



Leveraging the JAX AI Stack

A High-Performance Journey for PyTorch Developers

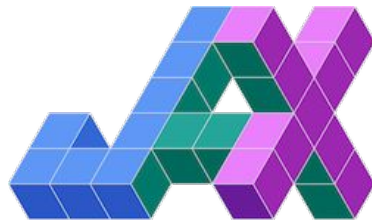


Why Consider JAX? Scalability

JAX Scalability: Scaling to 50,944 TPUs with JAX

- In November 2023, Google used JAX for an extremely large LLM training job
- **50,944 Cloud TPU v5e chips**
- Demonstrated **near-ideal linear scaling**

JAX is the foundation for Google's largest models, including **Gemini, Gemma, Imagen, and Veo.**



What is the JAX AI Stack?

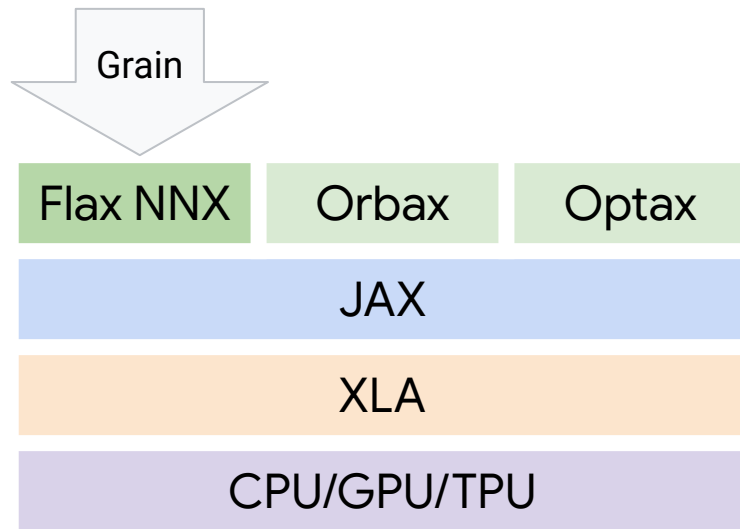
A curated set of interoperable libraries for high-performance ML research and development.

- **Core Philosophy:** Achieve Performance, Flexibility, and Scalability using **function transformations**.
- **Engine:** Uses the **XLA (Accelerated Linear Algebra)** compiler to generate highly optimized code for the target hardware.
- **Portability:** Enables running the same code, often without modification, across **CPUs, GPUs, and TPUs**.



The Full Stack: A Modular, Layered Design

- **Grain:** Data Loading (optional)
- **Flax NNX:** Neural Networks
- **Optax:** Optimizers
- **Orbax:** Checkpointing
- **JAX:** Function Transformations & NumPy API
- **XLA:** Compiler



Roles of the Core Libraries

- **JAX:** The foundation. Provides a NumPy-like API and the core function transformations (`jit`, `grad`, `vmap`).
- **Flax NNX:** The model builder. A Pythonic neural network library, analogous to `torch.nn.Module`, for defining models.
- **Optax:** The optimizer. A library for gradient processing and optimization, analogous to `torch.optim`.
- **Orbax:** The checkpointer. A library for saving and restoring training state (model params, optimizer state), analogous to `torch.save` and `torch.load` but built for distributed systems.
- **Grain:** The data loader. A high-performance, deterministic data loading library, analogous to `torch.utils.data.DataLoader`.



The JAX Engine: Composable Function Transformations

JAX's power comes from wrapping Python functions in transformations that change *how* they execute.

- `jax.jit()` -> **Compiles** the function with XLA for high speed.
- `jax.grad()` -> Creates a new function that computes **gradients**.
- `jax.vmap()` -> **Vectorizes** or "auto-batches" the function.



