant concepts that play a large role in shaping modern ML frameworks.

- **Object-Oriented Programming (OOP)** — used by frameworks such as PyTorch and scikit-learn.
- **Functional Programming (FP)** — used heavily in JAX and its supporting ecosystem.

Frameworks like PyTorch, have an object-oriented approach, and ensure reusability using patterns like Factory and Builder(5). On the other hand, JAX, which is built around functional programming principles, uses immutability and favors performance. This paper will conduct a comparative analysis of how OOP design patterns influence modularity, clarity, and performance in ML frameworks, using PyTorch and JAX.

OOP in ML

Both object-oriented programming (OOP) and functional programming (FP) are two distinguished approaches when it comes to building complex software systems. OOP is built around the idea of objects that are self-contained pieces of data that when combined create a structure and behavior. By using encapsulation, inheritance, and polymorphism (8). In ML frameworks, OOP provides several benefits:

- **Modularity**: individual model components (layers, optimizers, datasets) are encapsulated in classes.
- **Reusability**: inheritance enables new architectures to be built from existing components.
- **Flexibility**: model behavior can be modified by overriding class methods.

These development tools give developers the freedom to make systems so that individual components can be modified or extended without changing the entire code base. In ML, this allows for fast changes and deployments as well as the reuse of model architectures. We can use the Sci-Kit learn library as an example of this. The library utilizes OOP principles such as polymorphia and inheritance to implement models which user can create instances of to manipulate data(Koculu). On the other hand, functional programming organizes code around functions and focuses on constancy and clarity. FP emphasizes:

- **Immutability**,
- **Explicit parameter passing**,
- **Absence of side effects**.

This minimizes error and improves clarity but, in many cases, it makes it harder to manage different components and make quick changes.

Issues with Clarity

Examples of how a lack of clarity can cause issues in a ML system can be seen through IBM's Watson project. Watson was project was aimed to be a revolutionary tool in the healthcare industry with the aim of providing high quality health care without the need of physicians. Though this idea was revolutionary at the time and Watson proved to have great performance in testing it failed to provide the same accurate results out in the field. According to Prahlada, Watson's

- **opaque decision-making processes,**
- **undocumented model behaviors,**
- **poor ability to trace model reasoning,**

caused the project to be a bust in the medical field (3). Though functional programming wouldn't have been the saving grace for Watson it could have possibly provided some insight to Watson's reasoning and gave the model a fighting chance to prove its abilities out in industry.

Issues with Model Drift

Another equally critical challenge faced by ML models is model drift, which is best managed by designing modular systems. Model drift occurs when a model is unable to adapt to new trends in data which is quiet common in industries such as e-commerce and finance. Fashion is one of the industries that deals with this issue as fashion trends, demand, and seasons make the industry volatile. Large retailer must constantly adapt their models to recommend and stock popular items to keep profits high (4). To keep models adaptable developers, implement OOP principle which allow models to be easily modified. Designing models with OOP principles at mind also allows models to be scalable as more efficient models are build and need to be integrated into existing environments (1).

Code Analysis

Many deep learning frameworks opt for an object-oriented design, but JAX chooses a functional design paradigm. JAX relies mainly on pure functions meaning that parameters must be explicitly passed as inputs to the function. This functional design allows JAX to leverage powerful transformations like jit, vmap and grad, which in turn allows for more efficient compilation (Patrick Sutanto). As JAX promotes clarity its limitations are also evident due to it avoiding object-oriented abstractions. In traditions OOP specifically here, we will be addressing PyTorch, developers encapsulate layers using classes like nn.Module that allows for the ability to push parameters updates. In JAX this is not possible since parameters must be stored in immutable data structures and passed manually between separate functions, which can make it extremely difficult and complicated in large-scale projects, hindering the maintainability of the architecture as parameter are added and it continues to grow.

Example Code

Below we will show an example of how object-oriented and functional programming are different in practice when implementing the discussed above machine learning frameworks.

