Making the Machine Learning Reconstruction Pipeline Scalable

Jack Carlton
University of Kentucky

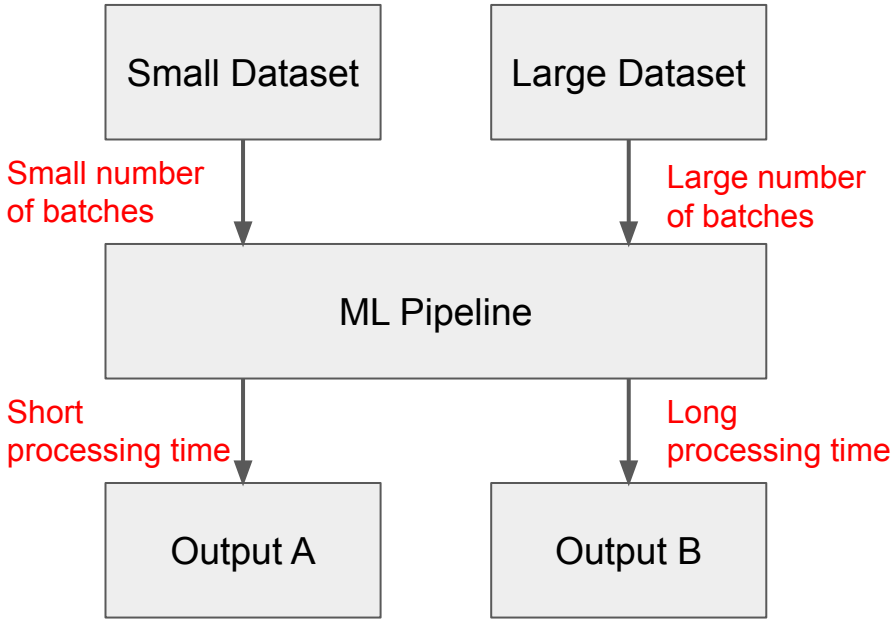
Breaking down “Scalability”

I choose to break down scalability into three categories:

1. Data
    a. ML pipeline does not change behavior as data set grows
        i. Still scales in execution time
2. Compute
    a. ML pipeline does not change behavior as compute power is changed
        i. Still scales in execution time
        ii. Ex. Pipeline runs on developer’s laptop or a CPU/GPU cluster
3. Codebase
    a. ML pipeline does not (greatly) change behavior as complexity grows
        i. New models, stages, or data products require additional code, not (major) code rewrites
        ii. Iteration speed does not (greatly) degrade with system size (i.e. keep things modular!)

What Do We Mean by “Scaling” (Data)

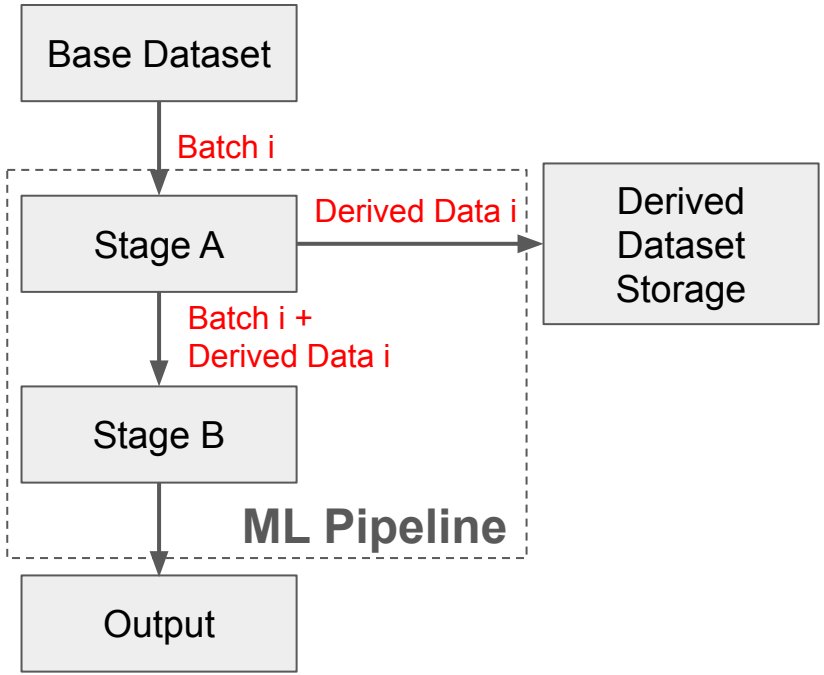
- **Data volume should be able to grow without changing how the pipeline behaves**
- Adding the following should not cause pipeline behavior changes:
  - More events
  - More derived data products
    - predictions, masks, regressions, etc.
  - More passes over the same data
  - Larger event representations
- Implications:
  - Memory usage must not grow with dataset size



Ideal Behavior for Different Sized Datasets

Techniques to Ensure Scalability (Data)

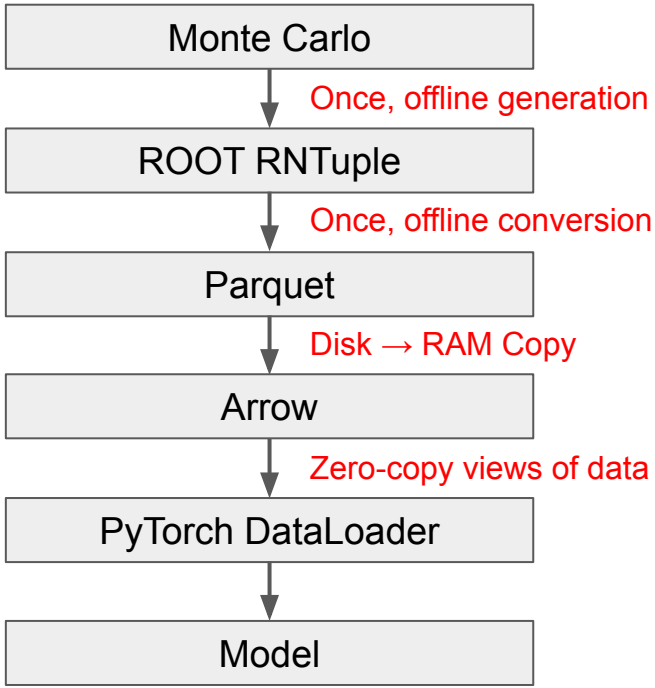
- Stream data in bounded batches
  - RAM usage should not scale with data set size
  - Allows RAM usage to be “tunable”
- Base data should be immutable
  - No modifications or extensions to ML pipeline input data files
- Separate derived data products
  - Predictions, masks, and regressions are produced as independent datasets
- Reference data by IDs, not by object
  - When passing data modules, use file paths or map IDs not in memory collections
  - Similar to why we use pointers in C++



Simple Example Pipeline Including Derived Datasets

Technologies for Scalability (Data)