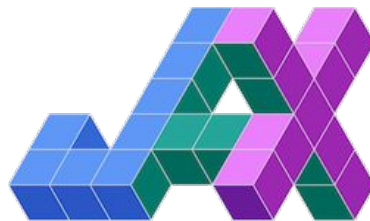
These transformations can be arbitrarily composed, e.g.,

```
jax.jit(jax.grad(loss_fn)).
```

Key Transform: `jax.jit` (Compilation)

The `jit` (Just-In-Time) transform uses XLA to compile a JAX-compatible Python function into highly optimized machine code for the target accelerator.

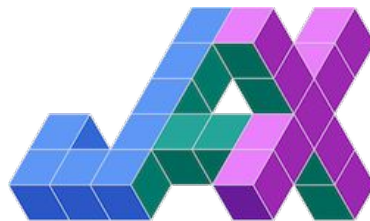
- Dramatically speeds up execution compared to standard Python by fusing operations, optimizing memory use, and avoiding interpreter overhead.
- Conceptually similar to `torch.compile()`, but `jit` is the fundamental and standard way to get performance in JAX.



Key Transform: `jax.grad` (Differentiation)

`grad` provides powerful and flexible automatic differentiation.

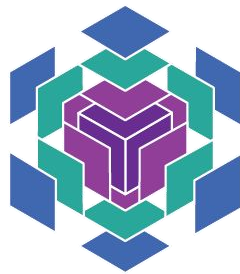
- It is a **transformation** that takes a numerical function (e.g., a loss function) and returns a *new function* that computes its gradient.
- This is different from PyTorch's `loss.backward()`, which computes gradients and stores them as side-effects on `.grad` attributes.
- It composes seamlessly with `jit` and other JAX transforms.



Flax NNX: Pythonic Models

Flax NNX provides structure for defining Neural Networks, giving you a familiar object-oriented experience on top of JAX's functional core.

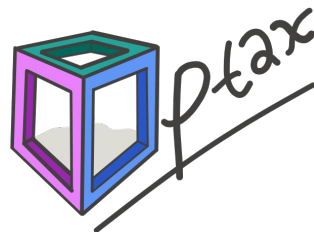
- **Pythonic Feel:** Define models by subclassing `nnx.Module`, making them feel like regular Python objects. Analogous to `torch.nn.Module`.
- **Manages State:** While it provides this convenient object-style, it is designed to work correctly with JAX's functional nature, managing the model's state (parameters, batch stats) in a way that is compatible with `jit` and `grad`.



Optax: Composable Optimizers

Optax is the gradient processing and optimization library, analogous to `torch.optim`.

- **Composable:** Build optimizers by chaining smaller, independent transformation blocks (e.g., `add_momentum`, `scale_by_adam`).
- **Stateful but Functional:** An optimizer's state is handled explicitly.
 1. `init()` function creates the optimizer's state.
 2. `update()` function takes gradients and the current state, and returns the parameter updates and the *new* optimizer state.



Orbax: Robust Checkpointing

Orbax handles saving and loading of training state (model params, optimizer state, etc.), designed specifically for the JAX ecosystem.

- **Distributed-Aware:** Manages saving/restoring JAX PyTrees, even when state is sharded across many accelerators and hosts.
- **Fault-Tolerance:** Critical for long-running jobs on preemptible infrastructure.
- **Asynchronous:** Can save checkpoints in the background to minimize impact on training throughput.

Grain: High-Performance Data Loading

Grain is Google's library for efficient data reading and preprocessing, designed for JAX.

- **Purpose-Built for JAX:** Solves data bottlenecks to keep fast accelerators fed with data.
- **Parallel Processing:** Uses multiprocessing to prepare batches in parallel, bypassing Python's GIL.
- **Distributed Sharding:** Integrates with JAX's distributed environment to automatically provide each process with a unique slice of the data.