PyTorch (OOP) Implementation

On the right side is the PyTorch (OOP) implementation of a very simple neural network encapsulated with a SimpleExample class that inherits from nn.Module. The parameters inherited are fc1 and fc2, and these are then stored as class attributes, and the forward method defines how data flows through the neural network. In this specific example, it is very easy to add new layers by modifying the class definition without changing how the model is called. Making PyTorch a great choice for large projects that require modularity and clear separation of different components.

```jax.py
import jax
import jax.numpy as jnp
from jax import random, grad, jit

def init_params(key):
    w1_key, w2_key = random.split(key)
    return {
        "w1": random.normal(w1_key, (10, 5)),
        "w2": random.normal(w2_key, (5, 1)),
    }
def forward(params, x):
    x = jnp.dot(x, params["w1"])
    x = jnp.maximum(x, 0)
    return jax.nn.sigmoid(jnp.dot(x, params["w2"]))
key = random.PRNGKey(0)
params = init_params(key)
output = forward(params, jnp.ones((1, 10)))
print(output)
```

```PyTorch.py
import torch
import torch.nn as nn
import torch.nn.functional as F

class SimpleExample(nn.Module):
    def __init__(self):
        super(SimpleExample, self).__init__()
        self.fc1 = nn.Linear(10, 5)
        self.fc2 = nn.Linear(5, 1)
    def forward(self, x):
        x = F.relu(self.fc1(x))
        return torch.sigmoid(self.fc2(x))
model = SimpleExample()
output = forward(params, random.normal(key, (1, 10)))
print(output)
```

Figure 1: Comparison of equal neural network implementations in JAX and PyTorch

JAX (Functional) Implementation

JAX on the left side performs the same computation as the PyTorch example but instead uses pure functions. The parameters are created and then stored in a dictionary and then passed explicitly to the forward function with the input data. This approach, unlike the PyTorch example, avoids side effects, and as a result the output depends only on the input arguments. Each transformation is easier to trace here since it is explicit. But this means that parameter updates and state updates must be handled manually by the developers. This can make larger projects harder to maintain if the project continues to grow or requires frequent modifications. Some industries such as finance, e-commerce, and healthcare require this type of clarity to adapt to ever changing consumer behavior and provide reasoning behind decisions.

Debugging Methodologies

Debugging JAX

Though JAX is more performant than Pytorch, JAX's compiler optimizations make it harder to debug than Pytorch. JAX uses just in compilation and converts your codes to a set of tracers. These tracers symbolize your code and are used to create a computation graph that is handed to XLA, a machine code compiler (10). The computation graph handed to XLA only contains numerical computations so XLA effectively removes debug statements to optimize your code. This process improves performance, but it also means that the code being executed is not exactly what you wrote within your function and debug statements will not appear in your console. JAX attempts to provide a solution to this with library debug functions such as (jax.debug.print()), but they are restrictive making debug difficult(11).

Debugging Pytorch

Pytorch code doesn't natively utilize XLA so it behaves as normal python code and debugs are logged as expected. Pytorch uses a dynamic computational graph which is built as the code is executed in contrast to JAX which builds it computation graph first. This allows debug statement to output

actual information rather than outputting cryptic data types as they would with JAX (13). Since the graph is built dynamically you can also use debuggers to step through functions and be able to see data inside variables rather than the debugger outputting data types. These debugging quirks are important to note as debugging gets increasingly complex as models become larger and more complex.

Possible Solution Libraries

Since modularity and performance are major concerns in many systems today, developers have started trying to create libraries that can provide both. Equinox is a library that utilizes Pytrees to mirror OOP principles within JAX. Pytrees can be used to provide to mirror encapsulation within JAX which allows code to be more modular and maintainable while providing the performance of JAX (Equinox). Although this library is small it has already multiple libraries iterated from it.

```
class JAXMLP(eqx.Module):
    fc1: eqx.nn.Linear
    fc2: eqx.nn.Linear

    def __init__(self, in_dim, width, out_dim, key):
        k1, k2 = jax.random.split(key, 2)
        object.__setattr__(self, "fc1", eqx.nn.Linear(in_dim, width, key=k1))
        object.__setattr__(self, "fc2", eqx.nn.Linear(width, out_dim, key=k2))

    def __call__(self, x):
        x = jax.nn.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Figure 2: Jax W/ Equinox Implementation

```
class TorchMLP(nn.Module):
    def __init__(self, in_dim, width, out_dim):
        super().__init__()
        self.fc1 = nn.Linear(in_dim, width)
        self.fc2 = nn.Linear(width, out_dim)