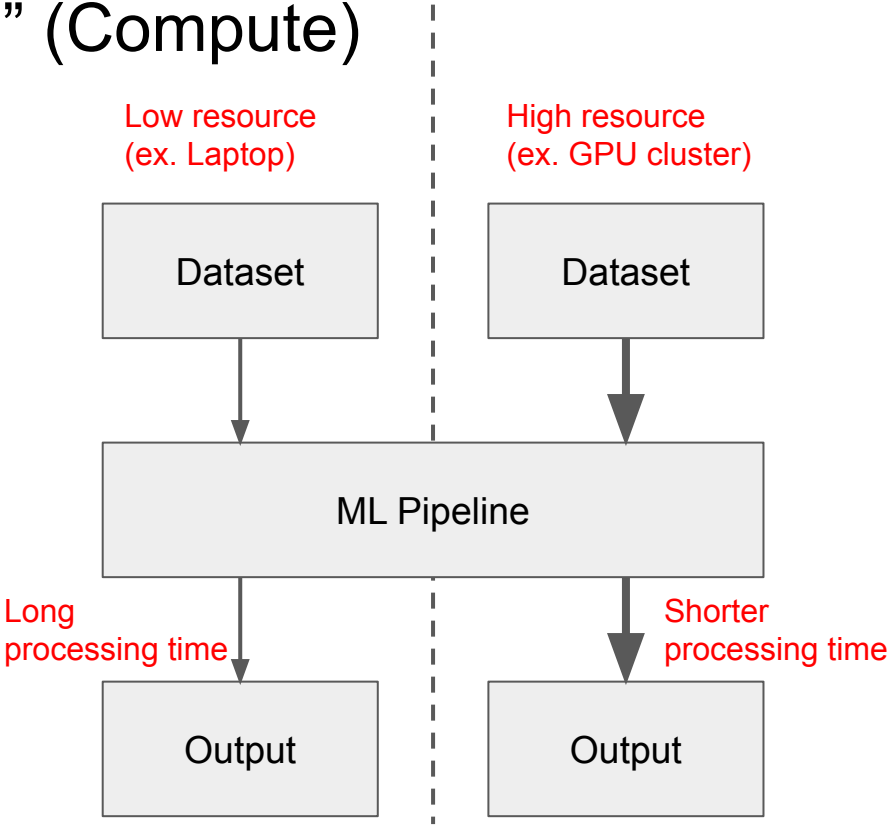
- ROOT RNTuple (or similar)
  - Holds the necessary data from the monte carlo
- Apache Parquet (columnar, on disk)
  - ML-native working datasets, append-only, shardable (aka “batchable”)
  - Can key rows by event ID, enabling batch-scoped joins for derived data products
- Apache Arrow
  - Single copy from disk into RAM, then zero-copy views all the way to Torch
- PyTorch tensors
  - Execution format for models
- Pytorch DataLoader
  - Handles batching, shuffling, parallel loading, prefetching, enforcing memory bounds



Flow of Data From Monte Carlo to a Model in the ML pipeline

What Do We Mean by “Scaling” (Compute)

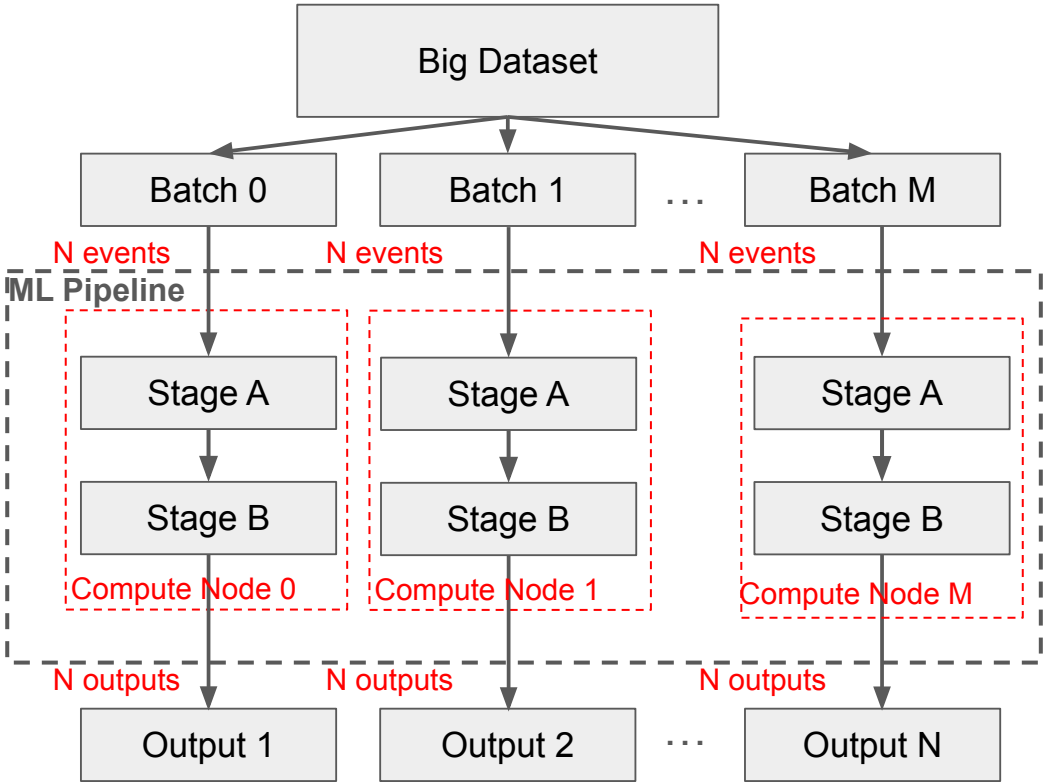
- **Computation resources should be able to grow without changing how the pipeline behaves**
- Adding the following should not cause pipeline behavior changes:
  - Number of GPUs
  - Number of CPU cores/threads
  - Node count
  - Memory Capacity
- Implications:
  - Algorithms must be agnostic to resources
    - Caveat: PyTorch and other frameworks may change their behavior for different devices/device counts for efficiency



Ideal Behavior for Different Amounts of Computing Resources

Techniques to Ensure Scalability (Compute)

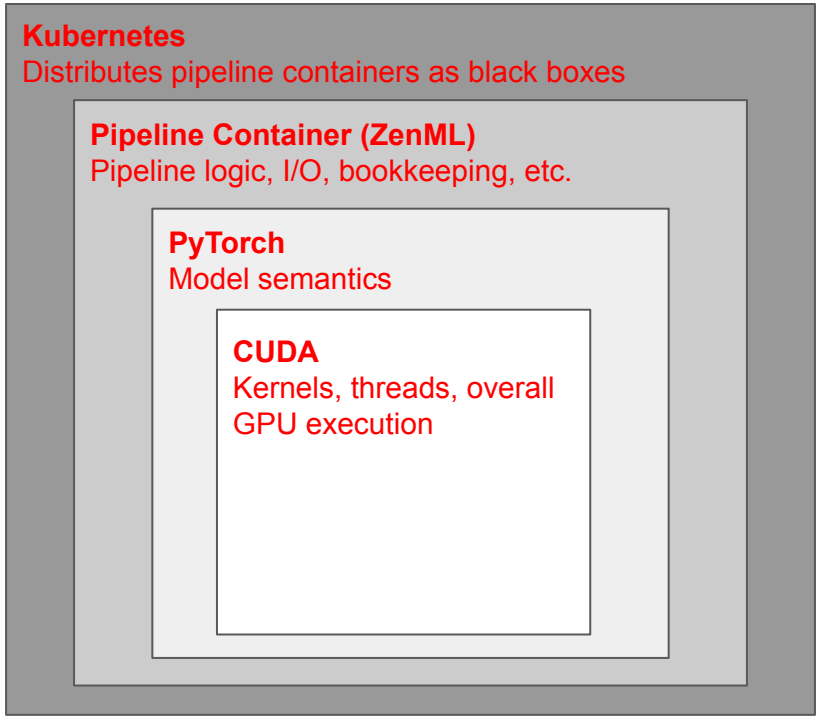
- Make algorithms resource-agnostic
  - No logic branches based on GPU count, core count, node count, or memory size, etc.
- Make algorithms easily parallelizable
  - Decompose computation into independent, composable units
  - Avoid global state dependencies in pipeline stages
  - Avoid “synchronization points”; i.e. points where models must make inferences on whole datasets



