Debugging JAX & Flax NNX:

A Guide for PyTorch Users

Leveraging Familiar Concepts and Mastering New Tools

Why is JAX Debugging Different? The JIT Impact

- **PyTorch**: Mostly eager execution. `print()`, `pdb` work directly on runtime values.
- **JAX**: Relies heavily on Just-In-Time (JIT) compilation (`@jax.jit`) for performance.
  - **Tracing Phase**: Python code runs once with abstract tracers (shapes/types) to build a computation graph. Standard `print()` or `pdb` only see these tracers. However tracers can be useful, since they carry shapes and `dtypes`.
  - **Execution Phase**: Compiled graph (e.g., XLA) runs later on CPU/GPU/TPU with concrete values. Standard tools can't inspect this directly.
- **Challenge**: How to inspect *runtime* values inside JIT-compiled code?

"`printf` Debugging" in JAX: `jax.debug.print()`

- The JAX equivalent of `print()` for use inside transformed functions (`jit, vmap, grad`).
- Embeds the print operation into the compiled computation graph.
- Outputs concrete runtime values during execution.
- Use `ordered=True` to ensure prints appear sequentially as written.

"`printf` Debugging" in JAX: `jax.debug.print()`

```
@jax.jit
def compute_intermediate(x):
  y = x * 2
  print("Standard print (tracer):", y)   # Sees tracers during compilation
  # Sees runtime values
  jax.debug.print("jax.debug.print (runtime value): {y}", y=y, ordered=True)
  z = y + 1
  return z

compute_intermediate(jnp.array(5.0))
# Output:
# Standard print (tracer): Traced<ShapedArray(float32)...>
# jax.debug.print (runtime value): 10.0
```

Interactive Debugging in JIT: `jax.debug.breakpoint()`

- The JAX equivalent of **`pdb.set_trace()`** or **`breakpoint()`** for use inside transformed functions.
- Pauses execution at runtime within the compiled code.
- Provides a (**`jaxdb`**) prompt similar to **`pdb`**.
- Allows inspecting runtime values of variables in the JAX context (**`p variable_name`**).
- Use **`c`** to continue, **`q`** to quit.
- Can be made conditional using **`jax.lax.cond()`**.

Interactive Debugging in JIT: `jax.debug.breakpoint()`

```
@jax.jit
def check_value(x):
  y = jnp.sin(x)
  jax.debug.print("Value before breakpoint: {y}", y=y)
  jax.debug.breakpoint() # Execution pauses here
  z = jnp.cos(y)
  return z

check_value(jnp.array(0.5))
# Output: Value before breakpoint: 0.47942555
# (jaxdb) p y
# Array(0.47942555, dtype=float32)
# (jaxdb) c
```

Visualizing Data Layout: `jax.debug.visualize_array_sharding`

- **Problem**: Need to verify data layout across devices in distributed settings with sharding.
- **Tool**: `jax.debug.visualize_array_sharding(array)` prints text diagram of layout.
- **How it Works**: Shows which data slice is on which device ID at runtime.
- **Usage**: Call inside JIT/distributed functions to check sharding from `Mesh/PartitionSpec`.

