Modyn: Data-Centric Machine Learning Pipeline Orchestration

MAXIMILIAN BÖTHER, ETH Zurich, Switzerland
TIES ROBROEK, IT University of Copenhagen, Denmark
VIKTOR GSTEIGER, ETH Zurich, Switzerland
ROBIN HOLZINGER, Technical University of Munich, Germany
XIANZHE MA, ETH Zurich, Switzerland
PINAR TÖZÜN, IT University of Copenhagen, Denmark
ANA KLIMOVIC, ETH Zurich, Switzerland

In real-world machine learning (ML) pipelines, datasets are continuously growing. Models must incorporate this new training data to improve generalization and adapt to potential distribution shifts. The cost of model retraining is proportional to how frequently the model is retrained and how much data it is trained on, which makes the naive approach of retraining from scratch each time impractical.

We present Modyn, a data-centric end-to-end machine learning platform. Modyn's ML pipeline abstraction enables users to declaratively describe policies for continuously training a model on a growing dataset. Modyn pipelines allow users to apply data selection policies (to reduce the number of data points) and triggering policies (to reduce the number of trainings). Modyn executes and orchestrates these continuous ML training pipelines. The system is open-source and comes with an ecosystem of benchmark datasets, models, and tooling. We formally discuss how to measure the performance of ML pipelines by introducing the concept of composite models, enabling fair comparison of pipelines with different data selection and triggering policies. We empirically analyze how various data selection and triggering policies impact model accuracy, and also show that Modyn enables high throughput training with sample-level data selection.

CCS Concepts: • **Computing methodologies** → **Online learning settings**; • **Information systems** → **Data management systems**; Computing platforms; Spatial-temporal systems.

Additional Key Words and Phrases: Machine Learning Pipelines, Online Learning, Data-Centric AI

ACM Reference Format:
Maximilian Böther, Ties Robroek, Viktor Gsteiger, Robin Holzinger, Xianzhe Ma, Pınar Tözün, and Ana Klimovic. 2025. Modyn: Data-Centric Machine Learning Pipeline Orchestration. In *Proceedings of International Conference on Management of Data (SIGMOD '25).* ACM, New York, NY, USA, 30 pages. https://doi.org/XXXXXXX.XXXXXXX

Authors' Contact Information: Maximilian Böther, mboether@ethz.ch, ETH Zurich, Switzerland; Ties Robroek, titr@itu.dk, IT University of Copenhagen, Denmark; Viktor Gsteiger, vgsteiger@student.ethz.ch, ETH Zurich, Switzerland; Robin Holzinger, robin.holzinger@tum.de, Technical University of Munich, Germany; Xianzhe Ma, xianzhema@student.ethz.ch, ETH Zurich, Switzerland; Pınar Tözün, pito@itu.dk, IT University of Copenhagen, Denmark; Ana Klimovic, aklimovic@ethz.ch, ETH Zurich, Switzerland.

SIGMOD '25, June 22–27, 2025, Berlin, Germany

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

Fig. 1. Mean accuracies of 9 selection strategies (50 % subset) and full data training (see Section 7.1.1).

1 INTRODUCTION

The datasets fueling today's production machine learning (ML) models, which typically come from a myriad of sensors or real-time user click streams, are continuously growing [12, 55, 106]. To maintain high accuracy, stale models deployed in the wild need to be retrained in order to incorporate new data, particularly as training data may experience distribution drifts [39, 43, 44, 53, 56, 61, 85, 92, 95, 101, 110]. In practice, models may be retrained as often as every day [28], while the volume of data that models train on can be as high as petabytes or even exabytes, depending on the application domain [31, 124].

The cost of continuously training an ML model depends on *how frequently* we retrain the model and *how much data* we use to train the model each time. The naive approach of retraining a model from scratch on the entire dataset when new data becomes available is prohibitively expensive and slow [46, 60]. To make retraining[1] models practical in real use cases, we need to minimize the frequency and the volume of data that a model is trained on, while maintaining high model quality. For example, Figure 1 shows how various data selection policies (x-axis) proposed in ML literature maintain model accuracy comparable to training on all data (dashed line) while training on only 50 % of the YEARBOOK image classification dataset [121]. Complementary to data selection, data drift detection can help to trigger retraining only when data characteristics change. This can save cost and/or increase model quality compared to fixed-interval retraining schedules.

However, finding the right data selection and triggering policies is non-trivial. While ML researchers have explored how to effectively select important samples in a dataset with various strategies [3, 4, 45, 47, 49, 62, 63, 73, 76, 79, 81, 86], it is not clear what policy to use for real-world datasets that grow and exhibit distribution shifts over time. ML studies in this space often focus on smaller, static datasets, such as CIFAR [51] and MNIST [54], and do not consider the total pipeline cost, or they focus on one particular metric (e.g., information retention in continual learning studies [16, 80]). While several drift detection techniques exist [34, 59, 85, 90, 105], existing studies focus on tabular data, synthetically inject drift, and do not train neural networks in response to drift [2, 8, 88, 89, 94, 113]. Using such techniques as triggering policies is non-trivial as it involves tuning many hyperparameters. Most pipelines today are still human-driven [42, 95].

Furthermore, it is challenging to implement data selection policies in large-scale growing dataset environments while maintaining high training throughput. Data ingestion is a common bottleneck in ML training [52, 67, 124]. Applying data selection policies requires accessing individual data samples rather than sequentially reading input data files. Such random access patterns can degrade training throughput. In Section 7.3, we show that multiple levels of batching, parallelizing, and prefetching of reads are essential to achieve high throughput. Such optimizations should be done

[1] In this paper, retraining refers to both finetuning or training from scratch.

transparently by a platform, while ML users focus on defining the logic of ML training and data preprocessing pipelines. While others have also acknowledged the need for a continuous training platform that enables users to explore data selection and (re)training policies [26, 75, 96, 108], current open-source systems only have limited support for retraining [10, 25, 65, 107]. We are not aware of any platform supporting sample-level data selection policies.

We present Modyn, a data-centric machine learning pipeline orchestrator that addresses this gap. To the best of our knowledge, in particular for modalities commonly used in DNN training such as images, Modyn is the first open-source orchestrator to support data selection and retraining decisions based on the incoming data. Modyn is an end-to-end platform that supports the entire pipeline lifecycle, including sampling-level data selection and management, triggering model retraining, continuously evaluating model quality, and managing model snapshots. In this paper, we contribute the following:

(1) We design an ML pipeline abstraction, which enables users to express how to continuously train a model on growing data. It allows declaratively specifying data-centric policies for model retraining and training data selection, while decoupling the implementation of these policies. We design the abstraction to capture a taxonomy of data selection and triggering policies.
(2) We build Modyn, an orchestrator that runs data-centric ML pipelines. Modyn supports various data selection techniques while optimizing for high-throughput sample-level data selection for multiple data formats. It also supports time-, data volume-, performance-, and data drift-based triggering policies while managing and continuously evaluating model versions. Modyn enables sample-level data selection with comparable throughput to sequentially ingesting data from local storage.
(3) We formalize ML pipelines and introduce *composite models* which describe the performance of a pipeline over its lifetime, and allow for a fair comparison of pipelines with different selection and triggering policies. We build an ecosystem around Modyn to facilitate policy exploration. Modyn comes with web-based tooling to compare pipelines in terms of system throughput, training cost, and model quality metrics. It also comes with a set of benchmark models and datasets with timestamped data for policy evaluation. For a subset of the accompanying benchmarks, we include case studies on selection and triggering policies, showing how these policies impact pipeline performance.