Parallelized View of Simplified Pipeline

Technologies for Scalability (Compute)

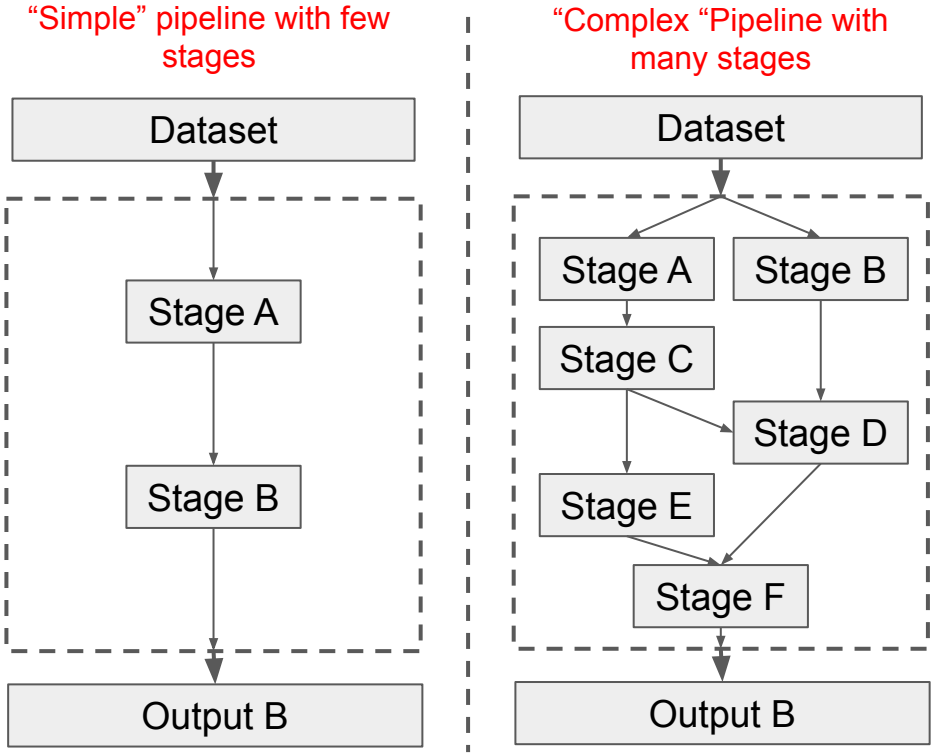
- PyTorch
  - Standard framework for modern ML models
  - Resource-agnostic execution model
- CUDA
  - Operates purely at the level of memory, kernels, and execution, independent of algorithm or pipeline semantics
  - Provides an API for launching and coordinating large numbers of parallel threads on GPUs
- Kubernetes
  - Schedules identical pipeline executions onto available compute nodes
  - Scales how many pipelines run concurrently, not what they do
  - Handles retry logic and resource limits



Simplified “Scope” Of Technologies
Outer Technologies Manage Inner Technologies

What Do We Mean by “Scaling” (Codebase)

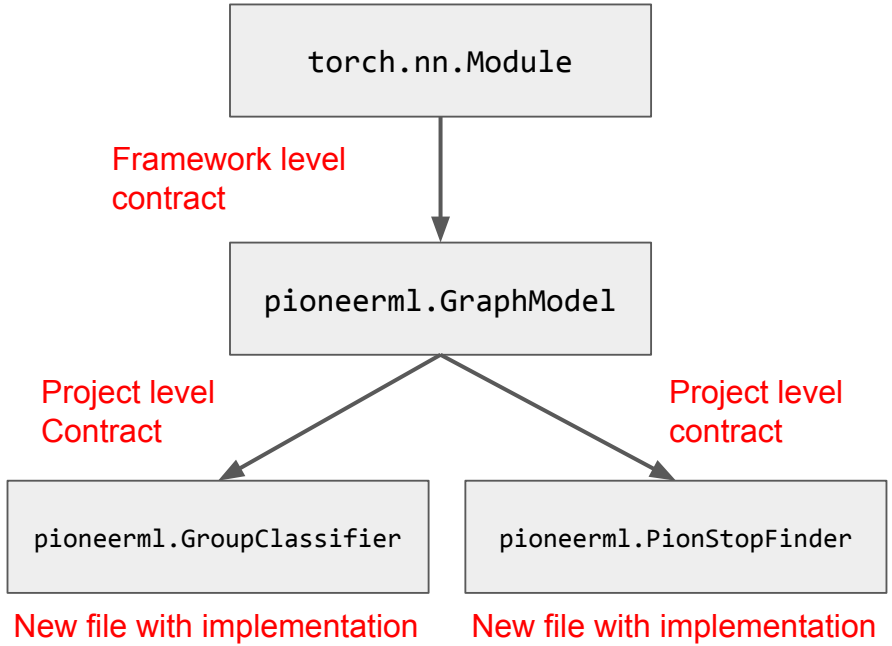
- **Codebase complexity should be able to grow without changing how the system behaves**
- Adding the following should not cause system wide behavioral changes:
  - More pipeline stages
  - More models / algorithms
  - More configuration options
- Implications:
  - New functionality should be added by extensions, not modification
  - System behavior should be locally understood (modularity)
  - Existing code should not require global refactoring to evolve



The “Global” Structure Should Not Change As Complexity Increases

Techniques to Ensure Scalability (Codebase)

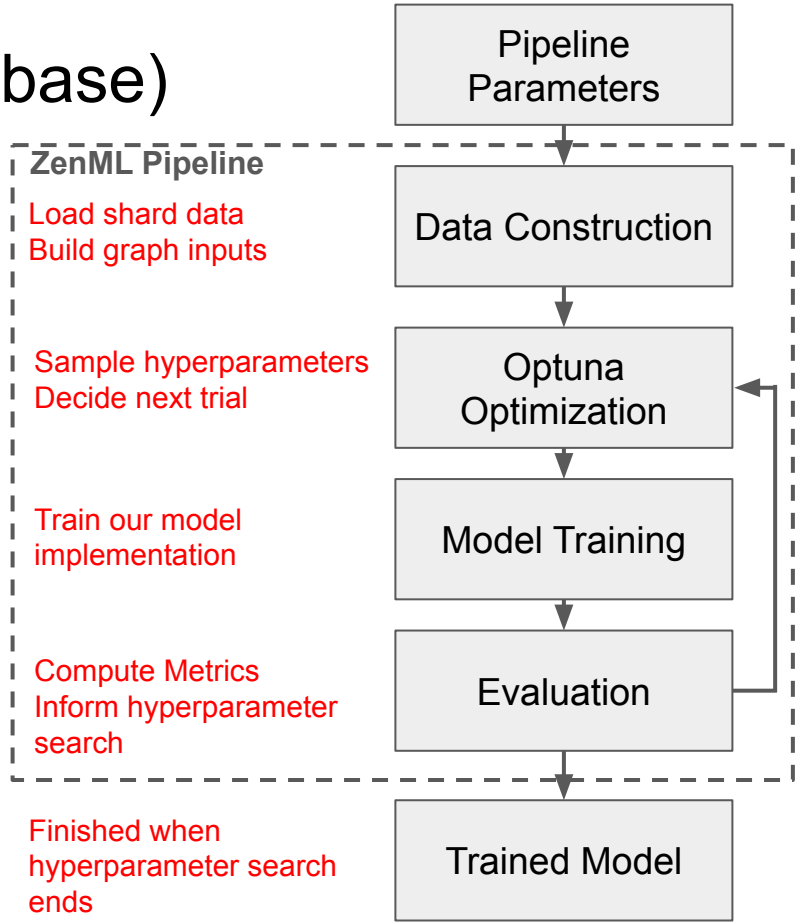
- Use abstraction where appropriate
  - Define stable base interfaces / abstract classes
  - Add new functionality by extending, not rewriting
- Avoid global coupling
  - Modules depend only on explicit inputs
  - Each module manages its own immutable local state
- Isolate responsibilities to achieve modularity
  - Each module has a single, well-defined responsibility
  - Changes remain local to the owning module