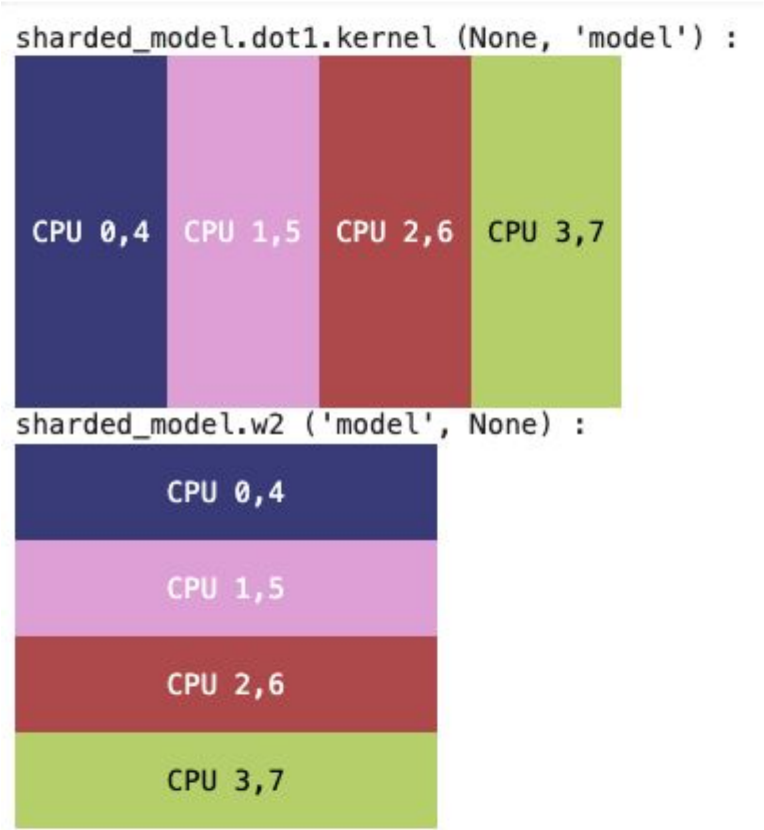
visualize_array_sharding Code Example

```python
# --- Assume mesh setup (e.g., 2x2) ---
# mesh = Mesh(devices, ('data', 'model'))
# P = PartitionSpec

@jax.jit
def my_distributed_func(x_sharded):
  # Visualize input array layout
  print("--- Input Sharding ---")
  jax.debug.visualize_array_sharding(x_sharded) # <<< Visualize

  y = x_sharded * 2 # Computation
  print("--- Output Sharding ---")
  jax.debug.visualize_array_sharding(y) # <<< Visualize
  return y
```

Visualizing Data Layout: `jax.debug.visualize_array_sharding`



Back to Basics: Temporarily Disabling JIT

Sometimes you just want standard Python debugging.

- `jax.disable_jit()` forces JAX functions to execute eagerly (like PyTorch/NumPy).
- Allows standard `print()` and `pdb/breakpoint()` to work as expected, inspecting runtime values directly.
- Also enables IDE debuggers like VS Code

Back to Basics: Temporarily Disabling JIT

- How to use:
  - **Context Manager**: `with jax.disable_jit()`: ... (Recommended for locality)
  - **Globally**: `jax.config.update("jax_disable_jit", True)`
  - **Environment Var**: `JAX_DISABLE_JIT=1`
- **Major Drawback**: Disables JIT optimizations -> significantly slower execution. Use temporarily for debugging!

Back to Basics: Temporarily Disabling JIT

```python
@jax.jit
def problematic_function(x):
  y = jnp.log(x)
  # pdb works here ONLY if JIT is disabled
  pdb.set_trace()
  return y * 2

# Execute with JIT disabled for this block
with jax.disable_jit():
  print("Running with JIT disabled...")
  result_no_jit = problematic_function(jnp.array(5.0)) # pdb triggers
```

Back to Basics: Temporarily Disabling JIT

Limits on `jax.disable_jit()`:

- If you’re using functional transforms, like `jax.vmap` and `jax.scan`, you won't be able to break into the function to inspect values
- But tools like `sow` from NNX are designed to be compatible with these transforms

Automatic NaN Hunting: The `jax_debug_nans` Flag

Problem: Numerical instability leading to **NaNs** inside JITted code can be hard to trace.

Solution: Enable the `jax_debug_nans` configuration flag.

- `jax.config.update("jax_debug_nans", True)` (in Python)
- Environment Variable: `JAX_DEBUG_NANS=1`

Automatic NaN Hunting: The `jax_debug_nans` Flag

How it Works:

- Monitors JIT computations for **`NaN`** outputs.
- If a **`NaN`** is detected, JAX automatically re-runs the function in eager mode (like **`disable_jit`**) to pinpoint the exact operation causing the **`NaN`** and raise an error there.

Automatic NaN Hunting: The `jax_debug_nans` Flag

Limitations:

- Can significantly slow down execution due to checks and potential eager re-runs.
- Might raise errors on intentionally created **`NaNs`**.
- Best used during debugging, disable for production/performance runs.