2 BACKGROUND AND MOTIVATION

In this section, we discuss the growing nature of real ML datasets (Section 2.1) and motivate the need for a new platform (Section 2.2).

2.1 Growing Datasets & ML Perspective

Real-world ML datasets are often dynamic, in contrast to static datasets such as ImageNet [24] that are typically used in ML research [14]. They either grow as more samples are collected (e.g., from continuous data sources like sensors or click streams) or shrink as data is deleted (e.g., due to privacy reasons). In this work, we focus on the challenges of training ML models on *growing data*.

Why growing data matters. Incoming data captures current trends and reveals *distribution shifts* that can be critical in many application domains [97, 110], like recommender systems [28, 37, 39, 120, 124] and language models [53]. For example, the GrubHub food delivery platform observed a 20 % increase in purchase rate when their model is retrained daily rather than weekly [28]. Even in the absence of significant distribution shifts, including additional data over time can enhance model performance as it improves generalization. For example, Tesla continuously collects street pictures to refine their autonomous driving models [106]. Growing data impacts training cost, as

the cost is proportional to *(i)* how often the model trains and *(ii)* the number of data samples it trains on [46, 60].

ML perspective on growing data. ML research so far has explored optimizing when to retrain a model and what data to select for training as two isolated dimensions. The field of *continual learning* (CL) [3, 4, 7, 26, 50, 58, 81], or incremental learning [18, 78, 117], adapts ML models to ongoing data streams by focusing on learning new tasks, defined as groups of classes. It is unclear how these techniques apply to real use cases, as CL research has focused on small datasets with synthetic perturbations that lack a true notion of time. Furthermore, both the focus on learning classes over time instead of adapting to distribution shift and the common assumption of limited storage are not realistic, as acknowledged by recent works in the CL community [32, 80]. Data selection policies outside of CL focus on selecting subsets of static datasets [47, 62, 63]. All techniques require sample-level data access on the dataset.

While there is work on detecting distribution shift [55, 85, 105], these papers often focus on theoretical aspects and do not actually train models [2, 8, 34, 57, 90], i.e., they only compare drift scores. Notably, Werner et al. [113] train random forests on tabular datasets, and Yuan et al. [122] consider synthetically perturbed variations of the MNIST dataset from continual learning. To the best of our knowledge, no paper explored applying drift detection techniques to training large neural networks on modalities such as images and text from non-synthetic benchmarks.

2.2 Platform Support

Managing when to retrain models and on what data to train models in large-scale growing data environments is challenging. It requires efficiently orchestrating continuous training pipelines with configurable triggers and fast access to arbitrary sets of data samples determined by a data selection policy. Model training orchestration and sample-level data fetching should be transparently optimized by a platform in order to help ML researchers focus on policy exploration and to help ML practitioners reliably deploy ML pipelines in production environments. Furthermore, drift detection techniques need to be closely embedded into the data flow, since they typically need to access the previously trained models and the data stream.

Current ML platforms do not address these requirements. The majority of ML training platforms, such as Weights & Biases [13] or MLFlow [20], are tailored more towards experiment tracking than continuous retraining. While a few (often commercial) platforms like NeptuneAI [70], Amazon SageMaker [5], Continuum [107], or Tensorflow TFX [65] have partial retraining support, deploying continuous retraining still requires a lot of manual plumbing [10, 75, 95, 108]. Commonly, platforms allow for the performance of the deployed model to trigger a retraining (e.g., SageMaker or tf-serving [74]). Notably, Hopsworks [41] supports drift detection on individual features of tabular data, and SageMaker's Model Monitor allows for the collection of drift metrics on tabular data using the Deequ library [93]. Images and other modalities are explicitly not supported currently. Especially for modalities commonly used in DNN training (images and text), data-centric decision making on when and what data to train on is, to the best of our knowledge, not supported by any available open-source training platform. Platforms such as Ekya [12], which optimizes retraining for vision models on edge devices, and Ekko [98], which optimizes model updates for recommendation systems, cater to specific use cases. The aforementioned platforms view the datasets as a big blob of data instead of indexing individual samples.

3 MODELING DYNAMIC ML PIPELINES

Continuous ML pipelines regularly run model trainings on an incoming stream of data S with a discrete time clock. The data arrives in batches $S_i \subset S$, i.e., sequences of new samples, where batch t

is given as $S_t = (s_1, \ldots, s_{n_t})$. Each sample $s_i \in S_t$ comprises a unique identifier, a label, a timestamp, other metadata, and a data payload.

Triggering. The triggering policy decides whether to trigger a new training[2]. We model a triggering policy as a function $\pi : \mathcal{P}(S) \to \bigcup_{n=0}^{\infty} \mathcal{P}([1, \ldots, n])$. Given a batch S_t, it determines which samples $s_i \in S_t$ trigger a new training process. Formally, it outputs a sequence $\pi(S_t) = (i \in [1 \ldots n_t] \mid s_i \in S_t \text{ causes trigger})$. The triggering policy can be stateful and utilize the observed history of samples, properties of them or the pipeline, to come to a triggering decision. Conceptually, the triggering policy decides on a per-sample basis. For efficiency, our implementation evaluates multiple samples simultaneously in batches, while keeping the semantics of per-sample decision-making. Note that triggering on each new data sample is impractical in production, as each newly trained model typically needs to undergo a set of extensive deployment checks, which are expensive to run at high frequency [43, 95].

Data selection. On each trigger, the selection policy chooses which samples to train on. Let $s_k \in S_t$ cause the overall r-th retraining trigger. The observed data until trigger r is $\mathcal{D}_r^{\text{tot}} = \{s_i \in S_t \mid i \leq k\} \cup \bigcup_{t'<t} \text{set}(S_{t'})$. A data selection policy is a function $\xi_r : \mathcal{D}_r^{\text{tot}} \to \mathbb{R}^{|\mathcal{D}_r^{\text{tot}}|}$ that assigns each item in the total observed data a weight. Thereby, the function defines the r-th *trigger training set* $\mathcal{D}_r \subset \mathcal{D}_r^{\text{tot}} \times \mathbb{R}^+$. An item is included if its weight is greater than 0. The data selection policy selects from all previously seen data samples, i.e., they can come from any $S_{t'}$ with $t' < t$, and all samples in S_t until s_k. The sample weights can be used to prioritize samples by multiplying their gradients with the weights during backpropagation. The trigger training set is a *subset* of all data points seen so far, so it may, but does not have to, contain samples from previous triggers.

3.1 Evaluating and comparing pipelines

To compare ML pipelines, we first need to define how to quantify the performance and cost of a pipeline. Evaluating the model quality and training cost of an ML pipeline on a growing dataset is more complex than evaluating a single model training on a static dataset. Two challenges arise.

First, ML pipelines train multiple models instead of a single one to deal with the growing data that might exhibit distribution shift. On each retraining trigger r we train a new model m_r. For a pipeline P, let $\mathcal{M}_P$ denote the sequence of all models trained during pipeline execution. Each $m_r \in \mathcal{M}_P$ is a 4-tuple containing its model weights w_r, the data it was trained on $\mathcal{D}_r$, and the start timestamp t_r^s and end timestamp t_r^e of the training data, i.e., $m_r = \langle w_r, \mathcal{D}_r, t_r^s, t_r^e \rangle$. We should not just evaluate a single model from $\mathcal{M}_P$, e.g., the last one, on the entire dataset since the model is trained on one particular distribution. Instead, we need to consider multiple models.

