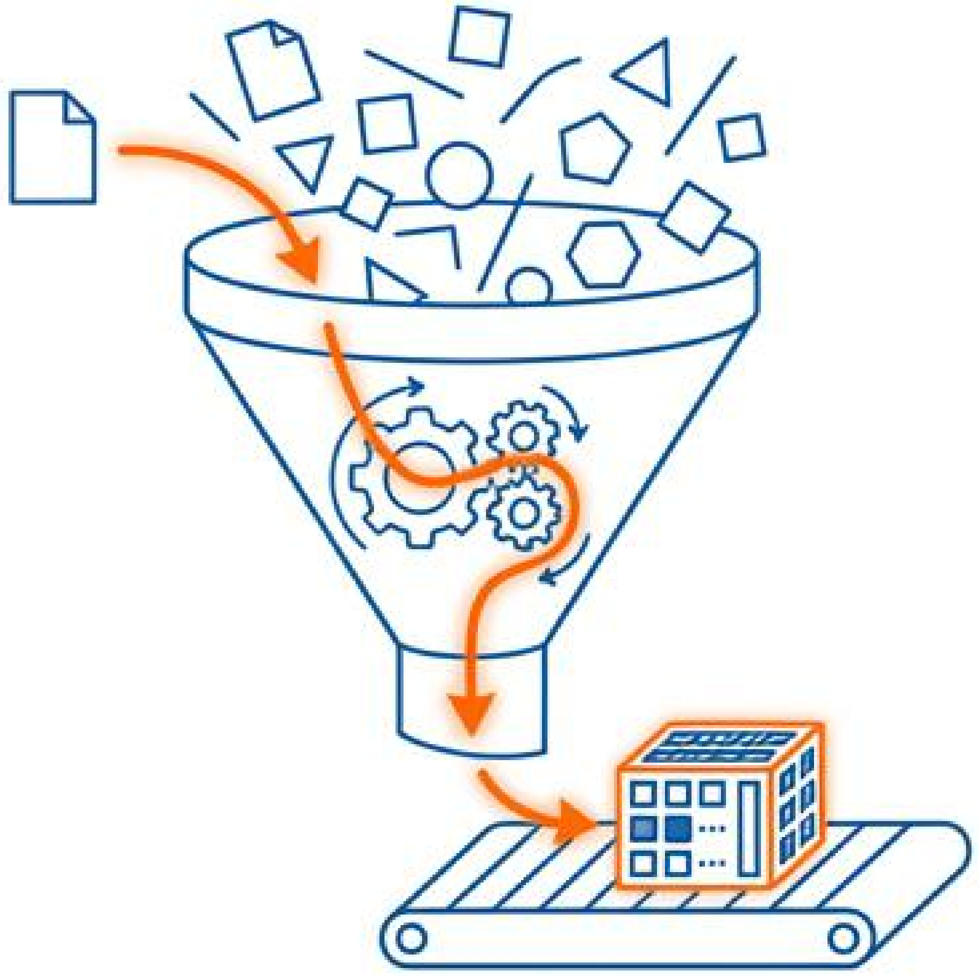
FOUNDATION TIER

MODULE 05

Data Loading

The ML infrastructure pipeline that feeds your models

Module 05

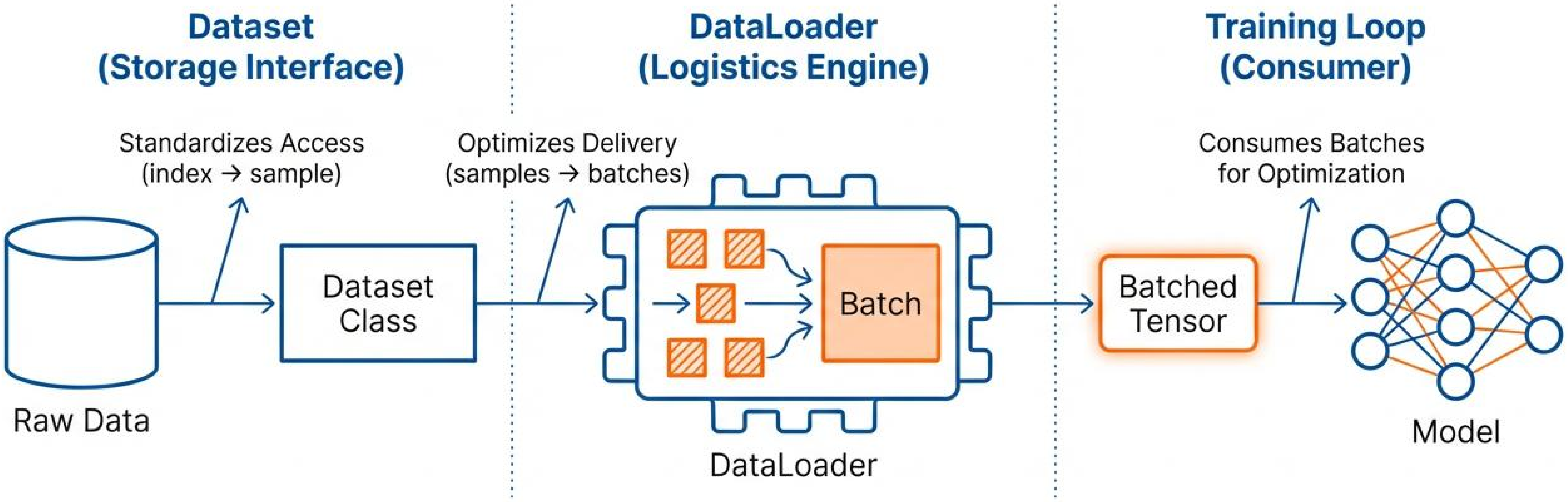


Data Loading

Building the bridge between raw storage and high-performance computation.