Inspecting Flax NNX Models: `nnx.display()`

Google

Understanding Your NNX Model: `nnx.display()`

Flax NNX is designed with inspectability in mind (uses standard Python objects)

- `nnx.display()` provides a clear view of NNX objects:
  - `nnx.Module` (models, layers)
  - `nnx.Optimizer`
  - `nnx.State`
  - Other contained JAX arrays/objects

Understanding Your NNX Model: `nnx.display()`

- Shows structure, parameters, state variables, shapes, types, and values.
- Creates a rich, interactive tree view in notebooks/Colab. Falls back to standard print otherwise.
- Analogous to `print(pytorch_model)` but often more detailed and interactive.

```
Linear( # Param: 15 (60 B)
  w=Param(value=<jax.Array float32(2, 5) ≈0.42 ±0.34 [≥0.029, ≤0.95] nonzero:10>),
  b=Param(value=<jax.Array float32(5,) ≈0.0 ±0.0 [≥0.0, ≤0.0] zero:5>),
  din=2,
  dout=5,
)
```

Google

Capturing Intermediate Values: `nnx.sow()`

Flax NNX `Module.sow()` - Capturing Intermediate Values

What is `Module.sow()`?

- A method within **`flax.nnx.Module`** designed to capture and store intermediate values computed during a module's forward pass.
- **Purpose**: Simplifies tracking internal computations (e.g., activations, gradients) without manually passing data structures through the call stack.
- Useful for:
  - Debugging model behavior.
  - Visualizing intermediate layer outputs.
  - Implementing custom loss functions based on internal states.

Flax NNX `Module.sow()` - Capturing Intermediate Values

How it Works: Core Arguments

```
self.sow(variable_type, name, value, *, reduce_fn=None, init_fn=None)
```

- **`variable_type`:** Specifies the variable wrapper (e.g., **`nnx.Intermediate`**). Used for later filtering/retrieval.
- **`name (str)`:** The attribute name under which the value will be stored on the module instance.
- **`value`:** The actual JAX array or Python object to be stored.
- **`(Optional) reduce_fn, init_fn`:** Customize how values are stored/aggregated if **`sow`** is called multiple times with the same name

Using `Module.sow()` - Storage, Retrieval & Customization

```python
class SimpleModel(nnx.Module):
  def __init__(self, rngs):
    self.dense = nnx.Linear(2, 3, rngs=rngs)
  def __call__(self, x):
    x = self.dense(x)
    # Sow the output of the dense layer
    self.sow(nnx.Intermediate, 'dense_output', x)
    return x * 2

model = SimpleModel(rngs=nnx.Rngs(0))
y = model(jnp.ones((1, 2)))

# Retrieve the sown value
intermediate_val = model.dense_output.value
# Output: (array([[...]]),) - A tuple containing the value
print(intermediate_val)
```

Flax NNX `Module.sow()` - Capturing Intermediate Values

Storage and Retrieval

- Sown values are stored as attributes on the module instance, named according to the name argument.
- The actual data is accessed via the `.value` property of this attribute (e.g., `model.dense_output.value`).
- **Default Behavior:** Calling `sow` multiple times with the same name appends the new value to a tuple stored in `.value`. Be mindful of potential memory growth with frequent calls.

Flax NNX `Module.sow()` - Capturing Intermediate Values

Advanced Features

- **Custom Aggregation:** Use `init_fn` and `reduce_fn` to define custom logic for combining multiple sown values (e.g., sum, product, average) instead of appending to a tuple.
- **State Management:** Sown variables (typed by `variable_type`) integrate with NNX graph utilities like `nnx.split(), nnx.state(), and nnx.pop()` for extracting or managing specific types of intermediate values from the module state.

Robustness with Chex Assertions

Note:
This section is optional if you have already gone through the Chex module.

Catching Errors Early: Chex Assertions

Chex: A library from DeepMind for reliable JAX code (testing, debugging)

- Provides powerful assertion functions.
- Static Assertions: Check properties independent of runtime values (shapes, dtypes, rank, structure).
- Work seamlessly inside `@jax.jit()` because they operate on traced information.

Catching Errors Early: Chex Assertions

Examples (static assertions):

- `chex.assert_shape(array, (batch, features))`
- `chex.assert_rank(array, 2)`
- `chex.assert_type(array, jnp.float32)`
- `chex.assert_trees_all_equal_shapes(tree1, tree2)`

Monitoring with TensorBoard

Google

Visualizing Training: TensorBoard Integration (Setup)

TensorBoard works well with JAX/Flax for monitoring experiments.

