Enhancing Reliability:

Integrating Chex with JAX and Flax NNX

Building More Robust High-Performance Applications

JAX: Incredible Power, Subtle Complexity

- JAX provides powerful function transformations (**`jit()`, `vmap()`, `grad()`**).
- These enable high performance on accelerators (GPUs/TPUs).
- Transformations operate on traced code (abstract values).
- Errors (e.g., shape mismatches, dtype issues) can become obscured, surfacing late in execution within compiled code.
- Debugging standard Python assert behaviour within traced/compiled functions can be uninformative or even impossible, e.g. when accessing values of abstract tensors during tracing.

Chex: Utilities for Robust JAX Development

- A dedicated library specifically for JAX users.
- Focuses on enhancing reliability and simplifying development.
- Core Pillars:
    - **Instrumentation**: Runtime assertions for data validation (shapes, types, values).
    - **Testing**: Utilities for comprehensive testing across JAX modes (e.g., `jit()` vs. `non-jit()`).

Assertions: The Core of Chex Instrumentation

Key Assertion Functions:

- **`chex.assert_shape(x, expected_shape)`**: Verifies array shape (supports **`None, ...`**). Crucial for JAX!
- **`chex.assert_type(x, expected_type)`**: Checks array dtype.
- **`chex.assert_rank(x, expected_rank)`**: Checks number of dimensions.
- **`chex.assert_scalar(x)`**: Checks for shape ().
- **`chex.assert_trees_all_close(t1, t2, … tn) / _all_equal(t1, t2)`**: Compares nested PyTrees (params, states).
- **`chex.assert_tree_all_finite(tree)`**: Checks for **`NaN/Inf`** in all arrays within a PyTree. Essential for numerical stability.

Designed for JAX's Execution Model

- Standard Python assert operates on concrete values.
- During JAX tracing (`jit()`, `vmap()`), functions often see abstract values (tracers).
- Standard assert:
    - Will not see actual values, and may not work correctly on tracers.
    - Be removed during compilation.
- Chex assertions:
    - Reliably inspect both abstract tracers (during tracing/compilation) AND concrete values (during runtime).
    - Provide clear, JAX-specific error messages.
    - Act as explicit, executable documentation of data assumptions.

Google

Chex vs. Common PyTorch Validation

PyTorch Approach (Often Ad-Hoc):

- Standard Python `assert x.shape == expected_shape`
- Manual print(`x.shape, x.dtype`) for debugging.
- Checking for `NaNs`: `torch.isnan(tensor).any()`
- Using Python debuggers (pdb, IDE).

Chex Provides (Structured & JAX-Aware):

- Dedicated Functions: `chex.assert_shape`, `chex.assert_type`, etc.
- JAX Transformation Compatibility: Works reliably inside `jit()`, `vmap()`.
- PyTree Support: Built-in checks for nested structures (`assert_trees_`...).
- Clear Error Messages: Specifically designed for JAX data structures.

Essential Validation within JAX Transformations

Why it's Crucial:

- **`@jax.jit`**: Ensures assumptions made during compilation hold at runtime. Catches unexpected shape/type changes inside compiled code.
- **`@jax.vmap`**: Validates shapes before vectorization (batched input), inside the mapped function (single item), and after vectorization (batched output).

Chex assertions provide safety nets that work correctly in these contexts.