Tier: Foundation
Prerequisites: Tensors, Layers, Losses

The Training Data Pipeline



Dataset: Abstracts away storage. Transforms messy files into clean, single samples.

DataLoader: Handles logistics. Groups samples, manages memory, and shuffles data.

Training Loop: Agnostic to source. Simply iterates over the loader to get Tensors.

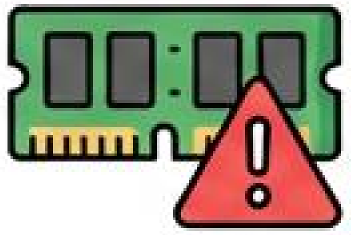
The Systems Challenge

Why naive iteration fails at scale

The Naive Approach

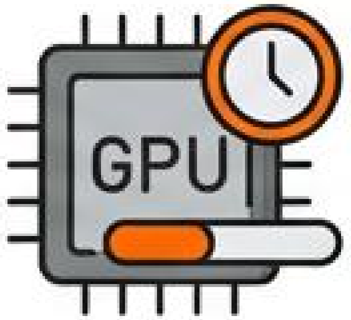
```
data = load_all_images() # 600GB? Crash.
for i in range(len(data)):
    x = data[i] # Single item
    train(x)    # GPU 99% Idle
```

The Engineering Constraints



Memory Limits

Real datasets (e.g., ImageNet 1.2M images) require ~600GB. This exceeds RAM on almost any machine. Data must be loaded lazily.



Computational Efficiency

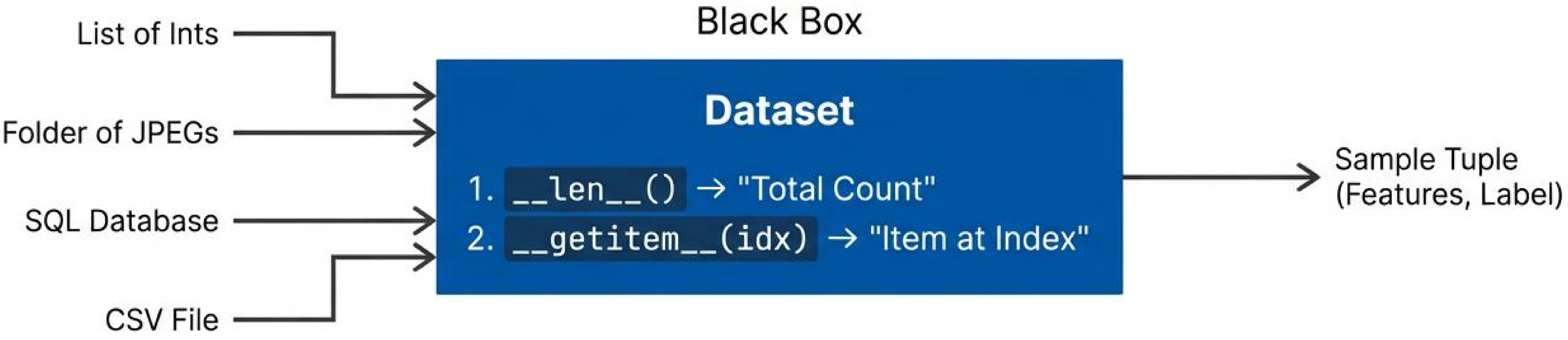
GPUs are massive parallel processors. Processing 1 sample at a time wastes 99% of throughput. We need Batching.

Statistical Correctness

Sequential training creates correlation artifacts (oscillating gradients). We need efficient Shuffling without moving gigabytes of memory.

The Dataset Abstraction

A universal contract for data access



The Invariant

The Dataset class hides the complexity of storage. The rest of the system only needs to know 'How many items?' and 'Give me item N'. This allows us to swap storage backends without changing the training code.

Code: The Dataset Interface

```python
from abc import ABC, abstractmethod

class Dataset(ABC):
    """Abstract base class for all datasets."""

    @abstractmethod
    def __len__(self) -> int:
        """Return the total number of samples."""
        pass

    @abstractmethod
    def __getitem__(self, idx: int):
        """Return the sample at the given index."""
        pass
```

Enforces Implementation. Subclasses MUST define these methods.