Setup:

1. **Install**: `pip install tensorboard` (You might need tensorflow or torch[tensorboard] for the writer).
2. **Create Summary Writer**: Points to a log directory. Can use TensorBoardX, or TensorFlow, or PyTorch utilities.
3. **Launch TensorBoard**: From terminal: `tensorboard --logdir logs`
4. **Access in browser** (usually `http://localhost:6006`).

TensorBoard: Create Summary Writer

```python
# Option 1: TensorBoardX
from tensorboardX import SummaryWriter
writer = SummaryWriter("logs/my_run_1")

# Option 2: TensorFlow writer
import tensorflow as tf
writer = tf.summary.create_file_writer("logs/my_run_1")

# Option 3: PyTorch writer (if torch installed)
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter("logs/my_run_1")
```

Google

Visualizing Training: TensorBoard Integration (Logging)

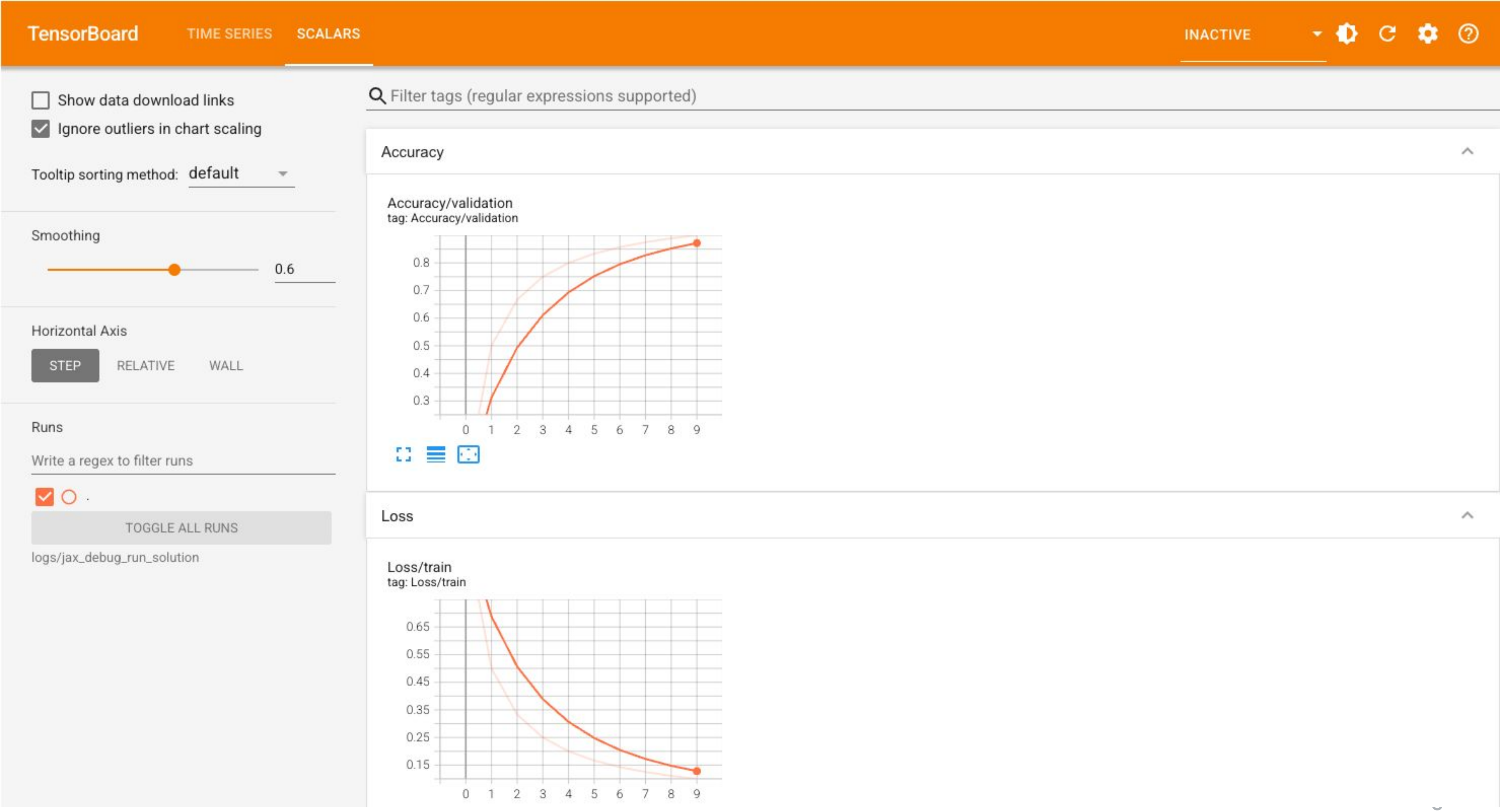
- Log metrics, images, etc., from your training loop.
- **Scalars** (Loss, Accuracy):
  - Use `tf.summary.scalar(...)` (TF writer, or PyTorch writer)
  - **Crucial**: Convert JAX arrays to Python scalars using `.item()` before logging.
- **Images**: Visualize inputs, predictions, attention maps (`writer.add_image(...)`).
- **Profiling**: JAX profiling (`jax.profiler`) can also output data viewable in TensorBoard for performance analysis.