Chex Assertions Inside @jax.jit

```python
@jax.jit
def process_data_jitted(x: chex.Array, y: chex.Array) -> chex.Array:
  """Processes two arrays under JIT, asserting shapes and types."""
  # Assertions work correctly within a jitted function
  chex.assert_shape(x, (3, 4)) # Check input x shape
  chex.assert_type(x, jnp.float32) # Check input x type
  chex.assert_shape(y, (4,))  # Check input y shape

  result = jnp.dot(x, y) # Shape: (3,)

  # Assert output shape
  chex.assert_shape(result, (3,))
  chex.assert_rank(result, 1)

  return result
```

Google

Chex Assertions Inside @jax.jit

```
# Example valid call
key = jax.random.PRNGKey(1)
x_valid = jax.random.normal(key, (3, 4), dtype=jnp.float32)
y_valid = jax.random.normal(key, (4,), dtype=jnp.float32)
output = process_data_jitted(x_valid, y_valid)
print(f"JIT assertion passed. Output shape: {output.shape}")

x_invalid = jax.random.normal(key, (4,4), dtype=jnp.float32)
output = process_data_jitted(x_invalid, y_valid)
```

```
AssertionError: [Chex] Assertion assert_shape failed: Error in shape compatibility check:
input 0 has shape (4, 4) but expected (3, 4).
```

Google

Multi-Level Validation with @jax.vmap

```
def process_single_item(item: chex.Array) -> chex.Array:
  """Processes a single item (e.g., shape (10,))."""
  # Assert shape for a SINGLE item inside vmap's logic
  chex.assert_shape(item, (10,))
  result = item * 2.0
  chex.assert_shape(result, (10,)) # Check single item output
  return result

# Vectorize the function
process_batch = jax.vmap(process_single_item, in_axes=0, out_axes=0)
```

Multi-Level Validation with @jax.vmap

```
# Example usage
key = jax.random.PRNGKey(2)
batch_size = 5
batch_input = jax.random.normal(key, (batch_size, 10))

# Assert shape of the full BATCHED input BEFORE vmap
chex.assert_shape(batch_input, (batch_size, 10))

batch_output = process_batch(batch_input)

# Assert shape of the full BATCHED output AFTER vmap
chex.assert_shape(batch_output, (batch_size, 10))
print(f"Vmap assertion passed. Output shape: {batch_output.shape}")
```

Disabling Assertions

When moving your code to production, or initializing unit tests, you can enable or disable Chex assertions globally.

```
import chex

# Disable all Chex assertions
chex.disable_asserts()


# Re-enable Chex assertions
chex.enable_asserts()
```

Checking Runtime Values Inside JIT: `@chex.chexify`

Warning: @chex.chexify currently does not work in a Colab notebook

The Challenge: Checking Values Inside `@jax.jit`

- `@jax.jit` first traces functions with **abstract values** (Tracers) representing shapes/dtypes, not concrete data.
- Python control flow based on **concrete values** (e.g., `if jnp.all(x > 0): ...` or `assert jnp.sum(x) < 10`) causes errors during tracing because the condition's outcome isn't known abstractly.
- JAX needs to determine a single execution path to compile.

Solution: `@chex.chexify`

A function **decorator** (`@chex.chexify`)

- **Enables runtime value checking** inside JAX-transformed functions (`jit()`, `vmap()`, `shard_map()`).

How it works:

1. Traces the function normally for JAX compilation (ignoring value-based Python checks).
2. During actual runtime execution (when concrete values are available), it allows Chex `assert` statements (`assert_trees_all_close()`, `assert_tree_finite()` etc.) based on **tensor values** to execute.

Solution: `@chex.chexify`

Key Distinction:

- `chex.assert_shape/type/rank`: Check properties compatible with JAX tracing. Efficient.
- **Primarily a debugging tool.**

Code Example - chex.chexify for Value Checks

```
import jax
import jax.numpy as jnp
import chex

@chex.chexify
@jax.jit
def check_finite_output(x):
  # These value assertions work inside jit because of chex.chexify
  chex.assert_tree_all_finite(x)
  y = x * 2
  chex.assert_tree_all_finite(y)
  return y
```