    def forward(self, x):
        return self.fc2(torch.relu(self.fc1(x)))
```

Figure 3: Pytorch Implementation

Batch Size	JAX (s)	PyTorch (s)
2048	0.002488	0.002362
4096	0.004571	0.004301
8192	0.007297	0.008198
16384	0.011115	0.011624
32768	0.018249	0.020284
65536	0.036951	0.038342

Figure 4: Results

Analysis of Nueral Network Performance

Above you can see two implementations of a neural network. One is implemented with the Pytorch library and the other with Equinox. We can see that both utilize classes, but Equinox implements them in such a way that the model objects are immutable. This allows equinox to utilize highly optimized JAX functions while also presenting them in an object-oriented manner. To test the efficiency of these implementations, we ran through a series of training cycles with different workloads. Each batch was filled with random noise data and the goal was to see how long the model would take to do a forward and backward propagation. The result of running these training cycles tells us that JAX is more effective at running heavy neural network workloads than Pytorch, but Pytorch has a significantly lower overhead. The JAX model outperformed the Pytorch model as batch sizes grew past 8192 packets. These computations were performed on a simple linear model, but as models become more complex, the speed up that JAX provides will be greater. It is important to note that these result were obtained by running the nueral network in a virtual Google Colab environment using a v5e-1 TPU. JAX and the v5e-1 TPU are custom Google products while Pytorch is a Meta product. Result may vary if other GPUs or environments are utilized to conduct this test.

Architectural Evaluation of Equinox

Equinox's hybrid design allows for some of the shortfalls of both OOP and FP to be addressed. Deep Nueral Networks can implement Equinox to allow for an encapsulated design using JAX's Pytrees while also retaining the performance JAX has to offer. By using Equinox, we can keep models modular so implementing changes to the models is easier which addresses issues such as model drift while also allowing models to become more efficient.

Since Equinox is a JAX library it faces the same debugging challenges and limitations that JAX is subject to. The abstraction made within equinox make debugging error more cryptic than the error seen within simple JAX implementations. The creators of Equinox have tried remedying this limitation by adding library debug functions to supplement the debug functions that already exist within JAX (12).

Equinox cannot fully mimic an object-oriented language which is still widely used in industry today. Equinox is proof that existing libraries such as JAX can be used to make machine learning object oriented and fast, but more research and development is needed to make them usable in real world applications.

Comparative Evaluation of Design Patterns

Object oriented and functional programming approach machine learning from different view point, but both rely on shared design principles that guide how these models are built, trained, and maintained. By looking at and comparing how different software design patterns appear across PyTorch, JAX, and Equinox we gain a clearer understanding of the tradeoffs between modularity, clarity and overall performance. There are two patterns in particular that stand

out here, Factory, and Builder pattern. Both highlight how these different and distinct frameworks structure computations, organize their components, and expose functionality to developers.

Factory Pattern

PyTorch frequently uses the Factory pattern to make it easy to create layers and optimizers which are core components of training neural networks. In Figure 2 and Figure 3 the component `nn.Linear` handles parameter initialization, so developers can build models without worrying about the low level setup. In figure 2 you observe how JAX does not have this parameter built in and it had to be created manually through two pure functions. This gives the developer more transparency to what is happening but it becomes harder to manage as the model grows. Equinox works at the middle ground by using a Pytree, a nested tree structure of Python containers, allowing for ease of parameter handling, giving JAX a factory like abstraction while not dismantling its functional structure.

Builder Pattern

PyTorch's training workflows reflects the Builder pattern, where the model, data loader, optimizer and schedulers are assembled piece by piece to form a complete training pipeline (0). The model class shown in figure 3 is only one component of this pipeline. JAX on the other hand avoids this pattern and instead makes use of function based compositions, giving the developer full visibility into the computations but this requires a larger template. Equinox offers an intermediate solution by allowing developers to organize their models in an object oriented structure while still using JAX's functional transformations that get applied at run time.

Summary

Both the Factory and Builder patterns illustrate key differences of OOP and FP when it comes to the architecture of complex and differentiated frameworks. PyTorch emphasizes the ease of use when modularity is involved, while JAX on the other hand emphasizes explicitness and clarity at the cost of convenience. Equinox demonstrates a middle ground for how object oriented principles can be built on top of one another and pushed onto a functional system to create a cleaner and more maintainable architecture without sacrifice to performance. These patterns show that framework on its own can fully address the architectural challenges of ML systems, shows the need for a hybrid design that leverages the strengths of both OOP and FP.

Limitations