Code Example (Logging Scalars with TensorBoardX Writer)

```
# Inside training loop (epoch = current step)
# Assume train_loss, val_accuracy are JAX arrays

writer.add_scalar('Loss/train', train_loss.item(), global_step=epoch)
writer.add_scalar('Accuracy/validation', val_accuracy.item(), global_step=epoch)

# Remember writer.close() when done
```


TensorBoard
TIME SERIES
SCALARS
INACTIVE
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing
0.6
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
TOGGLE ALL RUNS
logs/jax_debug_run_solution
Filter tags (regular expressions supported)
Accuracy
Accuracy/validation
tag: Accuracy/validation
0.8
0.7
0.6
0.5
0.4
0.3
0 1 2 3 4 5 6 7 8 9
Loss
Loss/train
tag: Loss/train
0.65
0.55
0.45
0.35
0.25
0.15
0 1 2 3 4 5 6 7 8 9

Profiling with XProf, aka JAX Profiler

Installing and Enabling XProf

Installing and enabling XProf in TensorBoard is easy!

XProf includes:

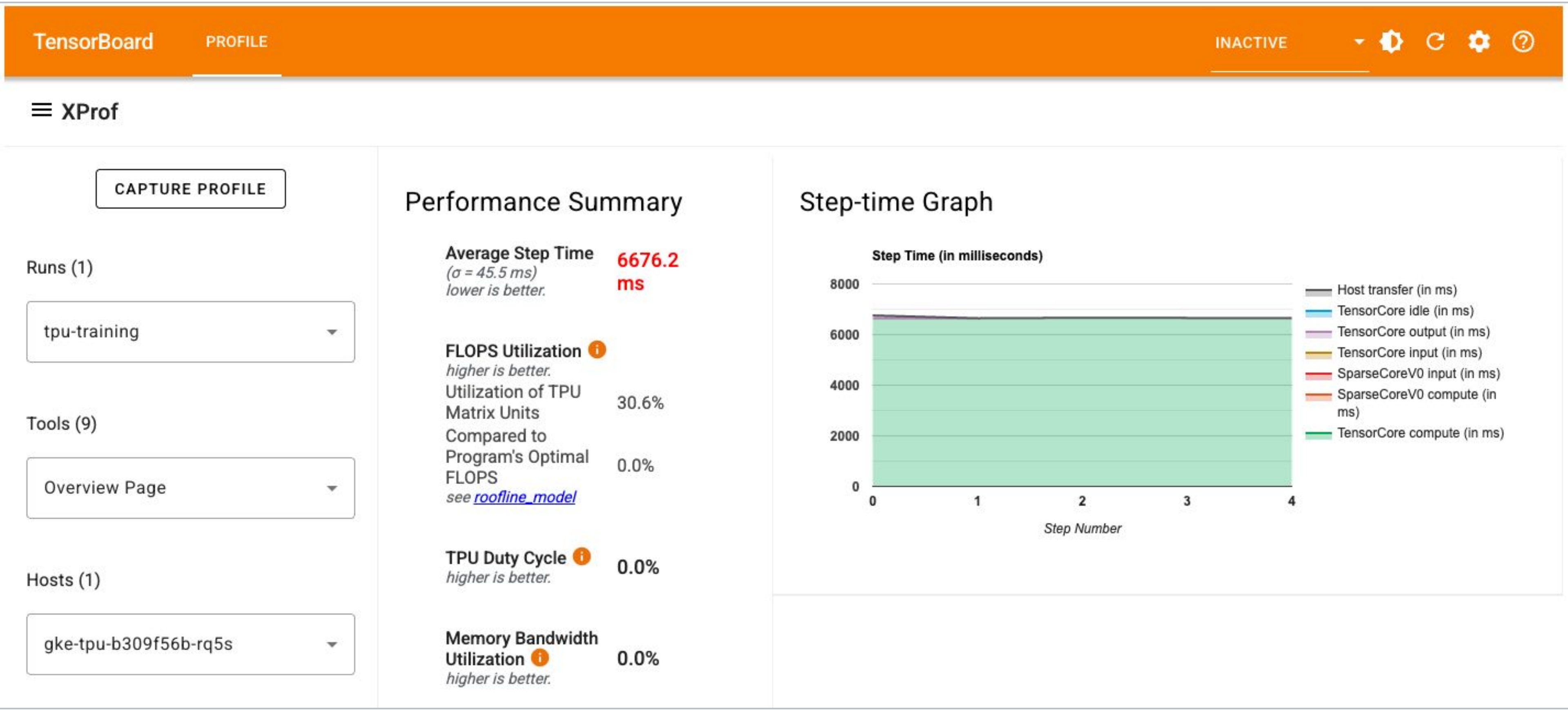
- Overview Page, Framework Op Stats, Graph Viewer, HLO Op Stats, Memory Profile, Memory Viewer, HLO Op Profile, Roofline Model, Trace Viewer

```
!pip install -Uq tensorboard tensorboard_plugin_profile

jax.profiler.start_trace(LOG_DIR) # Capturing trace for xprof

// Some training loop code

jax.profiler.stop_trace()
```