Code Example - chex.chexify for Value Checks

```python
# Example usage
err, result = check_finite_output(jnp.array([1.0, 2.0]))
chex.block_until_chexify_assertions_complete()
print('Finite call ok')

try:
  err_nan, result_nan = check_finite_output(jnp.array([1.0, jnp.nan]))
  chex.block_until_chexify_assertions_complete()
except Exception as e:
  print(f'Oops, exception: {str(e)}') # Print the error message
```

```
Finite call ok
Oops, exception: [Chex] chexify assertion 'assert_tree_all_finite' failed: Tree contains
non-finite value: nan.
```

Usage & Caveats - `@chex.chexify()`

When to Use `@chex.chexify`

- **Debugging**: Add temporary checks for complex invariants based on values inside `jit()`, `vmap()`, `shard_map()`.
- **Verifying Intermediate Results**: Ensure calculations within a transformed function meet specific numerical criteria (e.g., positivity, bounds, non-NaN).
- **Testing**: Validate internal algorithm states during tests.

Usage & Caveats - `chex.chexify()`

Caveats:

- **Doesn't currently work in Colab.**
- **Performance Overhead**: The potential double execution (trace + runtime) makes it significantly slower than standard Chex assertions. **Not suitable for performance-critical production code**.
- **Debugging Aid**: Primarily intended for finding bugs during development, not for permanent validation in deployment.
- **Complements Assertions**: Use `chex.assert_shape/type` for standard property checks; use `@chex.chexify` for specific **value-based** logic checks when needed for debugging.

Debugging Performance:

Detecting Recompilation with `@chex.assert_max_traces()`

Google

Concept - JAX Tracing & Recompilation Problem

`@jax.jit()` compiles Python functions for high performance.

- Compilation happens **once per unique input structure** (shapes, dtypes, static arguments). This process is called **tracing**.
- If a Jitted function receives inputs with a new structure, JAX re-traces and re-compiles it.
- **Recompilation is slow!** Frequent recompilation kills performance.

Concept - JAX Tracing & Recompilation Problem

Why Frequent Recompilation Happens:

- Passing arrays with varying shapes unintentionally (e.g., changing batch size often, different padding).
- Using Python scalars/strings/tuples derived from data inside the function, making them dynamic from JAX's perspective.
- Subtle changes in PyTree structure.

Concept - JAX Tracing & Recompilation Problem

`@chex.assert_max_traces()`:

- A decorator (**`@chex.assert_max_traces(n=...)`**).
- Monitors how many times a specific function is traced (recompiled).
- Raises an **`AssertionError`** if the number of traces **exceeds n**.
- Primarily a **debugging and testing tool**.

Code Example - `@chex.assert_max_traces()`

```
import jax
import jax.numpy as jnp
import chex
import functools

chex.clear_trace_counter() # Required for running multiple times

# --- Scenario 1: Works as expected ---
@functools.partial(jax.jit, static_argnums=(1,))
@chex.assert_max_traces(n=1) # Expect only ONE compilation for a static shape
def process_fixed_shape(x: chex.Array, shape_tuple: tuple):
    chex.assert_shape(x, shape_tuple)
    return x * 2.0
```

Code Example - `@chex.assert_max_traces()`

```
print("Scenario 1: Calling with consistent shape")
fixed_shape = (3, 4)
input_data = jnp.ones(fixed_shape)
# First call -> Traces (Count = 1)
output = process_fixed_shape(input_data, fixed_shape)
print(f"First call successful. Output shape: {output.shape}")
# Second call -> Reuses cache (Count = 1)
output = process_fixed_shape(input_data + 1, fixed_shape)
print("Second call successful (used cache).")
```

Code Example - `@chex.assert_max_traces()`

```
# --- Scenario 2: Triggers Error
# (if not static_argnums or called differently) ---
@jax.jit # No static_argnums this time
@chex.assert_max_traces(n=1)
def process_dynamic_shape(x: chex.Array):
    # Shape varies, forcing re-compilation if not handled carefully
    return x + jnp.sum(x)
```

