JAX AI Stack Quick Reference

Core Philosophy:
Achieve Performance, Flexibility, and Scalability using **function transformations** on Python & NumPy-like code. Leverages **XLA (Accelerated Linear Algebra)** compiler for optimized, portable code across CPUs, GPUs, and TPUs.

JAX Core: Functional Engine & Transformations

- **NumPy-like API:** `jax.numpy` (often imported as `jnp`) provides a familiar interface.
- **Functional Programming:** Functions ideally have no side-effects. Model parameters & optimizer states are passed as arguments and returned as results.
- **PyTrees:** Nested Python dict/list/tuple structures used to manage collections of parameters, gradients, and states. JAX transformations work seamlessly with PyTrees.

Key Function Transformations:

1. **`jax.jit` (Just-In-Time Compilation)**
    - **Purpose:** Compiles a JAX-compatible Python function into highly optimized machine code using XLA.
    - **Usage:** Typically used as a decorator: `@jax.jit`
    - **Benefits:** Dramatically speeds up execution by fusing operations, optimizing memory, and avoiding Python interpreter overhead.
    - **GSPMD (Automatic Parallelism):** `jit` can also orchestrate automatic parallelism.
        - Decorate function with `jit`.
        - Provide **sharding annotations** for inputs/outputs (how data is distributed).
        - XLA compiler partitions computation and inserts cross-device communication.
2. **`jax.grad` (Automatic Differentiation)**
    - **Purpose:** Transforms a numerical function (e.g., loss function) into a new function that computes its gradient.

    - **Usage:** `grad_fn = jax.grad(loss_fn)`
      `gradients = grad_fn(params, *args)`
    - **Note:** Composes with `jit` and other transforms. `jax.value_and_grad(fn)` is common to get both function's output (e.g., loss) and gradients.
3. **`jax.vmap` (Vectorization / Auto-Batching)**
    - **Purpose:** Transforms a function written for a single data point into one that efficiently maps across a batch dimension.
    - **Usage:** `batched_fn = jax.vmap(fn_for_single_example)`
    - **Benefits:** Simplifies code – write logic for one example, `vmap` handles batching.

Explicit Parallelism:

- **`shard_map` (Manual SPMD Parallelism)**
    - **Purpose:** Provides explicit, manual control over Single-Program Multiple-Data parallelism.
    - **Key Components:**
        - `jax.sharding.Mesh(device_array, axis_names)`: Defines a logical grid of devices with named axes (e.g., 'data', 'model').
        - `jax.sharding.PartitionSpec(...)`: Describes how tensor dimensions are sharded (split, e.g., `P('data')`) or replicated (e.g., `P(None)`) across `Mesh` axes.
    - **Usage:** Apply `shard_map` to a function, specifying input/output sharding with `PartitionSpecs` over a `Mesh`.
    - **Responsibility:** User explicitly defines cross-device communication (e.g., `jax.lax.psum` for sum-reduction).

Flax NNX: Pythonic Neural Network Models

- **Purpose:** Provides structure for defining and managing Neural Networks (like `torch.nn.Module`). NNX API focuses on an intuitive, Pythonic experience.
- **Model Definition**: Models are regular Python objects using standard object model and reference semantics.

```python
from flax import nnx
class MyModel(nnx.Module):
    def __init__(self, din, dout, *, rngs: nnx.Rngs):
        self.linear = nnx.Linear(din, dout, rngs=rngs)
    def __call__(self, x):
        return self.linear(x)
# Instantiate: model = MyModel(10, 2,
rngs=nnx.Rngs(params=jax.random.PRNGKey(0)))
```

- **State Management:** Integrates with JAX functional API (`jit, grad`) to manage state (e.g., 'params', 'batch_stats') explicitly but conveniently.
- **`nnx.Rngs`: Manages PRNG keys for initializations.**
- **`@nnx.jit`**: Decorator for JIT-compiling NNX model methods or functions operating on NNX models.
- **`loss, grads = nnx.value_and_grad(loss_fn)(model)`**: Computes loss and gradients for an NNX model. `loss_fn` should take the model as its first argument.
- **`nnx.Optimizer(model, tx=optax_optimizer)`**: Wraps an NNX model and an Optax optimizer.
    - `optimizer.update(grads)`: Applies gradient updates in-place to the model wrapped by the optimizer.

Optax: Composable Optimizers

- **Purpose:** Gradient processing and optimization library (like `torch.optim`).
- **Philosophy:** Optimizers are chains of composable gradient transformation building blocks (e.g., `optax.add_decayed_weights()`, `optax.scale_by_adam()`).
- **Usage (Stateful):**
    1. **Define:** `optimizer_tx = optax.adam(learning_rate=0.01)` (or `optax.sgd(...)` etc.)
    2. **Initialize (if not using `nnx.Optimizer`):** `opt_state = optimizer_tx.init(params)`

3. **Update (if not using `nnx.Optimizer`):**

   ```
   updates, opt_state = optimizer_tx.update(grads, opt_state, params)
   new_params = optax.apply_updates(params, updates)
   ```

- **With Flax NNX:** `nnx.Optimizer(model, tx=optax_tx)` abstracts state initialization and update application.

Orbax: Robust Checkpointing

- **Purpose:** Saving and loading training state (model params, optimizer state, PyTrees) (like `torch.save/load`).
- **Key Features:**
  - Designed for distributed (multi-host, multi-device) settings.
  - Manages saving/restoring JAX PyTrees.
  - Critical for fault-tolerance in long-running jobs.
  - Supports asynchronous checkpointing to minimize impact on training.

More Information

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