Simple Example of Abstraction For Adding New Models

Technologies for Scalability (Codebase)

- PyTorch
  - Provides a standard base abstraction classes for many model types (ex. `nn.Module`)
  - Enforces a consistent model interface (ex. `forward` method)
- ZenML
  - Encodes pipelines as composable, declarative units and orchestrates execution
  - Allows pipelines to grow by adding or reordering steps; easy to add new pipelines
  - Manages pipeline state, artifacts, and execution metadata outside user code
- Optuna
  - Isolates hyperparameter search code
  - Enables experimentation without modifying core implementations

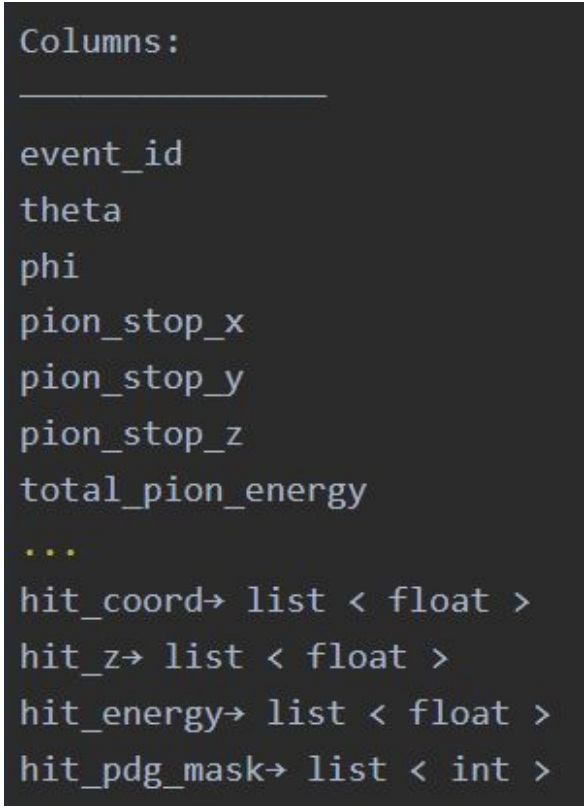


Simplified Example Pipeline for Training Models

Auxiliary Slides

What is Apache Parquet?

- **Apache Parquet is a columnar, on-disk data format that is widely used in ML workloads**
- What parquet does
  - Columnar storage → read only the columns (features) your model needs
    - Allows one to efficiently assign a subset of columns as inputs and another subset of columns as targets
  - Supports nested and variable-length fields
  - Data schema is embedded in the file, not inferred by code (ex. Not like numpy, where code defines `dtype` parameter)
- Why this matters for scalability
  - Efficient I/O for large datasets through compression and encoding
  - New models using different subsets of the data becomes trivial



Example time_groups.parquet file structure

What is Batching?

- **Batching means processing a fixed-size subset of events at a time, rather than the full dataset**
- What batching does
  - Groups individual events into batches of size N
  - Each batch is processed independently
  - Batches are discarded after use
- Why this matters for scalability
  - Memory usage is bounded by batch size (for single batch process)
  - Dataset size does not affect RAM usage
  - Enables streaming over arbitrarily large datasets

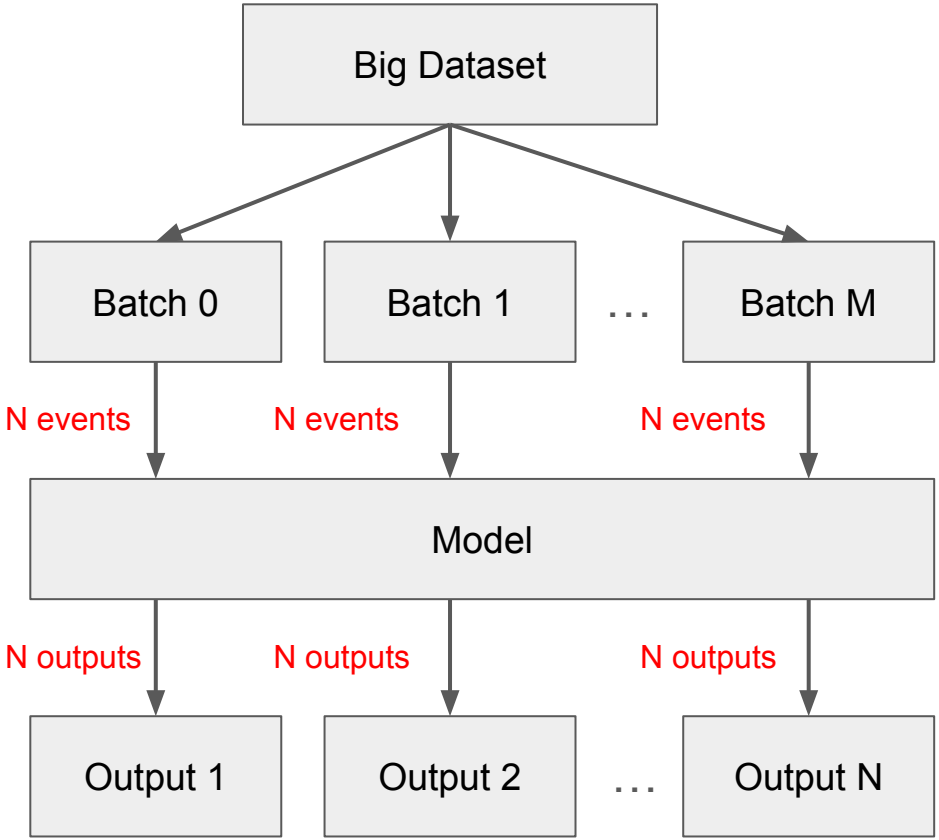
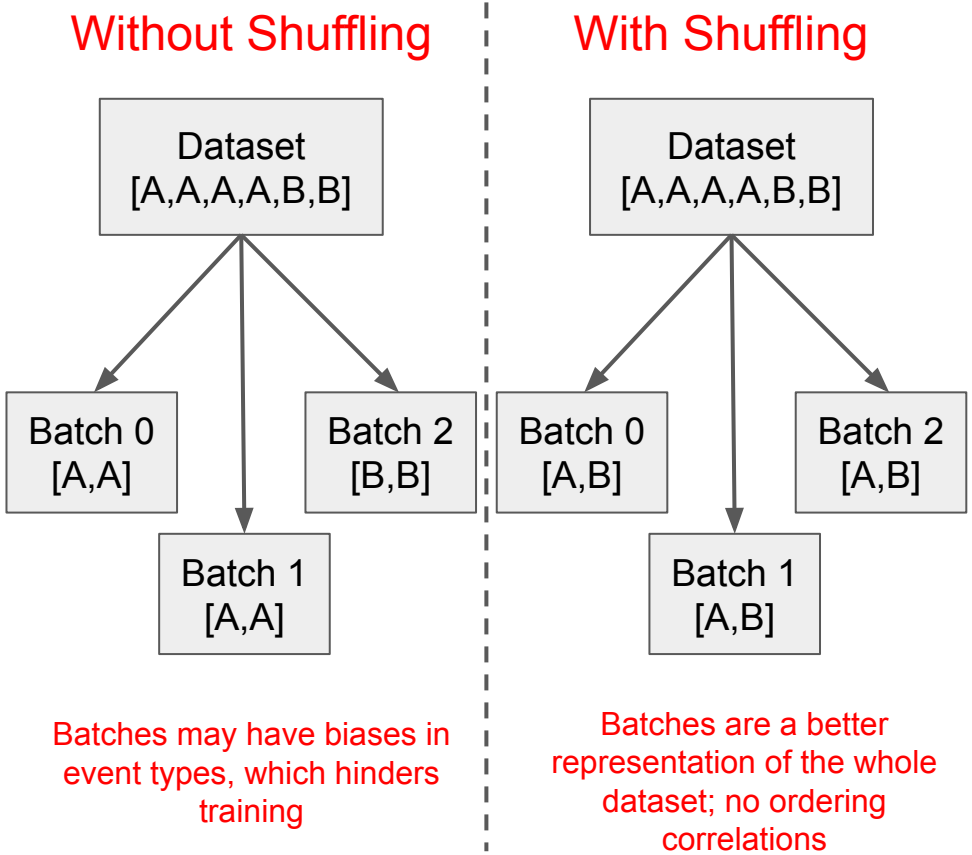