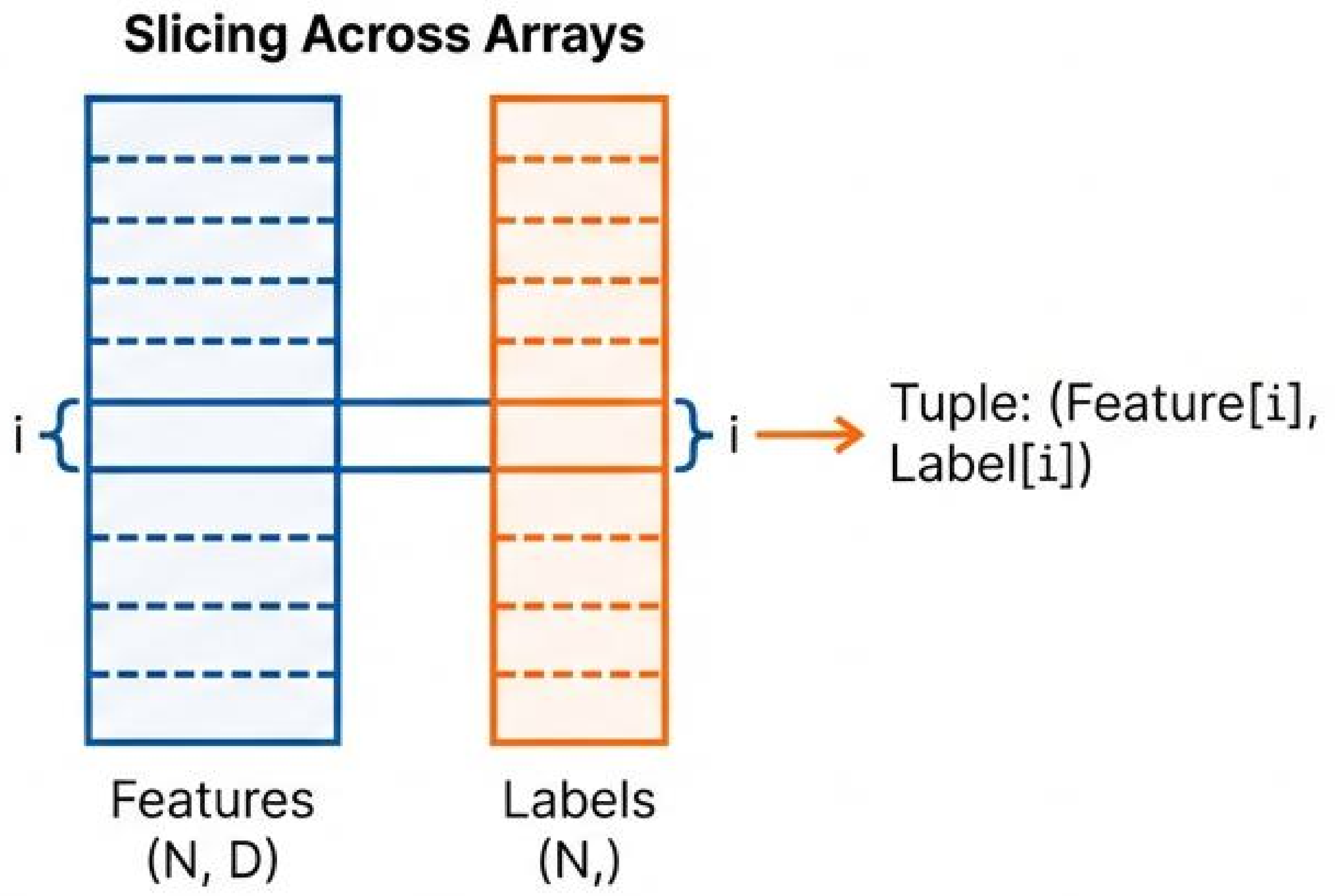
Source: `tinytorch/core/dataloader.py`
This uses Python's `ABC` (Abstract Base Class) module to define a strict interface. You cannot instantiate a raw `Dataset`; you must subclass it.

In-Memory Storage: TensorDataset

The simplest concrete implementation

Logic:

- Wraps pre-existing Tensors.
- **Constraint:** All tensors must match in Dimension 0 (Sample Dimension).
- **Access:** Slices all tensors synchronously.



Ideal for datasets that fit in RAM (e.g., MNIST, CIFAR).

