Flax NNX & JAX Quick Reference

Core Concepts & Philosophy

- **Flax NNX:** Modern, Pythonic neural network library for JAX. Recommended for new users. Built on JAX, designed for flexibility and performance.
- **Pythonic Interface:** Uses regular Python object semantics (classes, attributes, methods).
- **Python Graphs:** Internal representation for NNX objects. Enables standard Python reference sharing and *mutability* (in-place state changes).
- **Explicit State Management:** Clear separation between static config and dynamic state. Parameters initialized eagerly.
- **Explicit RNG:** Random Number Generation requires explicit handling via `nnx.Rngs`.

Key NNX Classes & Objects

- **`nnx.Module`**: Base class for all layers and models.
  - Define submodules/parameters in `__init__`.
  - Define forward pass logic in `__call__`.
  - Modules directly hold their own state (parameters, etc.).
- **`nnx.Variable`**: General container for dynamic state (can be mutated). Access underlying JAX array with `.value`.
- **`nnx.Param`**: Subclass of `nnx.Variable`. Used specifically for *trainable* model parameters (weights, biases).
- **`nnx.State`**: Container for other *mutable* state variables (e.g., BatchNorm statistics, counters). Often used as a dictionary-like object.
- **`nnx.Rngs`**: Object to manage JAX PRNG keys for reproducible randomness. Passed during module initialization. Layers that use RNGs (like Dropout) hold a forked copy of the Rngs object, not a shared reference.

State Management in

- **Static Configuration:** Regular Python attributes (e.g., `self.dropout_rate = 0.5`).
- **Dynamic State:** Stored in `nnx.Variable`, `nnx.Param`, or `nnx.State` objects (e.g., `self.weight = nnx.Param(...)`).

- **Accessing Values:** Use `.value` on `Variable`/`Param` instances (e.g., `self.weight.value`).
- **Updating State:** Direct assignment to `.value` or modifying `nnx.State` contents is possible within methods (handled by NNX mechanisms).

Functional API (Interfacing with JAX)

- **`nnx.split(module)`**: Decomposes an `nnx.Module` instance into static structure (`GraphDef`) and dynamic state (`State` PyTree). Necessary for using stateful objects with pure JAX functions.
- **`nnx.merge(graphdef, state)`**: Reconstructs an `nnx.Module` instance from its `GraphDef` and `State`.
- **`nnx.update(module, state)`**: Updates an existing `nnx.Module` object in-place with the content of a `State` PyTree.

Transformations: JAX vs. NNX		
Feature	**JAX (`jax.jit`, `jax.grad`, `jax.vmap`)**	**NNX (`nnx.jit`, `nnx.grad`, `nnx.vmap`)**
Works On	**Pure Functions**, PyTrees (lists, dicts, tuples)	**NNX Objects** (`nnx.Module`, `nnx.Optimizer`, etc.)
State Handling	**Manual:** Pass state in, get updated state back	**Automatic:** Handles state lifting & merging implicitly
Style	Functional	More Object-Oriented (can apply to methods)
Purity Req.	**Strict:** Functions must be pure (no side effects)	Manages state mutation within NNX objects correctly
Speed	Generally **Faster** (less overhead)	Slightly Slower (due to state management)
When to Use	Pure computations, data processing, low-level control	Working with `nnx.Modules`, models, optimizers; convenience

- **Pure Function:** Returns same output for same input, no side effects (e.g., no modifying globals, I/O). Required for `jax.jit`.
- **Caution:** Avoid JITting functions where standard Python control flow (`if`/`while`) depends directly on *values* of JAX array inputs. Can lead to re-compilation.

Common NNX Layers

- `nnx.Linear`
- `nnx.Conv`
- `nnx.BatchNorm`
- `nnx.LayerNorm`
- `nnx.MultiHeadAttention`
- `nnx.Dropout`
- `nnx.LSTMCell`, `nnx.GRUCell`

Optimization (with Optax)

- **Optax:** Preferred JAX library for optimizers (SGD, Adam, etc.).
- **`nnx.Optimizer(model, tx, wrt=nnx.Param)`**: Wrapper around an Optax optimizer (`tx`). Links the optimizer state to a specific model instance or type.
- **`optimizer.update(model, grads)`**: Applies gradient updates. Updates the model's parameters (specified by `wrt`) in-place.

Training Loop Key Functions

- **`loss, grads = nnx.value_and_grad(loss_fn)(model, ...)`**: Computes the loss value *and* the gradients of the loss with respect to the model's (`nnx.Module`) parameters/state. `loss_fn` typically takes the model as its first argument.
- **`optimizer.update(model, grads)`**: Applies the computed gradients (see above).

Utilities

- **`nnx.display(object)`**: Utility to visualize the structure (Python Graph) of an NNX object (like `nnx.Module`) in notebooks. Useful for debugging.

More Information

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