Second, to understand how a model's performance changes over time, we need to define windows over the evaluation data, as discussed by Shankar et al. [96]. Evaluation data should be separate from training data, e.g., by partitioning the stream S. These windows are temporal slices of the dataset on which we then calculate a quality metric per model. Let P_1 and P_2 be pipelines with different triggering policies π_1 and π_2 on the same stream of data. $\mathcal{M}_{P_1}$ and $\mathcal{M}_{P_2}$ contain different models, in particular with different timestamps. The intuitive solution of defining evaluation windows matching the training intervals of the models, i.e., each model m_r defines an evaluation window from t_r^s to t_r^e, is not fair across pipelines. Hence, we first need to decouple determining evaluation intervals from triggering and then define which model to use for which window.

We define an evaluation interval as a 3-tuple $\langle \tau^s, \tau^a, \tau^e \rangle$. τ^s and τ^e define the start and end of the range from which we consider evaluation data. τ^a defines the *anchor point* of the interval, which serves as a reference timestamp that we use in further definitions. Typically, $\tau^a = \tau^s$ or $\tau^a = (\tau^s + \tau^e)/2$. We define an *interval generation function* φ as a procedure that generates intervals

[2]We use the terms training and retraining interchangeably.

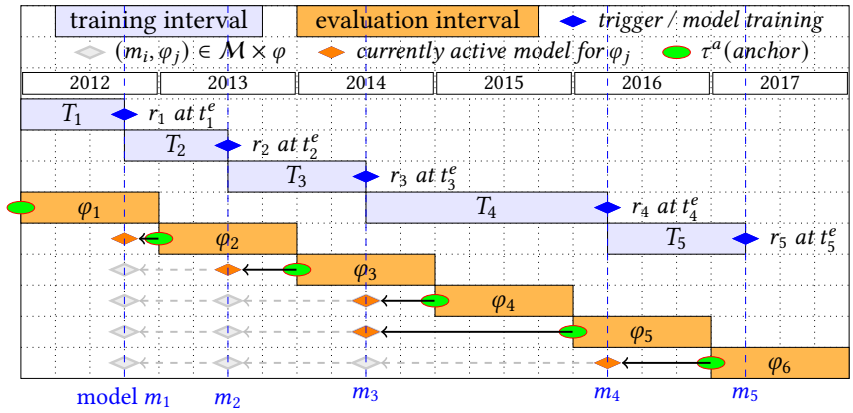

Fig. 2. Visualization of finding the currently active model.

on the evaluation dataset, i.e., it outputs a sequence of evaluation intervals $\left(\left\langle \tau_1^s, \tau_1^a, \tau_1^e \right\rangle, \ldots\right)$. We also use φ to denote this sequence. The generated intervals can, e.g., be fixed-length sliding- or tumbling windows. For any metric σ (e.g., accuracy) and a model $m_x \in \mathcal{M}_P$, let $\sigma\left(m_x, \varphi_i\right)$ denote the score of model m_x on data with timestamps in the i-th evaluation interval. We define the evaluation matrix $m_{\sigma,P} \in \mathbb{R}^{|\mathcal{M}_P| \times |\varphi|}$ where, for all $i \leq |\mathcal{M}_P|$ and $j \leq |\varphi|$, $m_{\sigma,P}\left[i, j\right] = \sigma\left(m_i, \varphi_j\right)$. Each model is evaluated on each window.

From matrices to sequence. Currently, each pipeline is associated with a 2-dimensional evaluation matrix (models and intervals). When comparing multiple pipelines, we have to consider another dimension for the pipelines themselves. To reduce the number of dimensions, we propose to define a *composite model* per pipeline. Formally, the composite model is a partial mapping $\mu_P : \varphi \rightarrow \mathcal{M}_p$. This allows us to condense the accuracy matrix $m_{\sigma,P} \in \mathbb{R}^{|\mathcal{M}_P| \times |\varphi|}$ into a sequence of evaluation results $\Lambda_{\sigma,P} = \left(m_{\sigma,P}\left[\mu_P\left(i\right), i\right] \mid i \leq |\varphi|\right) \in \mathbb{R}^{|\varphi|}$. This sequence represents the temporal performance of a pipeline. We call it the composite model performance, though the composite model is formally a mapping.

We propose and focus on two variants of composite models. In the *currently active* composite model, every evaluation window uses the most recent model that has completed training prior to the anchor of the evaluation interval, i.e., $\mu_P^{active}(\varphi_i) = \arg\max_{m_x \in \mathcal{M}_P}\{t_x^e \mid m_x = \left\langle w_x, \mathcal{D}_x, t_x^s, t_x^e \right\rangle \wedge t_x^e \leq \tau_i^a\}$. Intervals whose anchor is before the first model, i.e., when no model training has finished before the evaluation data comes, do not have a currently active model. It is a modeling decision of the interval generation function whether the anchor point lies on the left boundary of the interval, or, e.g., in the center, to allow for a mix of out-of-distribution and in-distribution data. Figure 2 visualizes this with a Gantt chart of the model and evaluation intervals. In this example, we set $\tau^a = \tau^s$. The training data intervals end with a trigger, indicated by the blue diamond. Conceptually, each evaluation interval searches (arrows to the left) for the first model that has finished training before its anchor at the beginning of the box. The model associated with the trigger belonging to the dashed vertical line is marked as currently active for the evaluation interval, indicated by the orange diamond. A model can be active for several (r_3) or no intervals (r_5).

The *currently trained* composite model is the model following the currently active model. Let i be the index such that for the j-th interval $\mu_P^{active}(\varphi_j) = m_i$. We define $\mu_P^{train}(\varphi_j) = m_{\min(i+1, |\mathcal{M}_P|)}$. For the edge case when the currently active model is undefined, we set the most recent model as currently trained. The currently trained model potentially benefits from training on data distributions similar to those in the evaluation set. We will see an example of the difference between μ_P^{train} and μ_P^{active}

in Section 7. These definitions emphasize the current performance of a pipeline. They might not capture other aspects such as retention of previous knowledge.

Further dimensionality reduction. For a comparative analysis of pipelines, plotting the temporal accuracy of composite models, i.e., plotting $\Lambda_{\sigma,P}$, may provide visual insights. To distill this information into a single metric, the series $\Lambda_{\sigma,P} \in \mathbb{R}^{|\varphi|}$ (the composite model accuracy) can be averaged into a pipeline score $\Sigma_{\sigma,P} \in \mathbb{R}$ to obtain an indication of the general pipeline performance over time. This is how we calculated the mean in Figure 1 to compare pipeline performance. Furthermore, this scoring is useful for ranking pipelines in an AutoML setting [40, 89].

Cost trade-off and pipeline comparison. Let $\mathscr{P}$ be a set of pipelines, each assigned a fixed cost P_c. Costs can be measured by the number of triggers, the number of samples trained on, or wall clock run time. The number of triggers is only fair when all pipelines use the same selection policy on only the new data since the last trigger as in this case each sample from the entire dataset is trained on at most once. The number of samples is fair across different selection policies but disregards overheads such as the cost of the triggering and selection policies. Wall clock time covers everything, but requires pipelines to be run on isolated machines.

Having assigned a cost, we can build the cost-accuracy feasible set $\mathbf{F}_{\mathscr{P}} = \left\{ \left\langle \Sigma_{\sigma,P}, P_c \right\rangle \mid P \in \mathscr{P} \right\}$. There might be several pareto-optimal pipelines. For visually comparing pipelines, we can plot this feasible set and get an understanding of how different pipelines perform with respect to the tradeoff between training cost and predictive performance.