Diagram that Shows how Batching Splits Data

What is Shuffling?

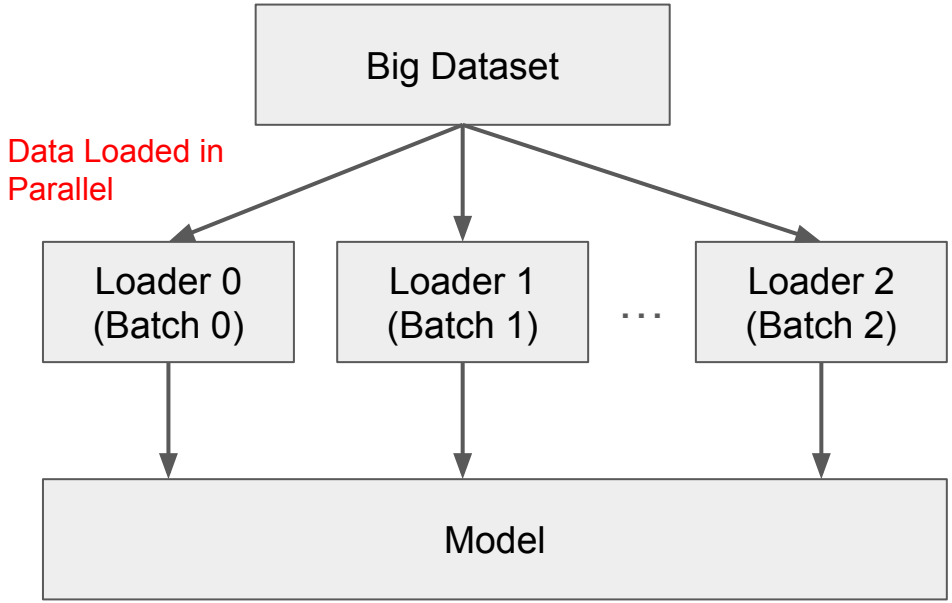
- **Shuffling changes the order in which events are seen, without changing the data itself**
- What shuffling does
  - Randomizes event order before forming batches
  - Ensures batches contain a mix of events
  - Changes between epochs (or passes over the data)
- Why this matters for scalability
  - Prevents bias from data ordering
  - Improves statistical independence between batches
  - Allows repeated passes over large datasets without correlation artifacts



Simplified Example of Shuffling vs. Not Shuffling a Dataset with Distinct Event Types A and B

What is Parallel Loading?

- **Parallel loading means loading multiple batches concurrently, using multiple workers**
- What parallel loading does
  - Multiple workers read and prepare batches simultaneously
  - The model always has a batch ready to process
  - Data loading is decoupled from model execution
- Why this matters for scalability
  - Helps prevent the model from waiting on disk I/O
  - Improves hardware utilization (especially GPUs)

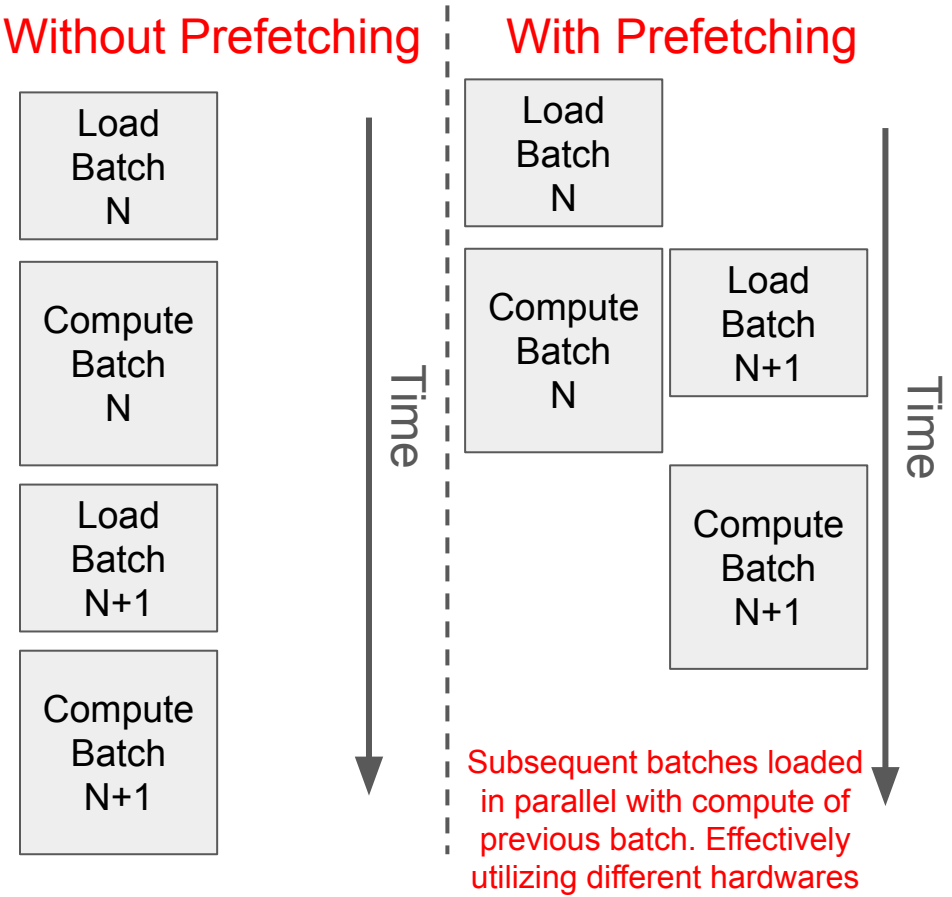


Note: Parallel loading does **not** necessarily mean parallel model execution

Parallel Data Loading Diagram

What is Prefetching?