Code: Implementing TensorDataset

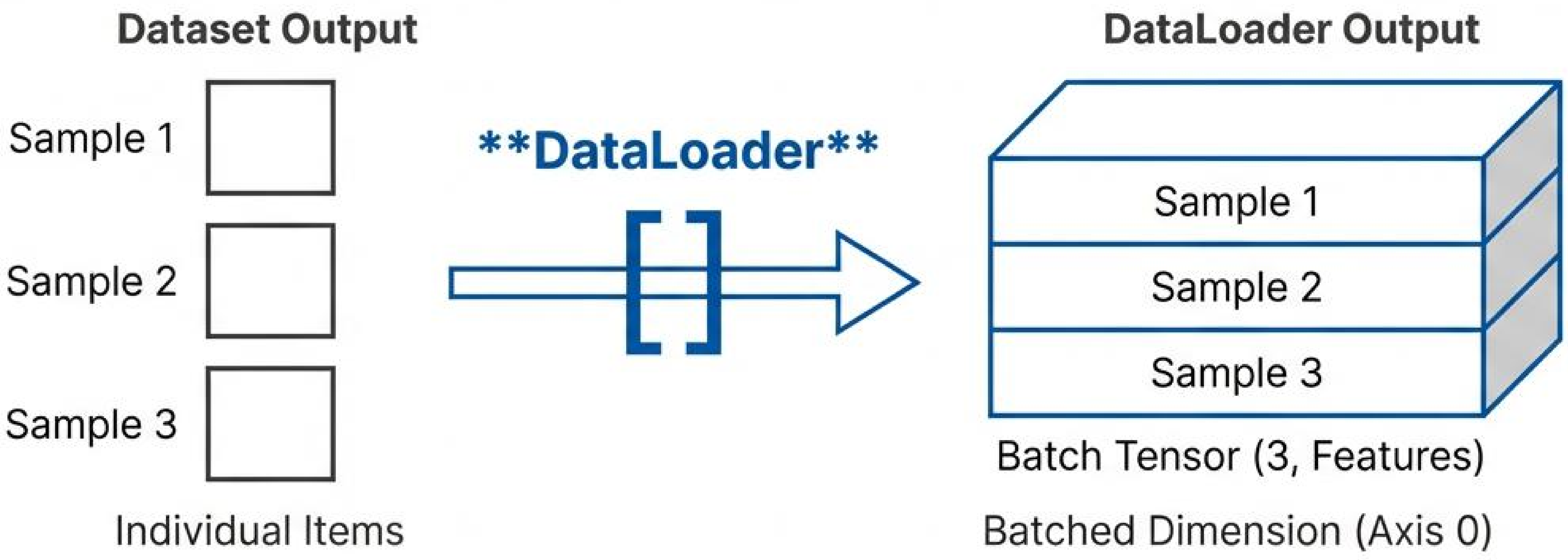
```
class TensorDataset(Dataset):
    def __init__(self, *tensors):
        # Validate all tensors have same size in dim 0
        first_size = len(tensors[0].data)
        assert all(len(t.data) == first_size for t in tensors)
        self.tensors = tensors

    def __getitem__(self, idx: int):
        # Return tuple of slices wrapped in Tensor
        return tuple(Tensor(t.data[idx]) for t in self.tensors)
```

Safety Check: Prevents misalignment (e.g., 100 images but 99 labels).

Wrapping: Ensures output remains a framework Tensor object.

The Batch Factory: DataLoader

Transitioning from Storage to Logistics



- **Responsibilities:**
  1. ****Batching****: Group N samples into one contiguous tensor.
  2. ****Shuffling****: Randomize order to break training correlations.
  3. ****Iteration****: Provide a memory-efficient stream via Python Generators.

Code: Initializing the DataLoader

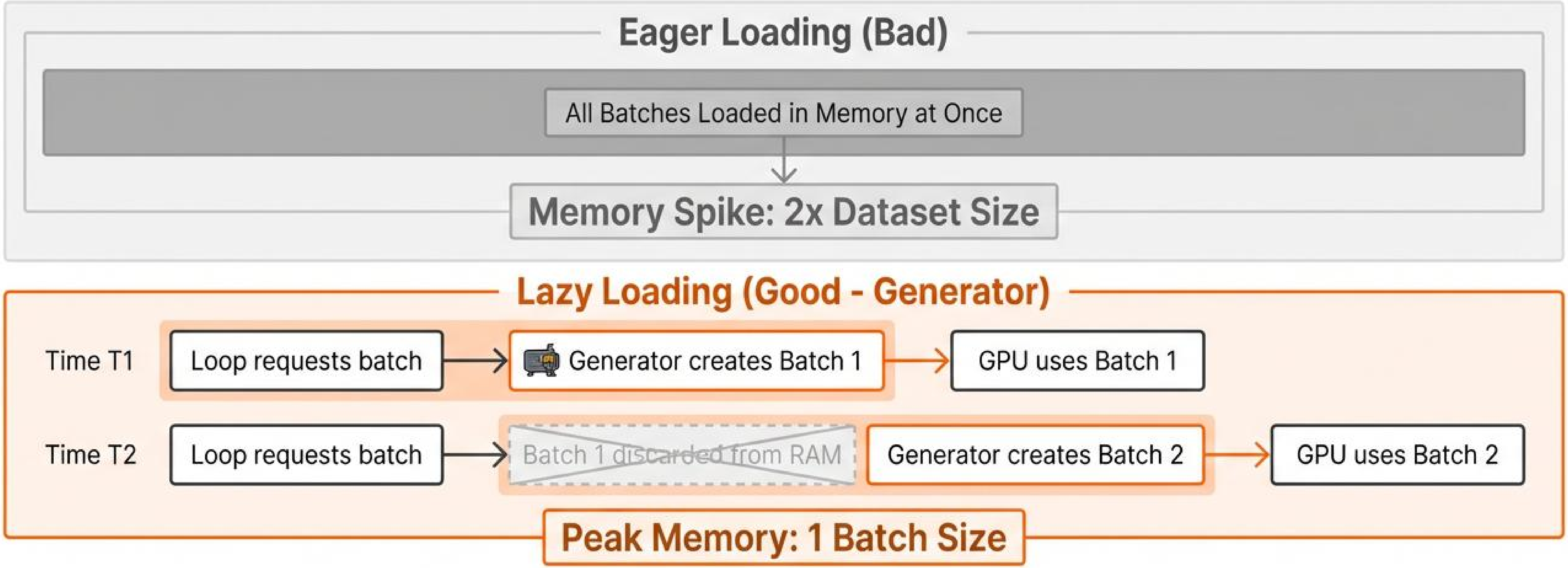
```
class DataLoader:
    def __init__(self, dataset: Dataset, batch_size: int, shuffle: bool = False):
        self.dataset = dataset
        self.batch_size = batch_size
        self.shuffle = shuffle
```

Parameters Guide

Parameter	Type	Role
dataset (JetBrains Mono)	Dataset (JetBrains Mono)	The source. Must adhere to the Dataset contract.
batch_size (JetBrains Mono)	int (JetBrains Mono)	**The Systems Knob** (in Engineering Orange #FF6600). Controls Throughput vs. Memory usage.
shuffle (JetBrains Mono)	bool (JetBrains Mono)	**TheCorrectness Knob** (in Engineering Orange #FF6600). `True´for training (generalization), `False´for validation.