4 MODYN'S DESIGN

MODYN is designed to implement the pipeline model described in Section 3. Hence, the core unit of execution in MODYN is a pipeline. Users *declaratively specify* the pipeline which allows to decouple the pipeline policy from how it gets executed and lets users focus on model engineering. Still, MODYN allows users to add new models and policies as Python modules and offers abstractions to support this (Section 5).

MODYN is designed to fill the gap identified in Section 2.2. To allow users to control which individual data samples to access for training, MODYN's storage component assigns each sample a unique ID and associates metadata with each ID. Instead of seeing the dataset as a blob of data, MODYN offers a `get_sample_by_id` interface to fetch data according to the selection policy during training. Next, to support the rich landscape of selection and triggering policies in its declarative interface, MODYN introduces a taxonomy of these policies (Section 5) and implements abstractions to apply these techniques to common DNN data modalities like text or images. Furthermore, we design MODYN's rich evaluation infrastructure to support the ideas outlined in Section 3.

Figure 3 shows MODYN's components and the basic flow of pipeline execution. MODYN ingests data from a data source, such as stream processing engines (e.g., Flink [17]) or batch processing frameworks (e.g., Spark [123]). We assume that expensive preprocessing operations, e.g., filtering and downscaling a stream of images, happen offline, i.e., before ingestion into MODYN. Online ML preprocessing (e.g., image augmentation) happens within MODYN. While data preprocessing for ML provides challenges in itself [119], existing work is addressing those challenges [25, 33, 92]. MODYN is positioned between the preprocessing of the data and the serving of models. MODYN expects a *labeled* input data stream. Such labels can be either obtained automatically (e.g., track which advertisements a user clicked on) or from human-in-the-loop annotation systems [116]. MODYN outputs a stream of trained models that can then be deployed, using tools like TorchServe [84], BentoML [11], or Triton Inference Server [71].

Overview of control flow and data flow. The user submits a pipeline via MODYN's CLI to the supervisor ⓪, which implements the triggering policy and orchestrates the execution. MODYN

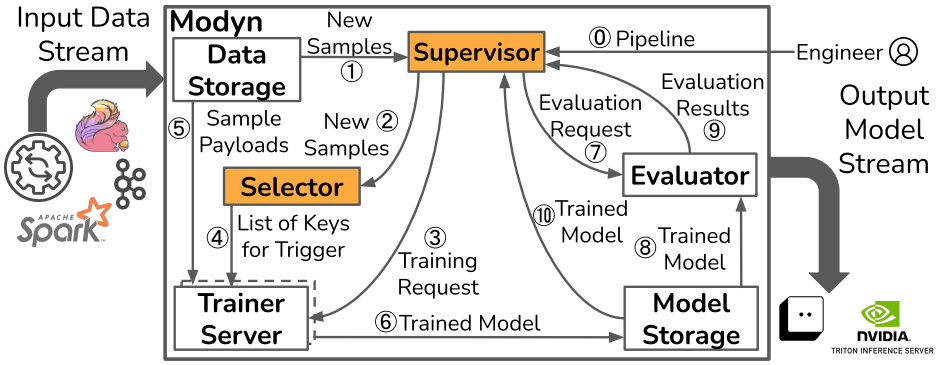

Fig. 3. Modyn's system design.

stores data samples streaming in from external sources in its storage, which assigns a unique key to each sample. The data storage component informs the supervisor about new samples by their key ①. The supervisor checks whether any data point in the incoming batch causes a trigger and forwards potential triggers and the sample keys to the selector ②, which implements the data selection policy. Upon trigger, the supervisor contacts the trainer server to start a training process ③. The trainer server requests the trigger training set (keys and weights to train on) from the selector ④. Then, it loads the actual data from the storage ⑤ and, depending on the configuration, also the previous model from the model storage. The trainer server then runs a training according to the configuration. The trained model is then stored in the model storage component ⑥. The supervisor can send an evaluation request to the evaluator ⑦, which receives the newly trained model from model storage ⑧, evaluates it and returns the results ⑨. The supervisor can also receive the new model for new triggering decisions ⑩. Finally, the model can be deployed.

Example pipeline. Figure 4 shows a declaratively-specified Modyn pipeline. At minimum, a description comprises *(1)* the model specification, *(2)* the training dataset and a corresponding bytes parser function that defines how to convert raw sample bytes to model input, *(3)* the triggering policy, *(4)* the data selection policy, *(5)* training hyperparameters such as the the learning rate and batch size, *(6)* training configuration such as data processing workers and number of GPUs, and *(7)* the model storage policy, i.e., a definition how the models are compressed and stored. A training may involve fine-tuning a model or training a model from scratch with randomly initialized weights; this is a configuration parameter in the triggering policy. The very first training can run on a randomly initialized or externally provided model.

5 IMPLEMENTATION

We describe the supervisor and triggering policies (Section 5.1), the selector and data selection policies (Section 5.2), data retrieval (Section 5.3), and the remaining components (Section 5.4). We build Modyn with the goal of providing an easy-to-use, extensible, and efficient execution platform for data-centric ML pipelines. We aim to build an ecosystem around Modyn to facilitate policy exploration in practical use cases of ML training in growing data environments.

To balance performance and ease-of-use, Modyn components are either written in C++ (e.g., storage service), purely in Python (e.g., trainer service), or Python with C++ extensions (e.g., selector service). While code on the hot path of data fetching is written in C++ to avoid stalls, the pluggable algorithm modules are written in Python. Having a clean Python interface allows ML researchers to implement policies in a familiar language without worrying about systems aspects. For compatibility, we use existing tooling like PyTorch where possible.

```yaml
model:
  id: ResNet18
  config:
    num_classes: 42
data:
  dataset_id: mnist
  transformations: ["transforms.Normalize(...)"]
  bytes_parser_function: |
    def bytes_parser_function(data: memoryview) -> Image:
      return Image.open(io.BytesIO(data)).convert("RGB")
trigger:
  id: DataAmountTrigger
  num_samples: 100
training:
  use_previous_model: True
  batch_size: 1234
  optimizers: ...
  optimization_criterion:
    name: "CrossEntropyLoss"
  selection_strategy:
    name: "CoresetStrategy"
    storage_backend: "database"
    tail_triggers: 0
    presampling_config: ...
    downsampling_config: ...
model_storage:
  full_model_strategy:
    name: "PyTorchFullModel"
  incremental_model_strategy:
    name: "WeightsDifference"
evaluation: ...
```

Fig. 4. Excerpt from an example MODYN pipeline.

MODYN uses gRPC and FTP for data and control flow, and supports Docker Compose for deployment. The codebase, totaling ca. 20 000 lines of Python and 2 500 lines of C++ (excluding tests), is publicly accessible[3], and undergoes rigorous unit and integration testing, as well as linting, establishing it as more than a research prototype.

To overcome the limitations imposed by the Global Interpreter Lock (GIL) in Python, our implementation employs a hybrid processing and threading approach. It utilizes a gRPC ThreadPool and `multiprocessing.Processes`, leveraging the `SO_REUSEPORT` socket option. This combination enables the system to handle multiple gRPC requests concurrently, achieving true parallelism despite the GIL constraints.

5.1 Supervisor

The supervisor orchestrates the execution of pipelines. Pipelines are submitted via MODYN's CLI. The CLI is the interface between supervisor and user. MODYN uses Pydantic models [83] to guide users in specifying their pipelines. For each submitted pipeline, the supervisor spawns a `PipelineExecutor`, which implements a state machine following the control flow outlined in Section 4. The client frequently polls the supervisor for the current status and displays the current pipeline stage and training progress.