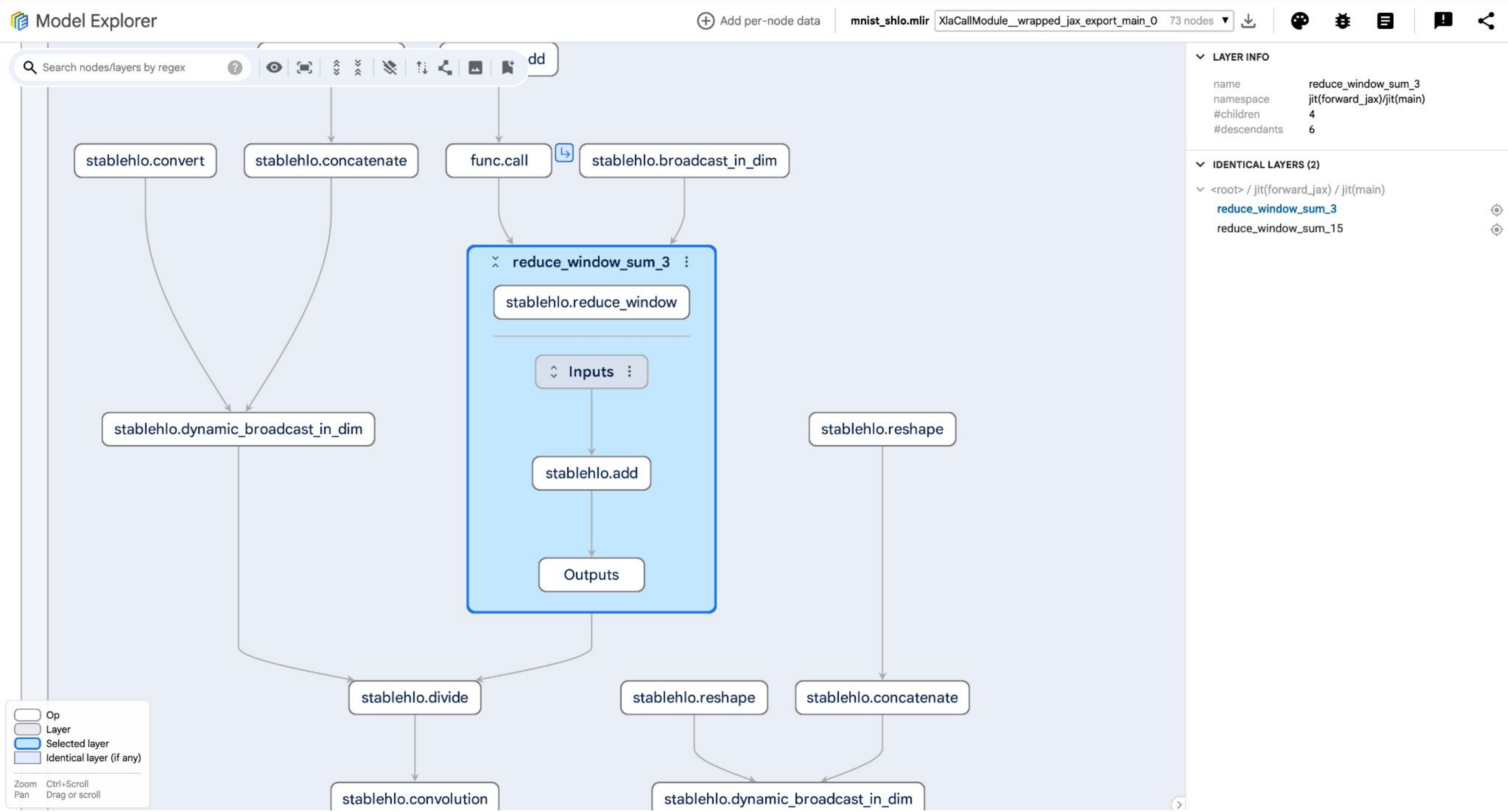

TensorBoard
PROFILE
INACTIVE
XProf
CAPTURE PROFILE
Runs (1)
tpu-training
Tools (9)
Overview Page
Hosts (1)
gke-tpu-b309f56b-rq5s
Performance Summary
Average Step Time
(σ = 45.5 ms)
lower is better.
6676.2 ms
FLOPS Utilization
higher is better.
Utilization of TPU Matrix Units
30.6%
Compared to Program's Optimal FLOPS
0.0%
see roofline_model
TPU Duty Cycle
higher is better.
0.0%
Memory Bandwidth Utilization
higher is better.
0.0%
Step-time Graph
Step Time (in milliseconds)
8000
6000
4000
2000
0
0
1
2
3
4
Step Number
Host transfer (in ms)
TensorCore idle (in ms)
TensorCore output (in ms)
TensorCore input (in ms)
SparseCoreV0 input (in ms)
SparseCoreV0 compute (in ms)
TensorCore compute (in ms)

Digging into your model with Model Explorer

Exploring the Model with Model Explorer

Model Explorer is a powerful graph visualization tool designed to help you understand and debug your ML models.

- Hierarchical Layout
  - View your model's structure clearly with nested layers that you can expand and collapse as needed.
- Metadata Overlays