Code Example - `@chex.assert_max_traces()`

```python
print("\nScenario 2: Calling with varying shapes (demonstrates re-tracing)")
try:
    print("Calling with shape (2, 2)")
    # First call -> Traces (Count = 1)
    process_dynamic_shape(jnp.ones((2, 2)))
    print("Calling with shape (3, 3)")
    # Second call -> Re-traces (Count = 2) -> ERROR!
    process_dynamic_shape(jnp.ones((3, 3)))
except AssertionError as e:
    print(f"\nCaught expected error:\n{e}")
```

Usage & Benefits - `@assert_max_traces()`

When to Use `@assert_max_traces()`

- **During Development**: Wrap key `@jit()`-compiled functions (especially training steps, model forward passes) to ensure they aren't recompiling unexpectedly.
- **In Unit/Integration Tests**: Verify that functions compile a fixed number of times under expected usage patterns.
- **Debugging Performance Issues**: Helps pinpoint where costly recompilations are happening.

Usage & Benefits - `@assert_max_traces()`

Benefits:

- **Catches Performance Regressions**: Prevents silent slowdowns caused by accidental recompilations.
- **Improves Understanding**: Forces you to think about which inputs are static vs dynamic for `@jit()`.
- **Early Error Detection**: Finds issues related to unstable input structures sooner.

Note: Generally removed or disabled in production code, as the overhead (though small) isn't needed and legitimate recompilations might occur (e.g., handling different batch sizes explicitly).

Chex & Flax NNX

Leveraging Chex within Flax NNX Models

Rationale:

- Neural networks involve complex data flow through layers.
- Ensuring correct shapes/types/values is critical for:
  - Correct layer connections (output shape matches next input shape).
  - Validating model input matches expectations.
  - Debugging training issues (`NaNs` in activations/gradients).

Leveraging Chex within Flax NNX Models

Integration Points in nnx.Module:

- **`__init__`**: Validate static configuration arguments (less common for Chex).
- **`__call__`**: Primary location. Validate inputs, intermediate activations, and final outputs.

Flax NNX Example 1: Input/Output Validation

```
class SimpleMLP(nnx.Module):
  def __init__(self, din: int, dmid: int, dout: int, *, rngs: nnx.Rngs):
    self.linear1 = nnx.Linear(din, dmid, rngs=rngs)
    self.linear2 = nnx.Linear(dmid, dout, rngs=rngs)
```

Flax NNX Example 1: Input/Output Validation

```
def __call__(self, x: chex.Array) -> chex.Array:
  # Validate input: Expecting [batch, features]
  chex.assert_rank(x, 2) # Must be 2D
  chex.assert_axis_dimension(x, 1, self.linear1.in_features) # Check dim
  chex.assert_type(x, jnp.float32) # Check type

  # Forward pass
  x = self.linear1(x)
  x = nnx.relu(x)
  x = self.linear2(x)

  # Validate output: Expecting [batch, out_features]
  chex.assert_rank(x, 2)
  chex.assert_axis_dimension(x, 1, self.linear2.out_features)

  return x
```

Google

… And much more

Chex has a lot more than just assertions:

- **Dataclasses**: Chex provides a JAX-friendly dataclass implementation.
- **Warnings**: Chex also offers utilities to add common warnings, such as specific types of deprecation warnings.
- **Test variants**:
    - `@chex.variants(with_jit=True, without_jit=True)`
- **Faking multi-device test environments**: Fake a real multi-device environment with a multi-threaded CPU

Chex: A Best Practice for Reliable JAX Development

Recap:

- Chex provides essential validation tools tailored for JAX.
- Assertions (shape, type, rank, tree) are key.
- Works seamlessly with JAX transformations (`jit(), vmap()`).
- Integrates naturally into Flax NNX models (`__call__`) and training loops.

Benefits:

- **Improved Reliability**: Catch data errors early.
- **Enhanced Debugging**: Clearer, JAX-specific error messages.
- **Clearer Code Intent**: Assertions act as executable documentation.

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