Triggering policies. During execution, the supervisor decides to trigger using a triggering policy. MODYN currently supports amount-, time-, performance-, and drift-based triggering policies. Amount-based triggers fire every n data points, while time-based triggers fire after a time interval has passed. Performance-based triggers trigger when the accuracy degrades. They require labels which might arrive late in practice [97]. Drift triggers, however, work unsupervised and detect

[3]Available at https://github.com/eth-easl/modyn.

covariate shift, i.e., they compare the distribution of the incoming data to some reference data. We leverage the evidently [30] and alibi-detect [109] libraries for calculating similarity metrics and hypothesis testing.

Data drift variants. For unstructured data such as images, we transform it into a latent embedding space, and optionally project it to lower dimensionality, e.g., using PCA. Modyn uses the most recent model of the pipeline to generate embeddings. Modyn builds up a sliding window of current data and reference data (current data window at the last trigger). In a defined interval, a similarity metric such as MMD between the two windows is obtained. Based on the similarity metric, we need to make a binary decision about whether there is drift between the reference and current data, i.e., whether we trigger. Modyn supports threshold-based decisions, i.e., we trigger when the metric is higher than a threshold. As this threshold needs to be tuned for each dataset and metric, Modyn supports dynamic decision making (`AutoDrift`). It keeps track of a window of previously observed drift scores and triggers when a new drift score is in a configurable percentile of these scores as a simple outlier detection.

Similarity metrics. While tabular data as used in previous work [88, 94, 113] can be used directly, for images and text, Modyn generates embeddings as dense latent representations, and calculates drift metrics on those embeddings. The embedding dimensions become features as in the tabular data domain. We find that some distance metrics, such as the Kolmogorov–Smirnov or Hellinger distance [27], are commonly used for univariate distributions. In univariate drift detection, we derive one distance metric *per feature*, e.g., for 512-dimensional embeddings, we obtain 512 distance values that need to be reduced into a scalar. Multivariate extensions or natively multivariate metrics provide a scalar distance value, even for multivariate distributions. We focus on the multivariate MMD metric since we did not find readily available multivariate implementations of other metrics. Additionally, it has not been explored how to best reduce multiple univariate metrics into a scalar value, how to decide whether the data has drifted, and MMD performed best in initial experiments.

Open questions. Using drift detection on unstructured data such as images is an active area of research. First, the impact of the embedding space, i.e., which model is used to generate embeddings, has not been explored. Second, it has not been studied what is a sensible interval to run detection, what metric to choose in which scenarios, and how big the windows should be. Last, it is not clear what is the best way to make the binary triggering decision, and it likely depends on the metric, dataset, embeddings, etc. Note that our goal is to demonstrate how Modyn enables the use and exploration of different triggering policies rather than advocating for a particular policy. We are actively exploring these questions and discuss the first results in Section 7.2.

Execution modes. Modyn advocates the principle of *what you evaluate is what you deploy*. Managing separate codebases for research and production is error-prone. Hence, any pipeline can be executed in either *experiment mode* or *production mode*. In production mode, the data storage informs the supervisor when new data points arrive. In experiment mode, the data storage simulates new data points streaming in by announcing existing data points as "new" to the supervisor. The experiment mode can be used to (re)play traces and compare how policies perform given the same environment. The insights gained from these experiments can then be used to find a configuration for production.

5.2 Selector

The selector implements data selection policies, which generate the trigger training set $\mathcal{D}_r$ upon the r-th trigger per pipeline.

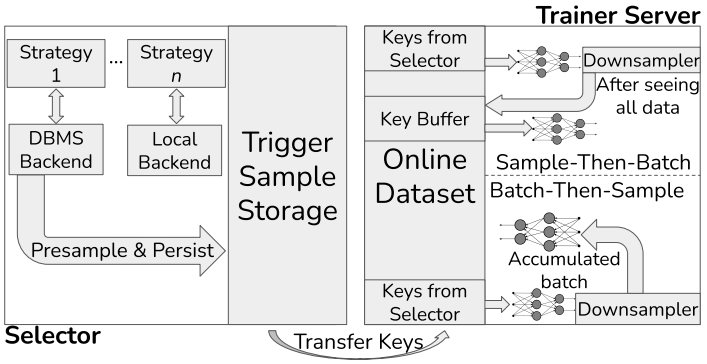

Fig. 5. Data selection flow in Modyn.

5.2.1 Selection policies. A selection policy defines what data to train a model on upon trigger. Every selection policy has a window upon the past data, i.e., a pool of data we could train on. This window can be infinite ("retrain" on all past data), just include the data since the last trigger ("finetune" on new data), or include all data up to n previous triggers. In order to either reduce the amount of data that we train on or increase information retention, we can then apply selection algorithms on this window of data. In the following, we discuss a taxonomy of selection policies.

Presampling and downsampling. We identify two types of selection algorithms. Presampling algorithms do not require any information from the model forward pass and are implemented at the selector. Examples include ingesting older samples to increase information retention, sampling in a class-balanced fashion, or use-case specific sampling (e.g., increasing the weight of pictures at night for autonomous driving pipelines).

Downsamplers are general-purpose techniques which leverage information from the model forward pass to pick the best samples to use for the backward pass [22, 47, 63, 79]. Downsampling happens at the trainer server. For example, the DLIS policy [47] samples data points based on the gradient norm obtained during the forward pass. Any downsampler can be combined with an offline or online presampling policy.

Offline/online presampling. Any presampling policy is offline or online. Offline policies maintain state by storing all samples during a trigger and running the actual selection on trigger. For example, a strategy sampling class-balanced from the data window requires storing all data first and only samples on trigger after determining the available classes. Online policies perform the sampling directly as data is received. Examples for online policies include continual learning algorithms such as GDumb [81], CLIB [50], and GSS [4].

Supported policies. Currently, for presampling, Modyn supports class-balanced sampling (similar to GDumb [81]), sampling uniformly at random, and trigger-balanced sampling. For downsampling, Modyn supports RS2 [73], loss sampling [47], DLIS [47], uncertainty downsampling [22], CRAIG [63], and GradMatch [48]. It also implements a warmup period of not using sampling for the first triggers to improve upon the initial model more quickly.

5.2.2 Implementation of policies. Presampling and downsampling policies are implemented as Python classes, each category sharing its own common interface. Modyn provides infrastructure, e.g., for storing state, to help engineers and researchers port algorithms. The overall flow of data selection is shown in Figure 5, which we detail in the following paragraphs. When informed about new samples, the selection policy updates its state using a *metadata backend* module provided by the selector. This state is used to calculate the set $\mathcal{D}_r$ on trigger r. This set is then stored on disk using an extension called `TriggerSampleStorage` (Section 5.3).

Backends for presampling. For implementing presampling strategies, Modyn provides two backends that share an interface to store the state of the sampling strategy. The first backend is the Postgres backend, which persists the samples to a Postgres table [102]. The advantage of this backend is the flexibility for implementing selection policies, since many policies can be expressed using SQL statements. We use SQLAlchemy [9] to allow for easy querying of data. Modyn provides query boilerplates using inheritance hierarchies, e.g., in order to implement a random sampling balanced across some parameter such as trigger or label, the developer inherits from the `AbstractBalancedStrategy` and specifies the column to balance on. The disadvantage of the Postgres backend is the slow insertion speed. Every sample has to be written into the database. We optimize the ingestion with Postgres' table partitioning mechanism. We partition the state table first by pipeline, then by trigger, and then round-robin with a modulus of 16. This avoids the degrading of insertion performance with growing number of triggers, since every trigger defines a new physical table. In order to further optimize the insertion speed, we use SQL bulk insertion and run several insertion threads for new batches of incoming keys.