Systems Insight: The Iterator Protocol

Memory Efficiency via Lazy Evaluation



Key Concept: **Lazy Evaluation**

We use Python's `yield` keyword. The batch exists in memory only for the split second the training loop needs it.

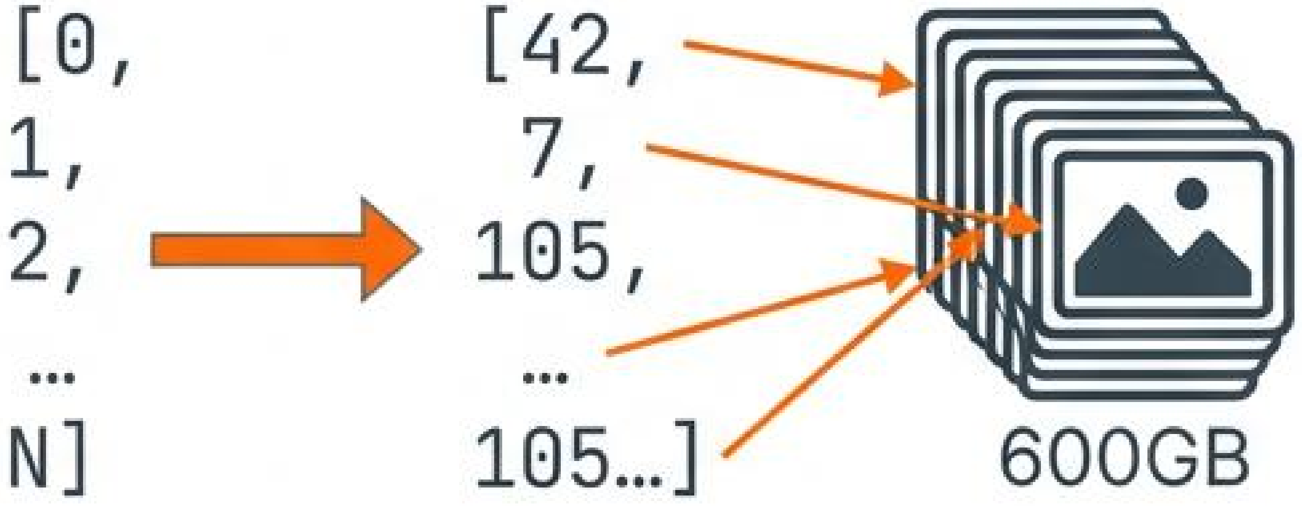
Systems Insight: Efficient Shuffling

Virtualizing the random shuffle

The Trap (Shuffling Data)

Moves Gigabytes. Very Slow.

The Solution (Shuffling Indices)



Moves Integers (KB). Instant.

```
def __iter__(self) -> Iterator:
    indices = list(range(len(self.dataset)))
    if self.shuffle:
        random.shuffle(indices) # Shuffles tiny integers, not massive images!
```

Code: The Generator Loop

```
# ... inside __iter__ ...

# Chunk the indices list
for i in range(0, len(indices), self.batch_size):

    # 1. Get batch of indices
    batch_indices = indices[i:i + self.batch_size]

    # 2. Fetch list of samples (The heavy lifting)
    batch = [self.dataset[idx] for idx in batch_indices]

    # 3. Collate and Yield
    yield self._collate_batch(batch)
```

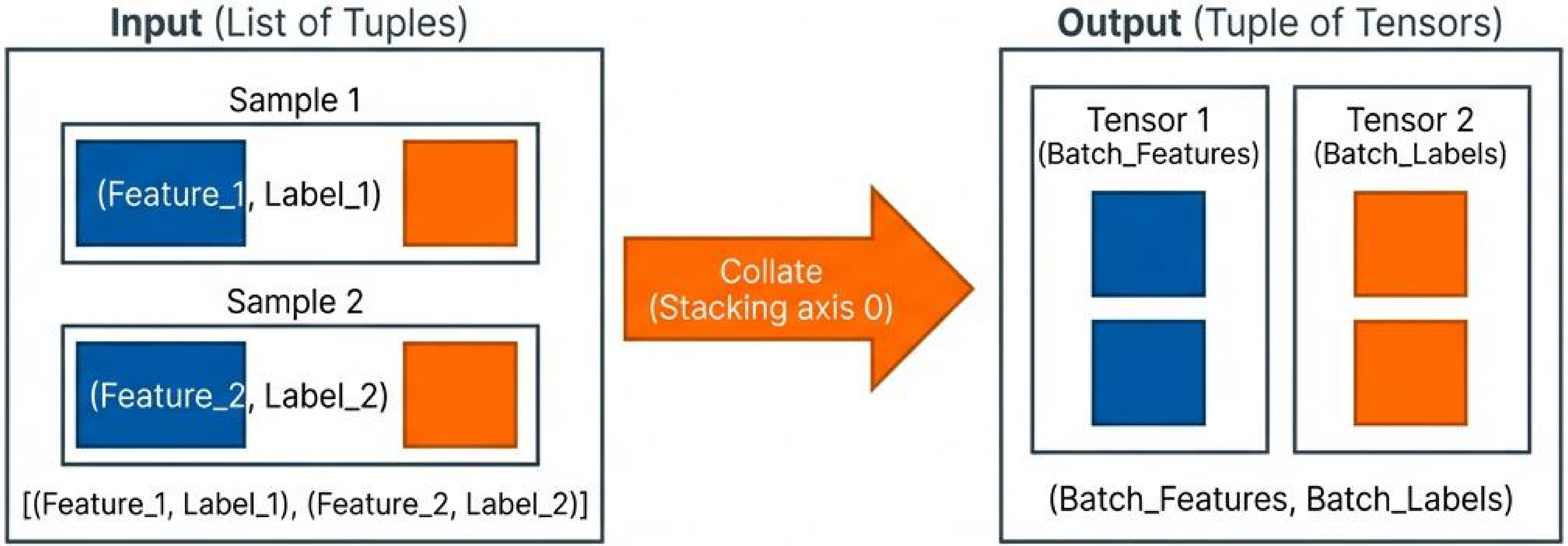
1. Slicing integers (Cheap)