  - Gain insights by overlaying metadata (e.g., attributes, inputs/outputs, etc) and custom data (e.g. performance) on nodes.
- Powerful Interactive Features
  - Focus on specific areas with search, navigate graphs smoothly with bookmarks and layer popups, customize node styles with queries, compare graphs side-by-side, and more.


Model Explorer
Add per-node data
mnist_shlo.mlir
XlaCallModule__wrapped_jax_export_main_0
73 nodes
Search nodes/layers by regex
dd
stablehlo.convert
stablehlo.concatenate
func.call
stablehlo.broadcast_in_dim
reduce_window_sum_3
stablehlo.reduce_window
Inputs
stablehlo.add
Outputs
stablehlo.dynamic_broadcast_in_dim
stablehlo.reshape
stablehlo.divide
stablehlo.reshape
stablehlo.concatenate
stablehlo.convolution
stablehlo.dynamic_broadcast_in_dim
Op
Layer
Selected layer
Identical layer (if any)
Zoom Ctrl+Scroll
Pan Drag or scroll
LAYER INFO
name reduce_window_sum_3
namespace jit(forward_jax)/jit(main)
#children 4
#descendants 6
IDENTICAL LAYERS (2)
<root> / jit(forward_jax) / jit(main)
reduce_window_sum_3
reduce_window_sum_15