For datasets with many samples, e.g., recommendation system datasets, using Postgres can be very expensive. We observe maximum insertion speeds of around 100 000 insertions/second. For simple strategies not requiring complex SQL queries (e.g., train on all the data since the last trigger), or if performance is key, Modyn offers a local backend. This is a C++ extension that writes data multithreadedly to a local disk, such as a high performance NVMe drive. These binary files are written and read avoiding unnecessary memory copies. Strategies such as training on all data, uniform presampling, or mixing old and new data can be implemented easily on this backend, trading off ease of implementation for speed. Each workload has different requirements and Modyn provides building blocks for these use cases.

Implementing downsamplers. Downsampling policies cannot be executed at the selector and need support from the trainer server. Modyn's training loop has a component which executes the downsampling policy specified in the pipeline. As shown in Figure 5, the presampled trigger training set is transferred to the trainer server, where it is then downsampled.

Analogous to offline versus online presampling, downsamplers can be run in either *sample-then-batch* (StB) or *batch-then-sample* (BtS) mode. Some downsamplers like RHO-LOSS [62] explicitly are proposed with BtS or StB mode, and others like DLIS [47] can be used in both modes. In BtS, the training loop runs inference on a batch and then selects a subset of that batch. This is repeated until we accumulate a new batch of the original batch size, on which we then perform a backward pass. In StB mode, the training loop starts with a sampling phase in which it continuously informs the downsampler about the forward pass, allowing the downsampler to build up state for all samples. Once this state is complete, it generates the downsampled dataset, and we run training on these keys. This sampling phase can be performed every training epoch or less often. Both StB and BtS mode are abstracted such that engineers just have to implement one version of the downsampling policy. While there can be multiple epochs of training per trigger, Modyn applies the budget constraint per epoch. For example, when we have 1 000 samples, a 10 % budget, and 10 epochs per trigger, we train for 10 epochs of 100 samples each, instead of a single epoch of 1 000 samples. Maintaining the epoch boundary allows, e.g., consistent setups for learning rate scheduling.

5.3 Fast Data Retrieval

Modyn supports different data selection policies, which means that the trigger training set is an arbitrary collection of previously stored samples. Regardless of the selection policy, the result is a list of training items, i.e., IDs of samples, to train on. This is a shift in architecture from traditional ML deployments, where the training data is typically a big chunk of data that can be read sequentially. Instead, Modyn supports *sample-level data selection*, i.e., retrieving samples based on their identifier.

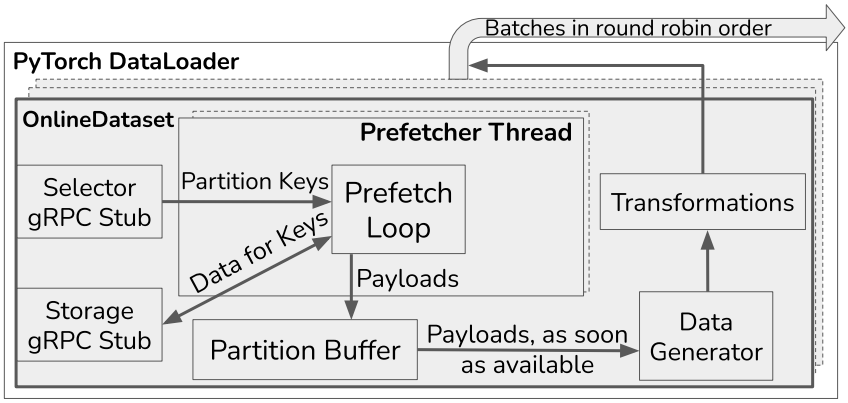

Fig. 6. The architecture of the `OnlineDataset`.

For big datasets with potentially billions of small samples like in recommendation systems, this can lead to data stalls during training. In this subsection, we describe how we engineer Modyn to avoid data stalls while supporting sample-level data selection. We first describe the storage component (Section 5.3.1). Then we describe the `OnlineDataset` abstraction that loads keys from the selector, payloads from storage, parses the bytes, and returns tensors into the training loop (Section 5.3.2). We furthermore explain how the selector quickly returns the list of keys (Section 5.3.3), and how, given a list of keys, the storage quickly returns the requested data (Section 5.3.4).

5.3.1 Data Storage. The storage is entirely written in C++ as we found the data wrangling to be particularly expensive in Python. A Postgres database is used to keep track of all available samples. Each ingested file can contain one or more samples, e.g., a JPEG file contains exactly one sample, while a CSV file contains potentially hundreds of thousands of samples. When the component encounters a new file, it extracts all the samples in that file and inserts the file, the sample IDs, and the labels into the database. The storage makes use of `FileSystemWrappers` which abstract I/O operations such as reading byte streams from files. Currently, Modyn implements a file system wrapper for the local file system, but this can be easily extended to support cloud file systems like S3. The storage then uses `FileWrappers` which abstract how to extract individual samples and labels from files. Examples include the `CSVFileWrapper` for variable-length CSV data, the `BinaryFileWrapper` for fixed-size columnar data, often used in recommendation systems training, and the `SingleSampleFileWrapper` for files containing exactly one sample, such as images.

The C++ implementation uses SOCI [99] to operate on the Postgres database. To optimize the ingestion and query performance, we partition the tables. Since for datasets with billions of samples, even SQL bulk insertion is too slow, we use the Postgres internal `COPY` command and stream the data over the raw connection.

5.3.2 The `OnlineDataset`. The `OnlineDataset` abstracts away the interaction with the different gRPC components from the training loop. The training loop (Section 5.4) uses a standard PyTorch DataLoader to fetch batches to train on. It is not aware of the ongoing network communication. This new abstraction is necessary due to Modyn's sample-level data selection. We cannot just load a big chunk of data and train on it. Instead, we have to load the data according to the list of keys in the trigger training set. The PyTorch DataLoader uses multiple workers. We split the trigger training set across these workers. The trigger training set consists of fixed-size partitions (Section 5.3.3). Each worker gets an equal share of each partition. The data loader fetches batches from the workers in a round-robin fashion.

In order to avoid data stalls when the data loader requests data, each worker (or dataset instance) implements a prefetching mechanism. This architecture is depicted in Figure 6. Each worker has a partition buffer of a configurable size. Upon creation, a worker spawns a configurable number of prefetching threads that issue gRPC requests. The size of the buffer defines how many partitions we prefetch overall, while the number of threads defines how many partitions we prefetch in parallel. To fetch a partition, we first obtain a list of keys from the selector, and then ask the storage for the payloads corresponding to these keys. The storage uses gRPC streaming to transfer the payloads to the workers.

As soon as data is available in the buffer, the main thread of the worker fetches the payload, applies transformations, and yields it to the data loader. This is important, since waiting for a partition to finish transferring would make the batch latency depend on partition size. The only exception is when if shuffling is enabled, i.e., we need to shuffle the samples in each partition and the order of partitions, as we need to alter the sample order. The first transformation always is a user-defined *bytes parser function* defining how to transform the bytes of the payload to a tensor, e.g., by decoding the bytes of a JPEG image or decoding a UTF-8 string. Afterwards, other transformations are applied as defined by the pipeline, such as image augmentations or tokenization.

