Efficient Data Loading

for JAX/Flax NNX with Grain

A Guide for PyTorch Users

Why Specialized Data Loading for JAX?

- **JAX is Fast**: Excels at parallel computation on accelerators (GPUs/TPUs).
- **Data Bottlenecks**: Standard Python loading (I/O, CPU transforms, GIL) can't keep up.
- **Starving Accelerators**: Inefficient data pipelines waste JAX's potential.
- **PyTorch Analogy**: Like `torch.utils.data.DataLoader`, Grain optimizes this for JAX.

What is Grain?

- **Purpose-Built for JAX**: Google's library for efficient data reading & preprocessing.
- **Core Goals**: Speed (multiprocessing, shared memory), Determinism (reproducibility).
- **Flexible & Simple**: Aims for declarative pipeline definition and clear APIs.
- **JAX Ecosystem Focus**: Integrates naturally with JAX concepts like distributed sharding.

Grain vs. PyTorch DataLoader - Conceptual Differences

- **Similar Goals**: Both load, transform, batch, and parallelize data feeding.
- **Grain's DataLoader API**: Explicitly separates concerns:
  - `DataSource` (reading) + `Sampler` (order) + `Operations` (transforms).
- **PyTorch DataLoader**: Dataset often handles reading + initial transforms.
- **Focus**: We'll use Grain's DataLoader API, conceptually closer for PyTorch users.

The Building Blocks (`DataLoader` API)

- **`DataSource`**: Accesses individual raw data records (needs `__len__`, `__getitem__`).
- **`Sampler`**: Decides which records to load, in what order. Provides per-record random seeds.
- **`Operations`**: A list of transformations (functions/callables) applied sequentially to process records (e.g., augment, batch).

Simplified Workflow: `grain.DataLoader`

- **Orchestrator**: Combines `DataSource`, `Sampler`, `Operations`, and optionally shards for distributed training.
- **Interface**: `grain.DataLoader(data_source, operations, sampler, worker_count)`
- **Built-in Parallelism**: `worker_count=0` (sequential debug mode), `worker_count > 0` (multiprocessing for speed).
- **Ease of Use**: Streamlines setup for common data loading patterns.

Code: Setting up `grain.DataLoader`

```python
# 1. DataSource (Example: Simple in-memory source)
class MySource(grain.RandomAccessDataSource):
    def __init__(self, num_records=1000):
        self._len = num_records
    def __len__(self): return self._len
    def __getitem__(self, idx):
        idx = idx % self._len # Handle wrap-around for epochs
        # Simulate loading data, e.g., an image and label
        return {'image': np.ones((32, 32, 3), dtype=np.uint8) * (idx % 255),
                'label': idx % 10}
source = MySource()
```

Google

Code: Setting up `grain.DataLoader`

```python
# 2. Sampler (IndexSampler for shuffling and epochs)
index_sampler = grain.IndexSampler(
    num_records=len(source),
    shuffle=True,
    num_epochs=None, # Run indefinitely
    seed=42
)
```

Google

Code: Setting up `grain.DataLoader`

```
# 3. Operations (List of transformations)
# Example: Convert image to float, then batch
class ImageToFloat(grain.MapTransform):
  def map(self, x: int) -> dict:
    return {'image': x['image'].astype(np.float32) / 255.0, 'label': x['label']}

transformations = [
    ImageToFloat(),
    grain.Batch(batch_size=64, drop_remainder=True)
]
```

Code: Setting up `grain.DataLoader`

```python
# 4. DataLoader Instantiation
data_loader = grain.DataLoader(
    data_source=source,
    operations=transformations,
    shard_options=grain.sharding.NoSharding(),
    sampler=index_sampler,
    worker_count=0, # Start with 0 for debugging
    # Disables thread prefetching when dataset in memory already
    read_options=grain.ReadOptions(num_threads=0)
)
print("DataLoader configured!")
```

Code: Iterating and Enabling Parallelism

```
# --- Basic Iteration (worker_count=0) ---
print("\nIterating (worker_count=0):")
data_iterator = iter(data_loader)
first_batch = next(data_iterator)
print(f"Batch data shape: {first_batch['image'].shape}")
print(f"Batch label shape: {first_batch['label'].shape}")
```

Code: Iterating and Enabling Parallelism

```
# --- Enabling Parallelism (worker_count > 0) ---
num_workers = 4
print(f"\nReconfiguring with worker_count={num_workers}")
data_loader_parallel = grain.DataLoader(
    data_source=source,          # Same source
    operations=transformations, # Same operations
    sampler=index_sampler,       # Same sampler
    worker_count=num_workers     # > 0 enables multiprocessing
)

parallel_iterator = iter(data_loader_parallel)
first_batch_parallel = next(parallel_iterator)
print(f"Parallel batch data shape: {first_batch['image'].shape}, Label shape:
{first_batch['label'].shape}")
```

Google

Implementing Custom Logic

- **Why Custom?** Augmentation, feature engineering, specific formatting needs.
- **Deterministic**: Inherit `grain.MapTransform`, implement `map(self, element)`.
- **Random**: Inherit `grain.RandomMapTransform`, implement `random_map(self, element, rng)`. Use the provided **RNG** for reproducibility!
- **Pickling**: If `worker_count > 0`, custom transforms must be picklable (avoid complex closures, non-picklable objects).

