Grain Quick Reference for JAX/Flax NNX

Goal: Efficient, deterministic data loading & preprocessing for JAX, avoiding CPU bottlenecks and keeping accelerators fed. Analogous role to torch.utils.data.DataLoader but purpose-built for JAX.

Core API: grain.DataLoader
Orchestrates data loading using composable building blocks.

```python
# Basic Structure
data_loader = grain.DataLoader(
    data_source: grain.RandomAccessDataSource,
    operations: List[grain.Transformation],
    sampler: grain.Sampler,
    worker_count: int = 0,  # 0 for sequential, >0 for parallel
    shard_options: grain.ShardOptions = grain.NoSharding(),
    # Disables thread prefetching when dataset in memory already
    read_options=grain.ReadOptions(num_threads=0)
)
```

Building Blocks:

1. **DataSource (e.g., grain.RandomAccessDataSource)**
    - **Purpose:** Provides indexed access to *raw* data records.
    - **Implementation:** Inherit and implement `__len__(self)` and `__getitem__(self, index)`.

Example:

```python
class MySource(grain.RandomAccessDataSource):
    def __init__(self, data_items): self._data = data_items
    def __len__(self): return len(self._data)
    def __getitem__(self, idx): return self._data[idx] # Load raw record
```

2. **Sampler (e.g., grain.IndexSampler)**

- **Purpose:** Defines the order records are accessed (shuffling, epochs) and provides per-record seeds for deterministic randomness.
- **Key Parameters:**
  - `num_records`: Total records in `DataSource`.
  - `shard_options`: Instance of `grain.ShardOptions` for distributed training (see Sharding).
  - `shuffle`: True / False.
  - `num_epochs`: Number of passes over data (None for infinite).
  - `seed`: Base random seed for shuffling and per-record seeds.

Example:

```python
sampler = grain.IndexSampler(
    num_records=len(my_source),
    shard_options=shard_options, # See Sharding section
    shuffle=True,
    num_epochs=None,
    seed=42
)
```

3. **Operations (List of `grain.Transformation`)**
   - **Purpose:** Sequential processing applied to *each record* (or batch).
   - **Common Built-ins:**
     - `grain.Batch(batch_size, drop_remainder)`: Groups records into batches.
   - **Custom Transforms:**

Deterministic: Inherit grain.MapTransform, implement map(self, element).

```python
class Normalize(grain.MapTransform):
    def map(self, data):
        data['image'] = data['image'].astype(np.float32) / 255.0
        return data
```

Random: Inherit `grain.RandomMapTransform`, implement `random_map(self, element, rng: np.random.Generator)`. Crucial: Use the provided `rng` for reproducibility!

```python
class RandomFlip(grain.RandomMapTransform):
    def random_map(self, data, rng):
        if rng.random() > 0.5:
            data['image'] = np.fliplr(data['image'])
        return data
```

Example List:

```python
ops = [Normalize(), RandomFlip(), grain.Batch(batch_size=128)]
```

Performance: Parallelism (worker_count)

- `worker_count = 0`: Sequential execution in the main process (good for debugging).
- `worker_count > 0`: Uses multiprocessing (N workers) to parallelize data reading and transformations, bypassing Python's GIL. Significantly faster for CPU-bound tasks. Uses shared memory for efficient batch transfer.
- `read_options=grain.ReadOptions(num_threads=0):` Disables thread prefetching when dataset in memory already

Distributed Training: Data Sharding

- **Purpose:** Ensure each JAX process gets a unique subset of the data.
- **How:** Configure `shard_options` in the `Sampler` and pass to `DataLoader`.

Recommended: `grain.sharding.ShardByJaxProcess()` automatically detects `jax.process_index()` and `jax.process_count()`.

```python
# In your distributed setup
try:
    # Auto-detects from JAX environment
    shard_options = grain.ShardByJaxProcess(drop_remainder=True)
except ImportError: # Fallback for single process
    shard_options = grain.ShardOptions(shard_index=0, shard_count=1,
drop_remainder=True)
```

```
# Use when creating the sampler:
sampler = grain.IndexSampler(..., shard_options=shard_options, ...)
# DataLoader will inherit sharding from sampler if not specified directly
```

Integration in JAX/Flax NNX Training Loop

```python
# 1. Create configured grain.DataLoader (parallel, sharded)
data_loader = grain.DataLoader(...)


# 2. Get iterator
data_iterator = iter(data_loader)


# 3. Training Loop
for step in range(num_steps):
    try:
        # Get batch (prefetched by Grain workers)
        batch = next(data_iterator)
    except StopIteration:
        break # Or reset iterator if num_epochs is finite


    # Optional: jax.device_put for local device sharding (if using
pmap/shard_map)
    # batch = jax.device_put(batch, ...)


    # Pass batch to JITted train step
    state = train_step(state, batch) # train_step uses batch['image'],
batch['label'] etc.
    # ... logging, etc. ...
```

Reproducibility & Checkpointing

- **Core:** Use fixed seeds (seed in Sampler) and the `rng` provided to RandomMapTransform.
- **Checkpointing:** Essential to save/restore the *data iterator's state* alongside model parameters for exact resumption.

- **Recommended:** Use **Orbax** Checkpointing. Grain provides integration points (`OrbaxCheckpointHandler` for `DataLoader`) to atomically save/load the iterator state with the rest of your training state (Flax NNX model, optimizer state).

Key Recommendations

- Use `grain.DataLoader` for simplicity and performance.
- Leverage `worker_count > 0` for speedup via multiprocessing.
- Use `grain.IndexSampler` with seeds for determinism.
- Use the `rng` in `RandomMapTransform` for reproducible augmentations.
- Use `grain.sharding.ShardByJaxProcess` for easy distributed setup.
- Checkpoint iterator state using Orbax integration for full reproducibility.

References:

- JAX AI Stack: https://jaxstack.ai
- Grain Docs: https://google-grain.readthedocs.io
- JAX: https://jax.dev
- Flax NNX: https://flax.readthedocs.io