5.3.3 Data partitioning. We need to retrieve the trigger training set, i.e., the keys to train on, as fast as possible. Instead of relying on a database, we persist the fixed trigger training set after presampling to disk using the `TriggerSampleStorage` (TSS). The TSS is a fast C++ extension that persists the list of keys and weights (c.f. Section 3) output by the presampling strategy to disk. The TSS uses the same binary file format as the local backend.

Writing to disk. The selection strategy does not pass all keys and weights at once to the TSS. Instead, it passes the keys as multiple partitions. Each partition is a fixed-size set of keys. For example, if the trigger training set consists of 1 000 keys and the partition size is 100, the strategy will pass 10 partitions to the TSS. This avoids high memory utilization by limiting the amount of keys loaded at once. Furthermore, the partitions provide a fixed-size unit of data transfer for the trainer server. The backends provide support for partitioning, i.e., limiting the memory usage. For the Postgres backend, we use Postgres' server-side cursors. For the local backend, we read the corresponding data via offsets. When the TSS writes the final partition to disk, n threads (within the C++ extension) write the keys and weights of the partition to disk in parallel.

Retrieving keys. When retrieving partition data for a worker, we iterate over all files for this partition. The requesting worker ID and the number of total samples correspond to a list of samples for this worker. However, as the number of dataloader workers does not necessarily match the number of threads we used to persist the training set to disk, we have to potentially parse subparts of files and correctly and efficiently assemble each worker's share of a partition. This is hidden in the C++ extension and only the final list of keys is returned.

5.3.4 Storage data retrieval. What makes the storage challenging is that it can receive requests with arbitrary sets of sample keys. When samples are requested, they are distributed across a set of files, and each may be residing at arbitrary locations within those files. The storage needs to efficiently build a buffer of data that makes it look as if the data came from one continuous file that contained all requested samples.

When a worker sends a list of keys to the storage for retrieval, the storage partitions this list into $n \geq 1$ parts to parallelize the retrieval from disk. Then, each thread obtains labels and a source file for each sample from Postgres, grouped by file. For each file, it instantiates a `FileWrapper` and extracts all samples in that file into a send buffer. When that buffer is full, or once all files have been iterated through, the thread emits the buffer to the worker. Besides parallelization, major speed gains for each thread stem from optimized `FileWrapper` implementations. For example, the

`BinaryFileWrapper` has an optimized bytes-to-int parsing function based on the endianness the file was written with, and operates on `std::ifstreams` to not load the entire file into memory.

5.4 Other Components

5.4.1 Trainer Server. The trainer server spins up trainers when requested, which execute a general-purpose training loop. Modyn currently implements a PyTorch-based trainer, but its design is agnostic to the ML framework. The trainer supports a variety of features like mixed-precision training or learning rate schedulers with correct support for data selection [73]. Modyn comes with some models (e.g., ResNets [38], DLRM [69, 72], and transformers [115]) and other models can be added easily. The trainer also performs online featurization, such as image augmentation.

5.4.2 Model Storage. This component is responsible for model storage and retrieval. It supports full model and incremental compression policies. The full model policy defines how to compress the entire model such that it can be restored from just the file itself, analogous to an I-frame in video encoding. Furthermore, the model storage can employ an incremental policy, which activates a configurable number of times between full model steps. In this mode, Modyn stores just the delta from the base model based on a specified difference operator. This is similar to a P-frame in video encoding. For full model policies, the model storage currently supports both the native PyTorch format and a custom, stripped binary storage format, with or without zip compression. For incremental policies, it currently supports an xor and subtraction based difference operator. Model compression over time is an active area of research [36, 103].

5.4.3 Evaluator. Each model trained during a pipeline can be evaluated on several evaluation intervals for multiple evaluation metrics. Modyn's evaluator implements various interval generation functions φ, e.g., tumbling- or sliding windows. It also supports both decomposable (e.g., accuracy) and holistic metrics (e.g., ROC-AUC).

6 BENCHMARK SUITE

A major hurdle for research on growing datasets is the scarcity of publicly accessible datasets that encapsulate temporal dynamics and distribution shifts. Modyn incorporates a benchmark suite that curates datasets, pipeline configurations, and models to run pipelines with. It comes with the necessary tooling for making them available on the user's machine as some datasets involve post-processing and metadata scraping. The suite includes:

(1) The Wild-Time benchmarking suite [121]: A compilation of five datasets, ranging from small to medium in size, each exhibiting distribution shifts.
(2) Kaggle arXiv and HuffPost datasets: The arXiv and HuffPost datasets from Wild-Time only have coarse-grained timestamps on a year resolution and have been filtered by unclear criteria. Modyn provides tooling to generate full, high resolution versions using the source data from Kaggle [6, 64].
(3) The Criteo 1 TB dataset [23]: The Criteo click stream dataset for recommendation systems training provides user data over 24 days, with roughly 180 million samples per day.
(4) The CGLM dataset(s) [80]: The paper on CGLM classifies images from landmarks on Wikipedia and uses the upload timestamps. Since the original data is not accessible, Modyn provides an open-source reproducible script and pre-scraped metadata to generate different versions of the CGLM dataset, e.g., by using the clean or non-clean and hierchical or non-hierarchical version of the original (non-continual) CGLM dataset [87, 114].
(5) The CLOC dataset [16]: CLOC is a big continual learning dataset on images with distribution shift. Modyn supports the version processed by Hammoud et al. [35].

While such data often is business-critical, to facilitate future research, we call for more datasets with distribution shift to be released. Releasing such datasets can help research to solve meaningful problems for practice. Modyn comes with tooling for analyzing pipelines. It provides an interactive dashboard based on Dash/Plotly that allows users to (a) analyze single pipelines, i.e., dive into the model and system metrics, and (b) compare pipelines to understand which policies perform best. Most plots in this paper have first been explored using this dashboard.

7 EVALUATION

We evaluate Modyn to answer the following three questions:

(1) How do data selection policies influence accuracy?
(2) How do different triggering policies compare? In particular, can drift-based policies be used to reduce pipeline cost while keeping accuracy?
(3) What is the impact of Modyn's parallelism, partitioning, and prefetching optimizations and how should the corresponding parameters be set to maximize throughput? How does the per-sample data ingestion throughput compare to reading data sequentially from local storage?

For all experiments, we use a server with two 16 Core AMD EPYC 7313 CPUs, 256 GB DRAM, a 4 TiB Samsung MZQL23T8HCLS NVMe, and a NVIDIA RTX 3090 GPU. We use gRPC 1.64.1, Postgres 15.2, PyTorch 2.2.1, NVIDIA GPU driver 545.23.06 with CUDA 12.3, on Ubuntu Server 22.04 with kernel 5.15. Modyn is compiled with GCC 12 and `-O3 -march=native`.

7.1 Data Selection

In this subsection, we explore the impact of data selection policies on pipeline accuracy. Each data selection policy needs to define a window of data, a presampling, and a downsampling policy. We pick the "finetuning" setting, i.e., we finetune the model from the previous trigger and set our window to contain the data since the last trigger. We mostly focus on downsamplers (in BtS mode) because they do not require domain-specific knowledge and are built for increasing accuracy on the current distribution.

Due to space constraints, we consider the yearbook dataset [121] and the CGLM-landmark dataset. We run all pipelines on three seeds and average the results. We shuffle the training data and use the currently trained composite model. We test presampling uniform at random (uniform, which samples a subset once and then trains on that for several epochs), class-balanced presampling, RS2 with and without replacement [73], loss downsampling [47], DLIS downsampling [47], and the margin, least conf., and entropy variants of uncertainty downsampling [22]. All policies are implemented in less than 130 lines of code.