2. **Accessing actual data** via Dataset contract

3. **Handing off control** to training loop

Concept: Collation

Transformation: **List of Tuples → Tuple of Tensors**



Operation: Stacking along a new dimension (Axis 0). This creates the contiguous memory block required by GPUs.

Code: Implementing Collation

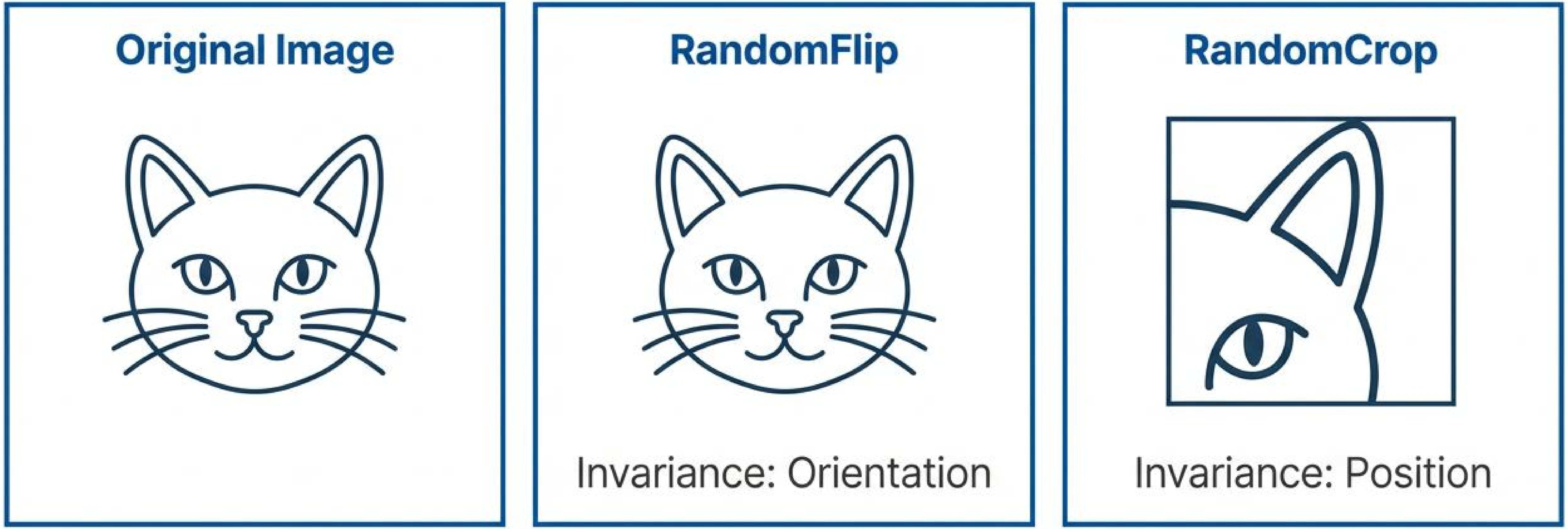
```
def _collate_batch(self, batch):
    num_tensors = len(batch[0]) # e.g., 2 for (features, labels)
    batched_tensors = []

    for i in range(num_tensors):
        # Extract all tensors at position i
        tensor_list = [sample[i].data for sample in batch]

        # Stack into batch tensor (Allocates new memory!)
        # Stack into batch tensor (Allocates new memory!)
        stacked = np.stack(tensor_list, axis=0)

        batched_tensors.append(Tensor(stacked))

    return tuple(batched_tensors)
```

1.5pt

Performance Bottleneck: This line allocates new contiguous memory and copies data. It is the most expensive CPU operation in the loader.
'Inter', Regular

Enhancing the Pipeline: Augmentation

Virtual dataset expansion



Systems Rule: Augmentation applies ONLY during training. Validation data must remain static to ensure metrics are comparable across epochs.

