JAX & Flax NNX Debugging Quick Reference

Core Challenge: The JIT Impact

- **JAX relies on** Compiles Python functions (tracing phase) into optimized code (execution phase).
- **Problem:** Standard Python `print()`, `pdb`, `breakpoint()` only see abstract *tracers* during tracing, not runtime *values* inside JIT-compiled code.
- **Goal:** Use JAX-specific tools or temporarily disable JIT to inspect runtime behavior.

Runtime Inspection (Inside JIT / Transformed Functions)

`jax.debug.print("msg: {var}", var=value, ordered=True)`

- **Purpose:** The JAX equivalent of `print()` for runtime values.
- **How:** Embeds print into the compiled graph.
- **Notes:** Use `ordered=True` for sequential output. Shows concrete values during execution.

`jax.debug.breakpoint()`

- **Purpose:** The JAX equivalent of `pdb.set_trace()` / `breakpoint()` for interactive debugging *inside* JIT.
- **How:** Pauses runtime execution, provides a `(jaxdb)` prompt.
- **Commands:** `p <var>` (print variable), `c` (continue), `q` (quit). Limited command set vs `pdb`.
- **Notes:** Can be made conditional using `jax.lax.cond()`. Standard Python `breakpoint()` can inspect *tracers* during tracing.

Enabling Standard Python Tools (Eager Execution)

`jax.disable_jit()`

- **Purpose:** Temporarily forces eager execution (like PyTorch/NumPy), making standard Python tools work.
- **How:**
    - Context Manager: `with jax.disable_jit(): ...` (Recommended)

    - Global: `jax.config.update("jax_disable_jit", True)`
    - Env Var: `JAX_DISABLE_JIT=1`
- **Effect:** Standard `print()`, `pdb.set_trace()`, `breakpoint()`, IDE debuggers work directly on runtime values.
- **Drawback: Significant performance loss.** Use temporarily for debugging.
- **Limits:** May not fully step into `jax.vmap` / `jax.scan` internals.

Automatic Error Detection

`jax_debug_nans` Flag

- **Purpose:** Automatically find the source operation causing NaNs inside JIT.
- **How:** `jax.config.update("jax_debug_nans", True)` or `JAX_DEBUG_NANS=1`.
- **Effect:** If NaN detected, JAX re-runs eagerly to pinpoint the error source.
- **Limitations:** Slows execution; doesn't work with `jax.shard_map`; may flag intentional NaNs.

Flax NNX Specific Tools

`nnx.display(object)`

- **Purpose:** Visualize the structure and state of NNX objects (Modules, Optimizers, State).
- **How:** Provides a hierarchical view, uses `treescope` for interactive display in notebooks/Colab if available.
- **Use:** Verify model architecture, check parameters/state initialization, inspect optimizer state. Analogous to `print(pytorch_model)`.

`Module.sow(variable_type, name, value, ...)`

- **Purpose:** Capture intermediate values (e.g., activations) during `nnx.Module.__call__` without altering function signatures.
- **How:** Stores `value` as an attribute (`name`) on the module instance. Access via `module.<name>.value`.
- **Notes:** Default appends to a tuple if called multiple times with the same name. Use `variable_type` (e.g., `nnx.Intermediate`) for organization.

Chex for Robustness & Debugging

Purpose: Library for reliable JAX code (assertions, testing).

Static Assertions: Check shapes, dtypes, ranks, structure (properties known at trace time).

- Examples: `chex.assert_shape`, `chex.assert_type`, `chex.assert_rank`, `chex.assert_trees_all_equal_shapes`.
- **Usage:** Place directly inside JIT-compiled functions.

Value Assertions: Check properties based on runtime values (NaN/Inf, closeness).

- Examples: `chex.assert_tree_all_finite`, `chex.assert_trees_all_close`.
- **Usage (in JIT):** Requires `@chex.chexify()` decorator.

`@chex.chexify`

- **Purpose:** Decorator to enable Chex *value* assertions inside JIT.
- **How:** Apply *outside* `@jax.jit`. Function returns `(error, result)`. Check `error.throw()` or `error.get()`.
- **Notes:** Performance overhead; debugging tool; **Doesn't work in Colab currently**. Usually need `chex.block_until_chexify_assertions_complete()` after call.

`@chex.assert_max_traces(n=...)`

- **Purpose:** Debug performance by detecting unexpected JIT re-compilations.
- **How:** Apply *inside* `@jax.jit`. Raises `AssertionError` if function is traced > `n` times.
- **Use:** Pinpoint causes of slow-downs due to unstable input shapes/types. Use `chex.clear_trace_counter()` if needed.

Monitoring & Visualization

TensorBoard

- **Purpose:** Visualize training metrics, images, profiling data.
- **Setup:** Create a Summary Writer (e.g., via `tensorflow`, `torch.utils.tensorboard`, or `tensorboardX`). Launch `tensorboard --logdir <log_dir>`.

- **Logging:** Use `writer.add_scalar`, `writer.add_image`, etc. **Remember `.item()` to convert JAX scalar arrays to Python scalars before logging.**
- **Profiling:** JAX profiler (`jax.profiler`) can output data viewable in TensorBoard.

Recommended Workflow Summary

1. **Static Checks:** `nnx.display`, Chex static assertions (`assert_shape`, etc.).
2. **Runtime JIT Issues:** Chex value assertions + `@chex.chexify` (for NaNs, etc.), `jax_debug_nans`, `jax.debug.print`, `jax.debug.breakpoint`.
3. **Complex Issues:** Temporarily use `with jax.disable_jit():` to enable standard `pdb`/`print`.
4. **Performance:** `chex.assert_max_traces`, `jax.profiler` + TensorBoard.
5. **Monitor:** Use TensorBoard throughout.

Profiling

Profiling is also essential for understanding and improving your code. Xprof is a great tool for profiling JAX and Flax NNX, and is compatible with TensorBoard. An excellent tutorial is available which includes profiling, so developers are encouraged to review that tutorial.

More Information

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