- **Prefetching means loading future batches while the current batch is being processed**
- What prefetching does
  - While the model computes on batch N the next batch (N+1) is loaded in the background
  - When computation finishes, the next batch is already ready
- Why this matters for scalability
  - Helps prevent the model from waiting on disk I/O



Simplified Prefetching Example Diagram

What is Enforcing Memory Bounds?

- **Enforcing memory bounds means placing a hard limit on how much data can be in memory at once**
- What enforcing memory bounds does
  - Maximum in-flight data size is fixed
  - Batch creation is throttled when memory is full
  - Makes memory usage predictable and stable
- Why this matters for scalability
  - Dataset size does not affect RAM usage
  - Enables streaming over arbitrarily large datasets

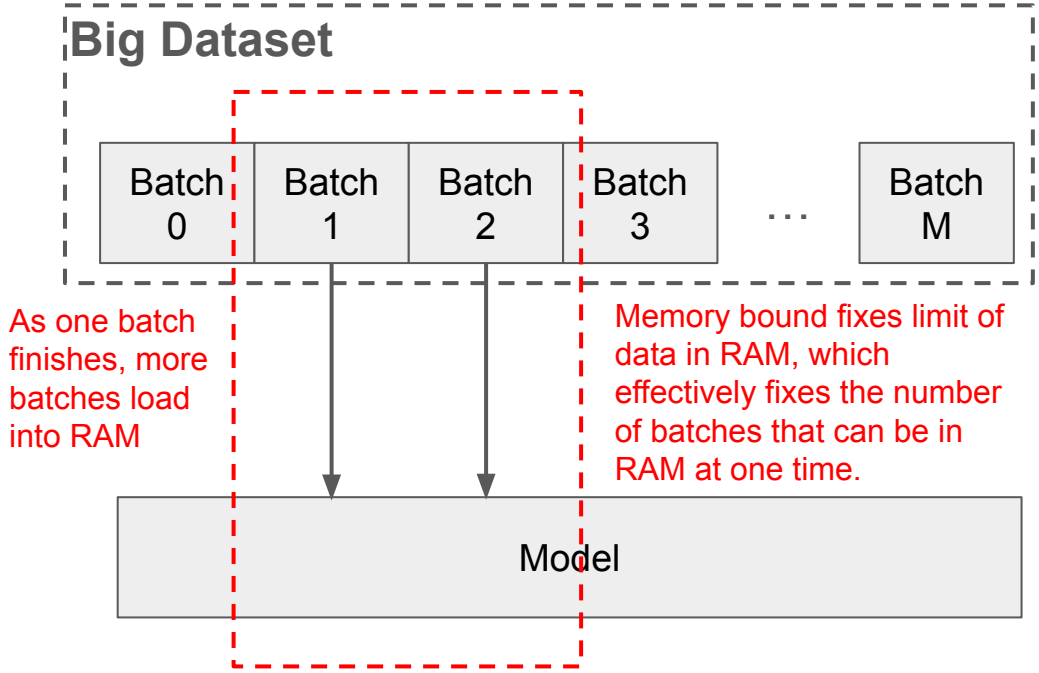
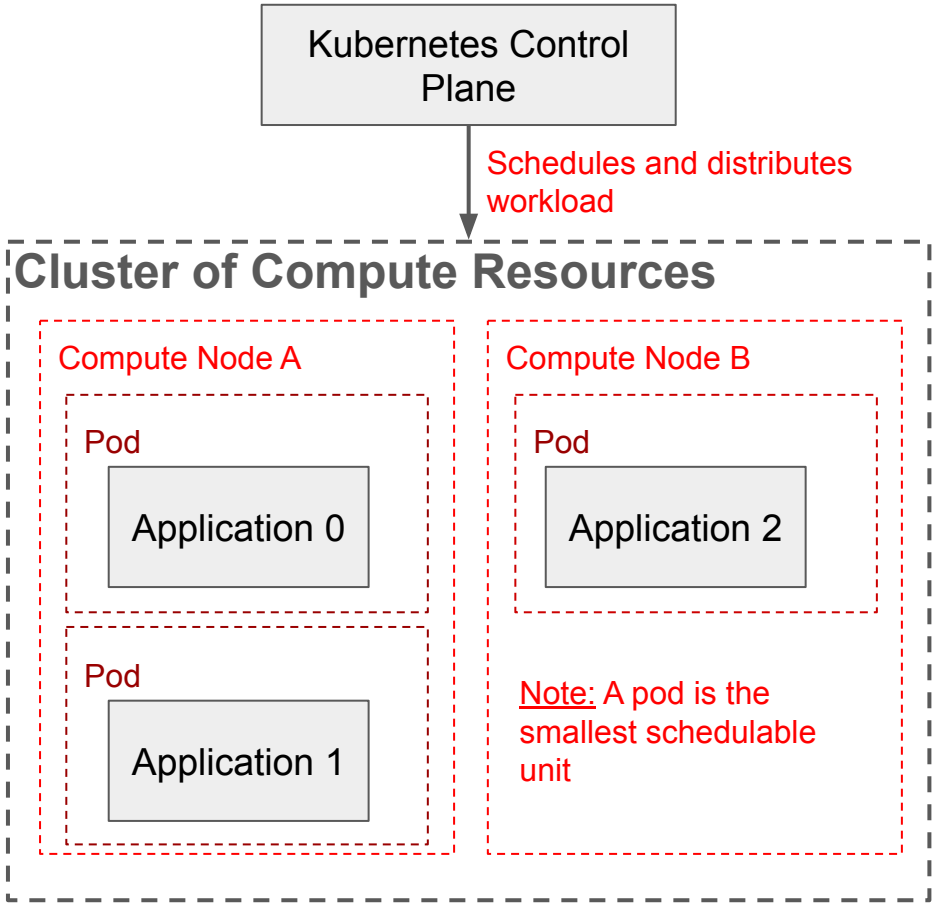


Diagram Showing How Memory Bounds Enforce a Limited Number of Batches Loaded in RAM

What is Kubernetes (K8s)?