Code: Custom Random Transformation

```python
# Example: Randomly scale data
class RandomScale(grain.RandomMapTransform):
    def random_map(self, element: dict, rng: np.random.Generator) -> dict:
        # Use the provided rng for reproducible randomness
        scale_factor = rng.uniform(0.8, 1.2)
        element['image'] = element['image'] * scale_factor
        return element
```

Code: Custom Random Transformation

```python
# --- Usage in DataLoader operations list ---
custom_ops = [
    RandomScale(), # Add instance to the list
    grain.Batch(batch_size=64, drop_remainder=True)
]

dl_custom = grain.DataLoader(data_source=source, operations=custom_ops,
                             sampler=index_sampler, worker_count=num_workers)
print("Configured DataLoader with custom random transform.")
```

Google

Scaling Out: Data Sharding

- **Why Shard?** Distributed training requires each process gets unique data slice.
- **Grain's Approach**: Configure sharding with `DataLoader.shard_options`.
- `grain.sharding.ShardOptions`: Holds `shard_index`, `shard_count`.
- `grain.sharding.ShardByJaxProcess`: Recommended helper; auto-detects index/count from JAX environment (`jax.process_index()`, etc.).

Code: Using ShardByJaxProcess

```
try:
    # Auto-detects from JAX environment (e.g., jax.process_index())
    shard_options = grain.ShardByJaxProcess(drop_remainder=True)
    print(f"Using ShardByJaxProcess: {shard_options}")
except ImportError: # Fallback if not in JAX distributed env
    print("Fallback: No sharding.")
    shard_options = grain.ShardOptions(0, 1, drop_remainder=True)

# Configure IndexSampler with these options
sampler_sharded = grain.IndexSampler(
    len(source),
    shuffle=True, seed=42
)

# Use shard_options when creating the DataLoader
```

Google

Behind the Scenes: Performance

- **Multiprocessing** (`worker_count > 0`): Bypasses GIL for parallel CPU work.
- **Shared Memory**: Efficiently transfers large NumPy arrays (e.g., batches) between processes, avoiding costly serialization.
- **Prefetching**: Workers prepare data ahead of time, hiding I/O and transform latency.
- **Asynchronous Ops**: Internal tasks run concurrently to avoid blocking data flow.

Using Grain in your Flax NNX Training Loop

- **Get Iterator**: `iterator = iter(data_loader)` from your configured Grain DataLoader.
- **Loop**:
  - `batch = next(iterator)` (Get data batch).
  - Optional: `jax.device_put(batch, ...)` for local device sharding/placement.
  - Pass batch to your Jitted JAX/Flax NNX training function.
  - Update Flax NNX state (params, optimizer) based on results.

Code: Conceptual JAX/Flax NNX Loop

```
import jax
import jax.numpy as jnp
# Assume: data_loader_sharded, TrainState, train_step

# Get Grain iterator
grain_iterator = iter(data_loader_sharded)
state = initial_state(...) # Your NNX state
```

Code: Conceptual JAX/Flax NNX Loop

```
num_steps = 10000
for step in range(num_steps):
    try: # 1. Get Batch from Grain
        batch = next(grain_iterator)
    except StopIteration: break

    # 2. Optional: Shard batch across local devices
    # batch = jax.device_put(batch, ...)

    # 3. Execute JITted training step
    state = train_step(state, batch) # Updates NNX state

    if step % 100 == 0: print(f"Step {step}...")
```

Google

Reproducibility: Checkpointing Input State

- **Problem**: Need to restore data stream position along with model weights, especially for reproducibility.
- **DatasetIterator**: Low-level API iterator has `.get_state()` / `.set_state()`.
- **Orbax Integration**: Recommended for `DataLoader`; standard JAX checkpointing library.
- **Benefit**: Saves/loads Grain iterator state alongside Flax NNX state atomically via Orbax `CheckpointManager`.

Summary & Recommendations

- **Use Grain**: Solves JAX data bottlenecks for better performance.
- **Boost Speed**: Use `DataLoader(worker_count > 0)` for parallelism.
- **Ensure Reproducibility**: Use samplers/seeds & `RandomMapTransform's` RNG.
- **Distribute**: Use `ShardByJaxProcess` in `IndexSampler` for JAX sharding.
- **Save Everything**: Checkpoint data iterator state (via Orbax) with model state.

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