This analysis focuses on PyTorch, JAX, and Equinox, and does not examine other major frameworks such as TensorFlow, Flax, or Haiku, which may apply design patterns differently. The performance comparisons made in this paper are based on simple linear models with synthetic data, so the results may not generalize to large scale frameworks or real world workloads. The evaluation made in this paper is qualitative, meaning that it does not evaluate these patterns through real world testing or large measures. In addition, Equinox is a growing library, and its long term usability and viability support remain unclear. These limitations suggest that while our analysis highlight meaningful architectural differences, further research across more diverse frameworks is still needed.

Conclusion

Summary of Findings

This paper examined how OO and FP programming influence the design of modern machine learning frameworks by comparing PyTorch, JAX, and Equinox. This analysis shows that PyTorch, an OOP based framework provides modularity and ease of use through the use of abstraction layers, but these abstractions can lead to a lack of clarity to the underlying computation. On the other hand, functional approaches like JAX provide greater transparency and performance benefits but at the same time introduce the issue of complexity and scaffolding that can be difficult to scale in large projects. Equinox demonstrates a workable comprise allowing for the strengths from both sides to be brought together. It adds an object like structure while preserving the transformation system JAX depends on.

Implications for ML Framework Design

These findings point to a larger trend. Suggesting that the architectural needs of modern machine learning systems cannot be fully met by either approach alone. As models continue to grow and scale in complexity, frameworks that bring together modularity, clarity and high performance will become more and more important. Hybrid designs like Equinox represent a promising direction for future development, offering a foundation for frameworks that can create a layer of abstraction, capable of supporting advanced machine learning workloads. As the research in this area continues to grow these ideas will be essential for advancing the field towards a more maintainable, transparent, and efficient machine learning systems.

Future Work

Scaling to Larger Architectures

Our analysis reviews multiple directions that could make future studies of design patterns in machine learning frameworks even stronger. One next step that is clear would be to use the ideas mentioned in the paper on larger, more realistic models. Meaning architectures that power models like GPT, BERT, Bit, etc. By using a mechanism that lets a model look at all the different parts of input at the same time and decide what parts are most important. Or even architectures like ResNet, VGG, EfficientNet, etc. These rely on stacks of increasingly complex layers to detect detect patterns in images, edges, shapes, texture and objects. What was done in this paper stayed within the limits of very simple linear models. Exploring how OOP and FP systems act under real world workloads would allow for a much clearer idea of how they handle scalability.

Evaluating Hybrid Frameworks in Production

Another topic that may be worth exploring would be how hybrid frameworks like Equinox perform in production settings. Here we focus on modularity, we fail to discuss how real systems come with challenges like distributed training, synchronization, and long term maintenance. It is still unclear whether or not hybrid designs create technical debt or simplify the issues listed above.

Bridging High Level Abstractions and Low Level Compilers

Lastly, exploring how high level abstractions and low level compilers interact. Right now frameworks force developers to make a choice between object oriented interfaces and transformation tools used by frameworks like JAX. It is possible that in the future a set of tools may dissolve this current boundary by allowing models written in PyTorch style code to be rewritten almost seamlessly, optimized and executed efficiently. Looking at small domain specific languages could assist in making frameworks that are easy to use but also allow for compiler level performance.

References

[0] PyTorch. "Optimizing Model Parameters." *PyTorch Tutorials*. Retrieved from https://docs.pytorch.org/tutorials/beginner/basics/optimization_tutorial.html, accessed 2025.

[1] Inc, R. (2025). AI in Production: Challenges and Strategies for Scaling Machine Learning Models. *Medium*. Retrieved from https://medium.com/@marketing_27918.

[2] Demystifying XLA: Unlocking the power of accelerated linear algebra. — by Muhammed Ashraf — Medium. (n.d.). https://medium.com/@muhammedashraf2661/demystifying-xla-unlocking-the-power-of-accelerated-linear-algebra-9b62f8180dbd

[3] Admin. (2025). The Black Box Problem in Medical AI: Lessons from Watson. *Kent Hospitals*. Retrieved from https://kenthospitals.com/health/black-box-ai-medicine.

[4] Giri, C., & Chen, Y. (2022). Deep Learning for Demand Forecasting in the Fashion and Apparel Retail Industry. *Forecasting*, 4(2), 565–581. https://doi.org/10.3390/forecast4020031.

[5] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., et al. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. *arXiv preprint* arXiv:1912.01703. https://arxiv.org/abs/1912.01703.

[7] Kidger, P., & Garcia, C. (2021). Equinox: Neural Networks in JAX via Callable PyTrees and Filtered Transformations. *arXiv preprint* arXiv:2111.00254. https://arxiv.org/abs/2111.00254.

[8] Python numerical methods. Inheritance, Encapsulation and Polymorphism - Python Numerical Methods. (n.d.). https://pythonnumericalmethods.studentorg.berkeley.edu/notebooks/chapter07.03-Inheritance-Encapsulation-and-Polymorphism.html

[9] JAX documentation. (2025). Pytrees. Retrieved from https://docs.jax.dev/en/latest/pytrees.html.

[10] JAX documentation. (2025). Key concepts. Retrieved from https://docs.jax.dev/en/latest/key-concepts.html

[11] JAX documentation. (2025). Introduction to debugging. Retrieved from https://docs.jax.dev/en/latest/debugging.html

[12] Kidger, P. (n.d.). Debugging tools¤. Debugging tools - Equinox. https://docs.kidger.site/equinox/api/debug/

[13] Karl, T. (2024, February 12). Jax vs pytorch: Comparing Two deep learning frameworks. New Horizons. https://www.newhorizons.com/resources/blog/jax-vs-pytorch-comparing-two-deep-learning-framew