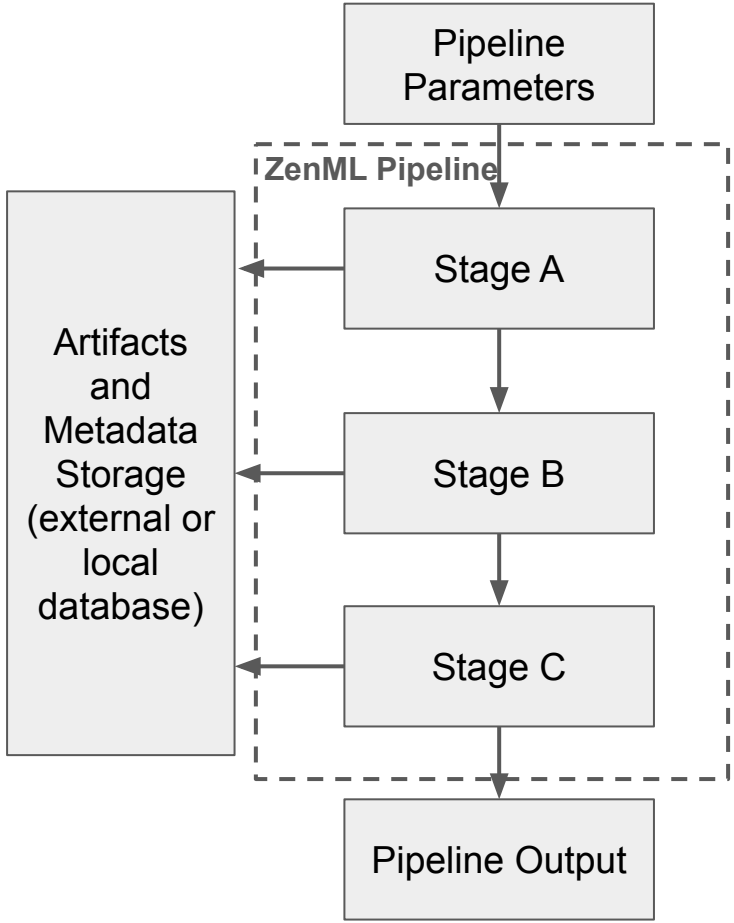
- **A system for running and managing containerized workloads across shared compute resources**
- What Kubernetes does
  - Schedules containers onto available compute nodes
  - Enforces resource limits (CPU, GPU, memory) per container
  - Isolates workloads from one another
  - Handles restarts and retries on failure
- Why this matters for scalability
  - Enables concurrent execution of many independent workloads
  - Prevents resource contention between workloads



Simplified Diagram of How Kubernetes Orchestrates Applications Across a Compute Cluster

What is ZenML?

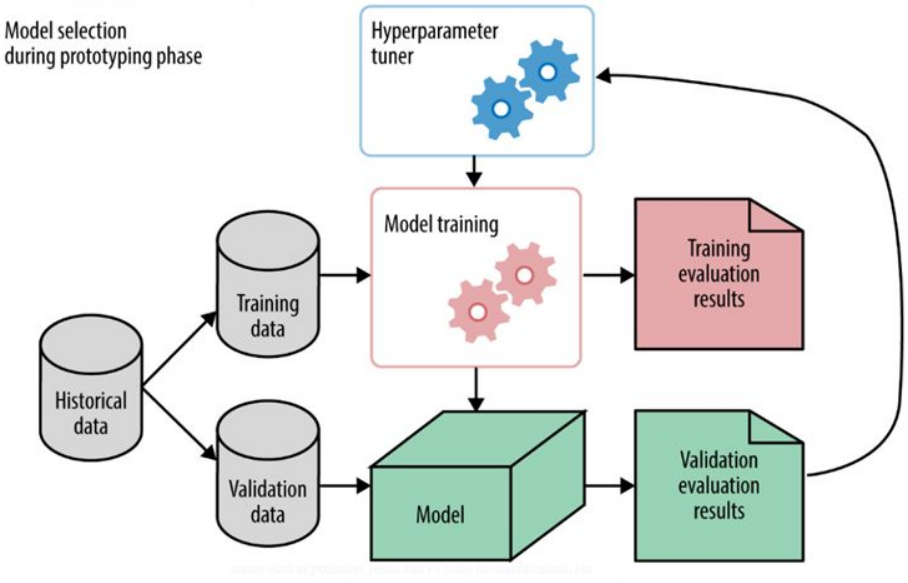
- **A framework for defining, orchestrating, and executing machine-learning pipelines with explicit steps, artifacts, and metadata**
- What ZenML does
  - Encodes workflows as composable, declarative pipelines
  - Orchestrates execution order and dependencies
  - Integrates with execution backends (local, containers, clusters) without changing user code
- Why this matters for scalability
  - Pipelines grow by adding or rearranging stages, not rewriting logic
  - Enables reproducibility, versioning, and parallel development
  - State and artifacts are managed outside user code



Example ZenML pipeline diagram

What is Optuna?

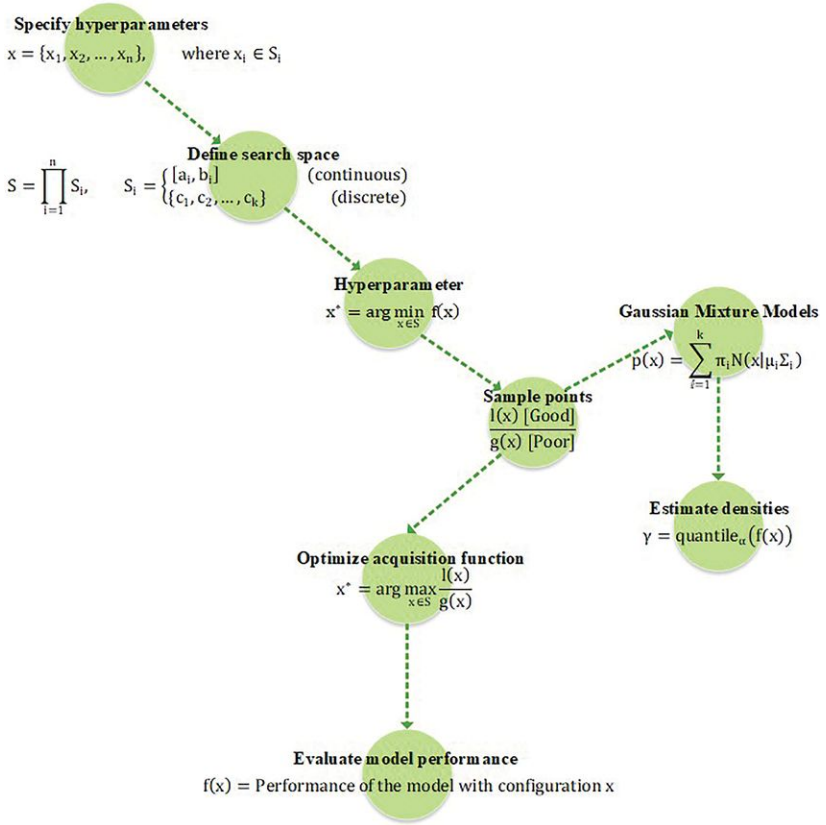
- Many hyper parameters means manual tuning is bad
  - Too slow
  - Suboptimal tuning means you spend more resources (time, computing power, etc.) training unused models
- Optuna is a python package that solves this problem
  - Framework for optimization black box objective functions
  - Technically not an ML package, but rather a package that supports many optimized sampling strategies



Example Optuna workflow diagram

How Optuna Works

- Optuna supports many sampling algorithms, examples:
  - Grid search
  - Random search
  - Gaussian process-based Bayesian optimization
- For single object functions, the default for Optuna is Tree-Structured Parzen Estimator (TPE)



TPE flow diagram (in a nutshell)

Note: See paper on arxiv

What is a Tree-Structured Parzen Estimator (TPE) (Part I)