Code: RandomHorizontalFlip

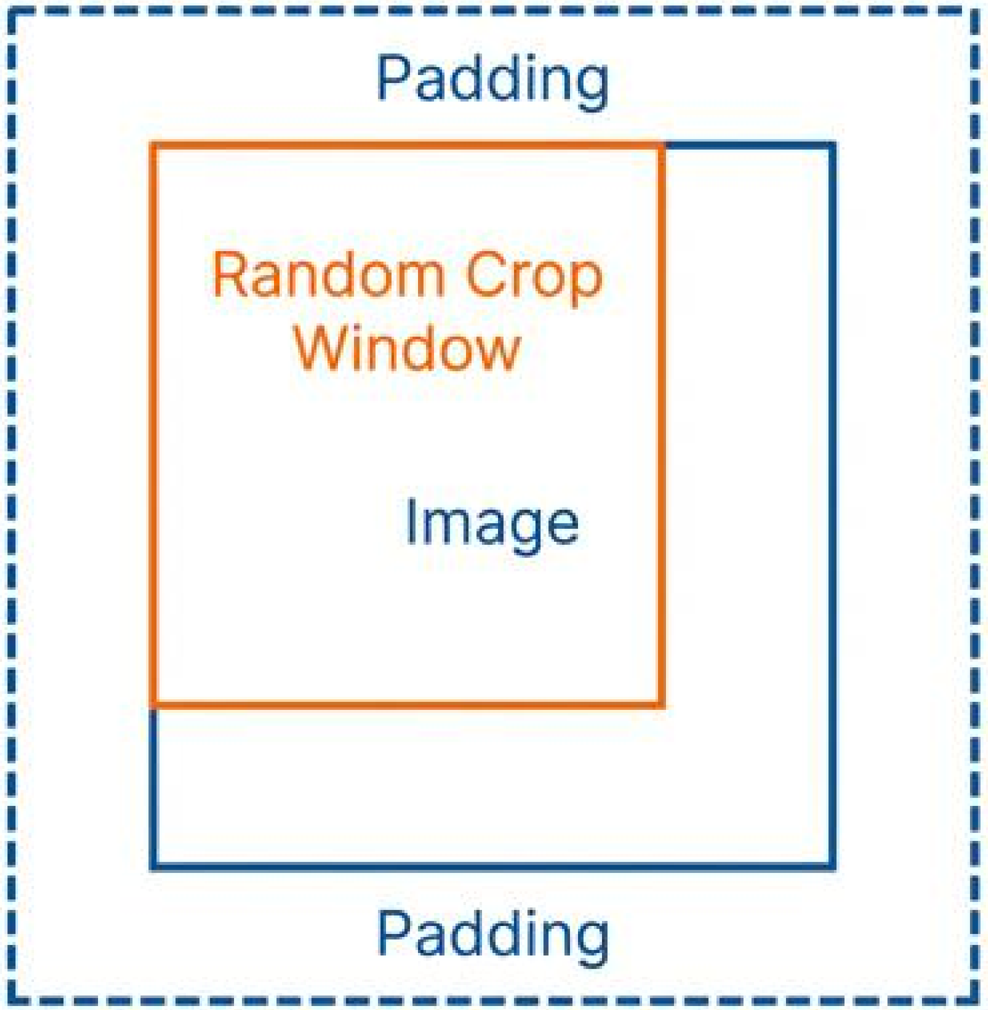
```
class RandomHorizontalFlip:
    def __init__(self, p=0.5):
        self.p = p

    def __call__(self, x):
        if np.random.random() < self.p:
            # Flip along width axis (last axis)
            return np.flip(x, axis=-1).copy()
        return x
```

Callable class pattern (functor). Dark Slate

Crucial: NumPy slicing/flipping returns a view. `copy()` forces contiguous memory, preventing errors later in the pipeline.

Code: RandomCrop

```
# Simplified logic within __call__
# 1. Pad image borders with zeros
padded = np.pad(data, self.padding, mode='constant')

# 2. Select random top-left corner
top = np.random.randint(0, 2 * self.padding + 1)
left = np.random.randint(0, 2 * self.padding + 1)

# 3. Slice back to original size
cropped = padded[..., top:top+h, left:left+w]
return cropped
```



Code: Composing Transforms

```
class Compose:
    def __init__(self, transforms):
        self.transforms = transforms

    def __call__(self, x):
        for transform in self.transforms:
            x = transform(x)  # Pipeline: output of one is input to next
        return x
```

```
transforms = Compose([
    RandomHorizontalFlip(0.5),
    RandomCrop(32, padding=4)
])
```

This matches the `torchvision.transforms.Compose` API pattern, allowing complex augmentation pipelines to be built from simple blocks.

Systems Trade-off: Batch Size

How to tune the knob

Metric	Small Batch (e.g., 32)	Large Batch (e.g., 512)
Memory	Low (Fits on Laptop)	High (Needs Cluster)
Throughput	Lower (Python overhead dominates)	Higher (Vectorized ops dominate)
Convergence	Noisy Gradients (Regularizing)	Stable Gradients (Can be too stable)

The Bottleneck

Small batches suffer from ‘Python Overhead’—the fixed cost of the `for` loop and `collate` function becomes a large percentage of total time.

Bottleneck Analysis

Where does the time go?

Diagnosis: Data Starved

The GPU is sitting idle for 45ms every step, waiting for the CPU to collate data.