Bridging from PyTorch:

Similarities, Key Differences, & Adaptations

Mapping Your PyTorch Skills to JAX/NNX

Your PyTorch Toolkit:

- **`print()`** for values
- **`pdb / breakpoint()`** for interactive steps
- Manual **`NaN/Inf`** checks
- **`print(model)`** for structure
- TensorBoard
- Hooks for intermediate values
- Profiler

Mapping Your PyTorch Skills to JAX/NNX

JAX/Flax NNX Analogues:

- **Value Inspection**: `jax.debug.print()` (in JIT), `print()` (if using `jax.disable_jit`)
- **Interactive Debugging**: `jax.debug.breakpoint` (in JIT), `pdb` or IDE (if using `jax.disable_jit`)
- **`NaN/Inf` Checks**: `chex.assert_tree_all_finite` (with `chex.chexify`)
- **Structure Inspection**: `nnx.display(model)`
- **Monitoring**: TensorBoard (similar setup/usage)
- **Assertions**: Chex (`assert_shape`, etc.)

Key Differences & Necessary Adaptations

- JIT is King: The biggest change. Debugging often means choosing:
  - Use JAX-specific tools (`jax.debug.*`, Chex + `chex.chexify`) to work within JIT.
  - Temporarily disable JIT (`jax.disable_jit`) to use standard tools, sacrificing performance.

Key Differences & Necessary Adaptations

- Hooks vs. Functional Style: PyTorch uses hooks heavily. JAX/NNX leans towards:
    - Returning intermediate values explicitly from functions (e.g., using `has_aux=True` in `jax.grad`).
    - Using `jax.debug.callback` for more complex host interactions.
    - Transforming functions (like gradient transforms) instead of in-place modification.

Key Differences & Necessary Adaptations

- **State Management:** NNX uses explicit state. Optimizer updates are now more functional (`optimizer.update(model, grads)`), making data flow clearer and easier to debug.
- **Error Messages**: JIT compilation can sometimes obscure error origins. Using `jax_debug_nans`, Chex, or `jax.disable_jit` helps pinpoint issues more accurately.

Recommended Debugging Workflow

A Systematic Approach to JAX/NNX Debugging

Static Checks First:

- Use **`nnx.display()`** to verify model/optimizer structure and initial state.
- Add Chex static assertions (**`assert_shape, assert_type`**) liberally.
- Manually check input/output shapes outside JIT.

A Systematic Approach to JAX/NNX Debugging

Runtime Issues within JIT:

- `NaN/Inf`? Use `chex.assert_tree_all_finite(model) + @chex.chexify`, or enable `jax_debug_nans.`
- Inspect values: Use `jax.debug.print`.
- Interactive step-through: Use `jax.debug.breakpoint`.

A Systematic Approach to JAX/NNX Debugging

Complex Issues / Need Standard Tools:

- Temporarily use with **`jax.disable_jit()`**: around the problematic code.
- Use standard **`print()`** and **`pdb/breakpoint()`**, or IDE debugger.

A Systematic Approach to JAX/NNX Debugging

Performance Issues:

- Suspect re-compilation? Use `chex.assert_max_traces`.
- Bottlenecks? Use `jax.profiler` (+ TensorBoard).

A Systematic Approach to JAX/NNX Debugging

Monitor Continuously:

- Integrate TensorBoard early for logging metrics and visualization.

Conclusion

- Debugging JAX/NNX requires adapting from PyTorch's eager-first model due to JIT.
- Master the JAX toolkit: `jax.debug.print, jax.debug.breakpoint`.
- Know when to use the "escape hatch": `jax.disable_jit`.
- Leverage Flax NNX's inspectability: `nnx.display()`.
- Build robustness with Chex assertions (static, value + `chex.chexify`).
- Monitor effectively with TensorBoard.
- Choose the right tool for the specific debugging task.

Learning Resources

Code Exercises, Quick References, and Slides

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

Learn 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