Comparison: Training Loop & Backpropagation

```
# PyTorch
import torch
import torch.nn as nn
import torch.optim as optim
```

```
# Define a simple model
class SimpleModel_Torch(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        return self.linear(x)
```

```
# Flax NNX
from flax import nnx
import jax
import jax.numpy as jnp
from optax import sgd
from typing import Any
```

```
# Define a simple model
class SimpleModel_NNX(nnx.Module):
    def __init__(self, *, rngs: nnx.Rngs):
        self.linear = nnx.Linear(1, 1, rngs=rngs)

    def __call__(self, x: jnp.ndarray):
        return self.linear(x)
```

Comparison: Training Loop & Backpropagation

PyTorch

```
model_torch = SimpleModel_Torch()
```

```
optimizer_torch = \  
    optim.SGD(model_torch.parameters(), lr=0.01)  
loss_fn = nn.MSELoss()
```

Dummy data

```
x_torch = torch.tensor(  
    [[2.0]], requires_grad=True)  
y_torch = torch.tensor([[4.0]])
```

Flax NNX

```
model_nnx = SimpleModel_NNX(rngs=nnx.Rngs(0))
```

Optimizer

```
optimizer = nnx.Optimizer(  
    model_nnx,  
    tx=sgd(learning_rate=0.01),  
    wrt=nnx.Param)
```

Dummy data

```
x_nnx = jnp.array([[2.0]])  
y_nnx = jnp.array([[4.0]])
```

Comparison: Training Loop & Backpropagation

```
# PyTorch Training step

# Zero the gradients
optimizer_torch.zero_grad()
output_torch = model_torch(x_torch)
loss_torch = loss_fn(
    output_torch, y_torch)
loss_torch.backward() # Compute gradients

# Update parameters
optimizer_torch.step()

print("PyTorch Loss:", loss_torch.item())
```

```
# Flax NNX Training step
@nnx.jit
def train_step(model, optimizer, x, y):
    def loss_fn(model):
        return jnp.mean((model(x) - y) ** 2)

    loss, grads = \
        nnx.value_and_grad(loss_fn)(model)
    # in-place updates
    optimizer.update(model, grads)

    return loss

# Pass the optimizer
loss_nnx = train_step(model_nnx,
                       optimizer, x_nnx, y_nnx)

print("Flax NNX Loss:", loss_nnx)
```

JAX Superpower: Flexible Parallelism

JAX offers powerful, flexible ways to scale across multiple accelerators, driven by the compiler.

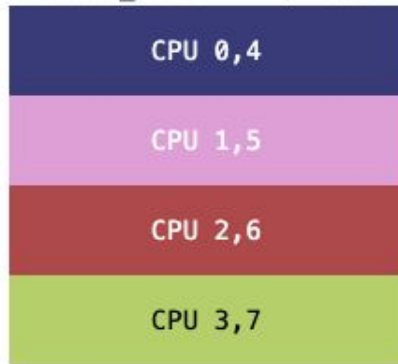
PyTorch: Library-Based

- You wrap your single-device model in a library object like `DDP` or `FSDP`.
- The library manages communication behind the scenes.
- `model = DDP(model)`

JAX: Compiler-Driven (SPMD)

- You describe the desired parallel *layout* of your data and parameters using sharding annotations.
- `jax.jit` compiles a new, optimized parallel program from scratch based on these annotations.
- This provides a more unified and flexible approach to different parallelism strategies (Data, Model, etc.).

`sharded_model.w2 ('model', None)`



Summary & The Full Workflow

- The JAX AI Stack provides a complete, high-performance workflow:
Grain (data) -> Flax (model) -> Optax (optimization) -> Orbx (checkpointing).
- JAX enables high-performance via composable **function transforms** (jit, grad, vmap).
- **Flax NNX** provides a familiar, Pythonic, object-oriented way to define models (`torch.nn.Module-like`).
- The functional paradigm makes state explicit — passing models and state into functions — which is key to performance, reproducibility, and scalability.
- JAX supports compiler-driven **parallelism** for unparalleled scaling.



Learning Resources

Code Exercises, Quick References, and Slides

- <https://goo.gle/learning-jax>



Community and Docs

Community:

- <https://goo.gle/jax-community>

Docs

- JAX AI Stack: <https://jaxstack.ai>
- JAX: <https://jax.dev>
- Flax NNX: <https://flax.readthedocs.io>