Production Solution: Prefetching. (Loading Batch N+1 while GPU computes Batch N).

TinyTorch vs. PyTorch

Shared DNA (Identical Concepts)

- `Dataset` & `DataLoader` classes
- Iterator protocol (`for batch in loader`)
- Transforms design (`Compose`, Callables)
  `JetBrains Mono`
- Index-based shuffling

The Difference (Implementation)

TinyTorch

- **Single-process** (Sequential)
- Easy to debug
- Good for education

PyTorch

- **Multi-process** (`num_workers=4`)
- **Hides latency** via **prefetching**
- Harder to debug (multiprocessing errors)

By building the single-process version, you now understand exactly what the PyTorch workers are doing under the hood.

Synthesis: Concept → Code

Abstract Storage	`Dataset class` (tinytorch/core/dataloader.py)
In-Memory Data	`TensorDataset`
Batching Strategy	`DataLoader class`
Infinite Stream	`__iter__ generator`
Data Generalization	`RandomHorizontalFlip / Compose`

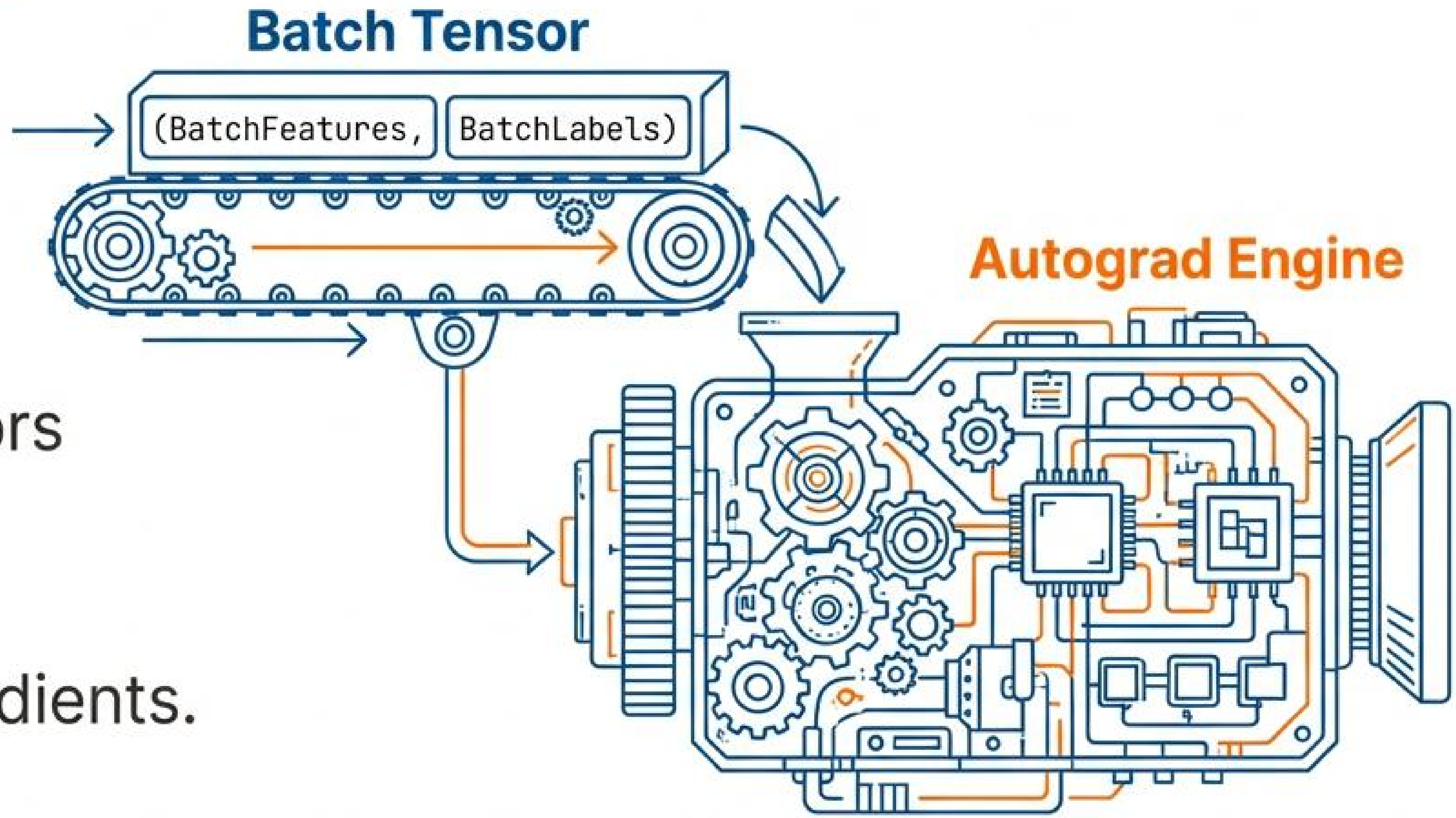
We have built the infrastructure. The Training Loop in Module 08 will simply consume this stream.

What's Next?

Module 06: Autograd

The Connection:
Now: We have (BatchFeatures, BatchLabels)`.
Next: We feed these Tensors into a computation graph.

Goal: Track operations to automatically compute gradients.



You have the fuel (Data). Now we build the engine (Autograd).