- Given a (possibly stochastic) black box function you want to minimize (ex. model loss)

$$y = f(\theta)$$

- Hyperparams ≡ θ
- Define the following:

$$\gamma \equiv \text{"good"/"bad" fraction (say, 0.2)}$$
$$y^* \text{ is chosen such that } P(y < y^*) = \gamma$$
$$\text{so } y^* \text{ is in, say, the best } 20\% \text{ of losses}$$
$$\max(y^* - y, 0) \equiv \text{improvment}$$

- Then, for any given theta we can define *expected improvement*

$$\mathrm{EI}(\theta) \equiv \mathbb{E}[\max(y^* - y, 0) \mid \theta] = \int_{-\infty}^{y^*} (y^* - y)\, p(y \mid \theta)\, dy$$

- Goal: Maximize expected improvement

Note: See paper on arxiv

What is a Tree-Structured Parzen Estimator (TPE) (Part II)

- For a 1 dimensional y, it turns out to be easier to work with $p(\theta \mid y)$, we can invert using Bayes' rule:

$$p(y \mid \theta) = \frac{p(\theta \mid y)\, p(y)}{p(\theta)}$$

- And substitute

$$\mathrm{EI}(\theta) \propto \int_{-\infty}^{y^*} (y^* - y)\, \frac{p(\theta \mid y)}{p(\theta)}\, p(y)\, dy$$

- Since we don't know every value of y this becomes an impossible task. We must make an approximation by dividing into "good" and "bad" distributions

$$p(\theta \mid y) \approx \begin{cases} p(\theta \mid y < y^*) \equiv \ell(\theta), & y < y^* \text{ (good)} \\ p(\theta \mid y \geq y^*) \equiv g(\theta), & y \geq y^* \text{ (bad)} \end{cases}$$

$$p(\theta) \approx \ell(\theta) \int_{y<y^*} p(y)\, dy + g(\theta) \int_{y \geq y^*} p(y)\, dy = \gamma l(\theta) + (1-\gamma) g(\theta)$$

Note: See paper on arxiv

What is a Tree-Structured Parzen Estimator (TPE) (Part III)

- The integral, by construction, only cares about the “good” region, so the integral simplifies to

$$\mathrm{EI}(\theta) \propto \frac{\ell(\theta)}{p(\theta)} \int_{-\infty}^{y^*} (y^* - y)\, p(y)\, dy$$

- But the integral is now constant in theta! So we have:

$$\mathrm{EI}(\theta) \propto \frac{\ell(\theta)}{p(\theta)}$$

- Where $p(\theta) = \gamma\, \ell(\theta) + (1 - \gamma)\, g(\theta)$ so we can write:

$$\mathrm{EI}(\theta) \propto \frac{\ell(\theta)}{\gamma\, \ell(\theta) + (1 - \gamma)\, g(\theta)}$$

Note: See paper on arxiv

What is a Tree-Structured Parzen Estimator (TPE) (Part IV)

- But γ is fixed, so

$$\begin{aligned}\arg\max_{\theta} \mathrm{EI}(\theta) &= \arg\max_{\theta} \frac{\ell(\theta)}{\gamma\,\ell(\theta) + (1-\gamma)\,g(\theta)} \\ &= \arg\max_{\theta} \frac{\ell(\theta)/g(\theta)}{\gamma\,\ell(\theta)/g(\theta) + (1-\gamma)} \\ &= \arg\max_{\theta} \frac{\ell(\theta)}{g(\theta)}\end{aligned}$$

- Where the final step is because x/(bx+c) is monotonic in x for x >0 ,let x = l/g

- In other words, we just need to find $\arg\max_{\theta} \frac{\ell(\theta)}{g(\theta)}$ which is doable via algorithm!

Note: See paper on arxiv

What is a Tree-Structured Parzen Estimator (TPE) (Part V)

- Now to define the algorithm, first we observe T (~10) trials randomly:

$$\textbf{Observed trials:} \quad \mathcal{D} = \{(\theta^{(i)}, y^{(i)})\}_{i=1}^{T}$$
$$\textbf{Quantile threshold:} \quad P(y < y^*) = \gamma$$
$$\mathcal{D}_{\text{good}} = \{\theta^{(i)} : y^{(i)} < y^*\}$$
$$\mathcal{D}_{\text{bad}} = \{\theta^{(i)} : y^{(i)} \geq y^*\}$$

- From this data, we want to build:

$$\ell(\theta) \equiv p(\theta \mid y < y^*)$$
$$g(\theta) \equiv p(\theta \mid y \geq y^*)$$

- So we define

$$\theta \equiv (\theta_1, \theta_2, \ldots, \theta_d)$$

- And assume:

$$\textbf{Independence assumption:} \quad p(\theta \mid y < y^*) \approx \prod_{k=1}^{d} p(\theta_k \mid y < y^*)$$

Note: See paper on arxiv

What is a Tree-Structured Parzen Estimator (TPE) (Part VI)

- This allows us to write:

$$\ell(\theta) \approx \prod_{k=1}^{d} \hat{p}_{\text{good}}(\theta_k)$$

$$g(\theta) \approx \prod_{k=1}^{d} \hat{p}_{\text{bad}}(\theta_k)$$

- We can fit to our samples to get a continuous distribution spaces for each param

$$\hat{p}_{\text{good}}(\theta_k) \leftarrow \text{fit to } \{\theta_k^{(i)} : \theta^{(i)} \in \mathcal{D}_{\text{good}}\}$$

$$\hat{p}_{\text{bad}}(\theta_k) \leftarrow \text{fit to } \{\theta_k^{(i)} : \theta^{(i)} \in \mathcal{D}_{\text{bad}}\}$$

- Then we sample from the “good” spaces for M candidates index by m

$$\tilde{\theta}_{m,k} \sim \hat{p}_{\text{good}}(\theta_k), \qquad k = 1, \ldots, d$$

- And finally, choose our next theta, add it to the data set, and repeat

$$\theta^{(t)} = \arg\max_{\tilde{\theta}_m} \frac{\ell(\tilde{\theta}_m)}{g(\tilde{\theta}_m)} = \prod_{k=1}^{d} \frac{\hat{p}_{\text{good}}(\tilde{\theta}_{m,k})}{\hat{p}_{\text{bad}}(\tilde{\theta}_{m,k})}$$

Note: See paper on arxiv

What is a Tree-Structured Parzen Estimator (TPE) (Part VII)

- How do we obtain our fits from the data?
- Uses Kernel Density Estimation (KDE)
  - The actual fit is done when determining each “gaussian kernel” (or sigma)
  - Likelihood-optimal is expensive ~O(n^2d)
    - n = # samples
    - d = # hyperparameters
  - Optuna uses the “heuristic” version, which requires choosing some value for c.

Given samples: $\{\theta_k^{(i)}\}_{i=1}^n, \qquad \theta_k^{(i)} \in \mathbb{R}$

Kernel density estimate:

$$\hat{p}(\theta_k) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{K}\left(\frac{\theta_k - \theta_k^{(i)}}{h_k}\right)$$

Gaussian kernel:

$$\mathcal{K}(u) = \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{u^2}{2}\right)$$

Bandwidth (likelihood-optimal):

$$h_k^\star = \arg\max_h \sum_{i=1}^{n} \log \hat{p}_{-i}(\theta_k^{(i)} \mid h)$$

Bandwidth (heuristic used in practice):

$$h_k = c \cdot \mathrm{std}\left(\{\theta_k^{(i)}\}_{i=1}^n\right), \qquad c > 0$$