7.1.1 yearbook dataset. The yearbook dataset classifies school yearbook pictures from 1930 to 2013. We follow Yao et al. [121] and use their "yearbooknet" CNN and the training hyperparameters with a batch size of 64, SGD with a learning rate of 0.001, and momentum 0.9. We also use their evaluation split. We trigger yearly, i.e., with the highest resolution possible for this dataset, train for 5 epochs per trigger, and use two warmup triggers where we do not apply data selection. Due to the small dataset size (33 431 training samples), we use a three year sliding window as an interval generation function (Section 3) to smoothen the accuracy curve and only run 50 % subset selection.

Full data training. In Figure 7, we show the accuracy matrix $m_{\sigma,P}$ of full data training on yearbook that we seamlessly obtain using Modyn's evaluation support. In the 1970s, we observe a drop in accuracy for models trained on data before this period, in-line with numbers from Yao et al. [121] (Figure 4a), indicating distribution shift. We hypothesize that, e.g., changing hairstyles over the decades could cause the shift. As expected, the highest accuracies lie on the diagonal of

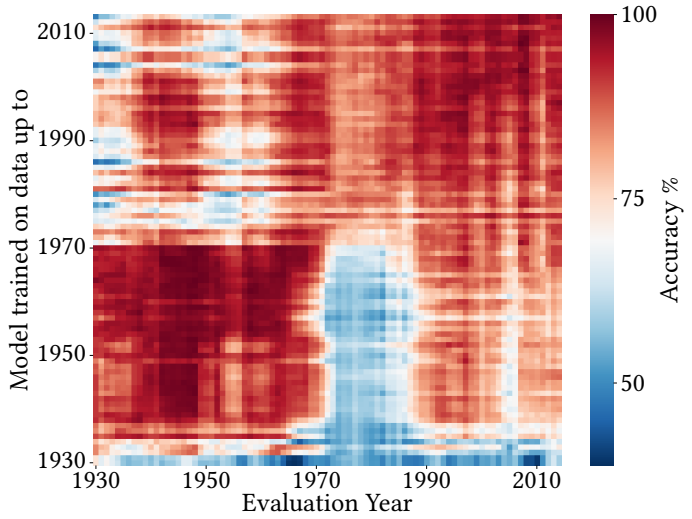

Fig. 7. Accuracy matrix for YEARBOOK full data training.

the matrix, and we can that see the first models underfit. The low accuracies in the upper left area show how models trained on newer data forget the past distribution.

50 % subset training. Figure 8 shows composite model accuracies per selection strategy in a boxplot. Generally, the uncertainty based downsamplers [22] perform best. Full data training has an average accuracy (pipeline score $\Sigma_{\sigma,P}$) of 92.3 %, and with 50 % selection, ENTROPY reaches 91.4 %, and LEAST CONF. and MARGIN reach 91.2 %. RS2 [73] reaches 88.8 % (w/o replacement)/88.4 % (w. replacement). LOSS and DLIS perform worse than uniform and class-bal. sampling on this dataset.

We investigate why the average accuracy is higher in Figure 9. DLIS's performance degrades during the drift period, while MARGIN is able to handle the drift better, similar to full data training. It is able to identify which data points are the most relevant during the shift. Overall, we find that with uncertainty-based downsamplers we almost reach full-data model accuracy with a 50 % training budget.

7.1.2 CGLM-LANDMARK dataset. This dataset classifies pictures from Wikipedia into 6 404 landmark classes. We follow Prabhu et al. [80] without filtering out uncleaned data, as downsampling might help to recognize unclean data. Despite applying weaker filter criteria, we obtain 361 671 samples

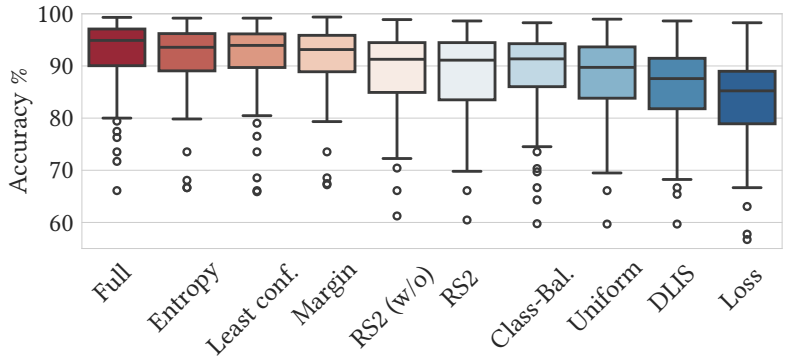

Fig. 8. Currently trained composite model accuracies for full data training and 50 % data selection on YEARBOOK.

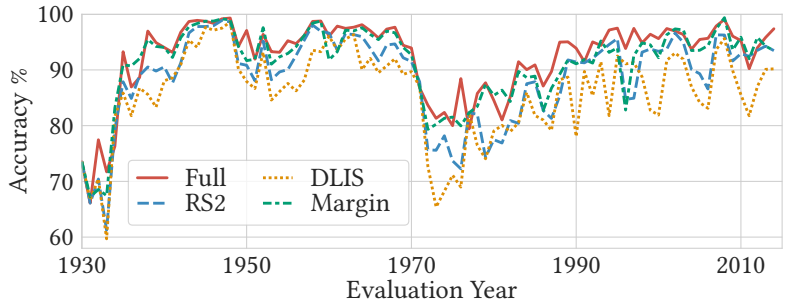

Fig. 9. Composite model accuracy over time for DLIS, MARGIN, and RS2 (w/o) on YEARBOOK.

before splitting the evaluation set, while Prabhu et al. [80] claim to obtain 430 K/580 K images (they mention both numbers). Since their data preprocessing is not public, we cannot investigate the differences. Following Prabhu et al. [80], we train a ResNet50 [38] with pretrained weights from ImageNet, and use SGD with a learning rate of 0.005 and momentum of 0.9. We use a batch size of 128 and train for 5 epochs per trigger. We trigger every year. Since the first years contain very little data, we use 5 warmup triggers. We evaluate using one year tumbling windows, and report top-5 accuracy since this is a hard classification task with 6 404 classes. We filter out the years 2005, 2006, and 2020 due to the low number of samples in the evaluation set.

Full data training. This dataset is a good example to showcase the difference between the currently trained and currently active composite model (Section 3.1). As seen in Figure 10, which shows the accuracy sequence $\Lambda_{\sigma,P}$, the currently trained model has a much higher accuracy over time, since due to its definition it connects the spikes instead of the point after the spike. The currently trained numbers are in-line with the numbers by Prabhu et al. [80]. The reason why the individual models have spikes is that many classes are mostly prevalent within a single year, i.e., there is a concentration of classes on one particular year. We explain this with the nature of the dataset: it is likely that landmark pages on Wikipedia get updated in batches, e.g., a user updates pictures of the Big Ben in London in 2015, and then they are not updated for several years again. Hence, models overfit to the current prevalent classes, forgetting about the old classes. In a traditional continual learning setup, this might not get noticed. Full data training has an average top-5 accuracy of 51.5 %.

50 %, 25 %, and 12.5 % subsets. For training on 50 % subsets, we show the top-5 accuracies of the composite models for the downsamplers in Figure 11. For this dataset with shifts in classes, MARGIN performs best (44 %), followed by DLIS (43.1 %) and RS2 (43 %). RS2, which simply goes through

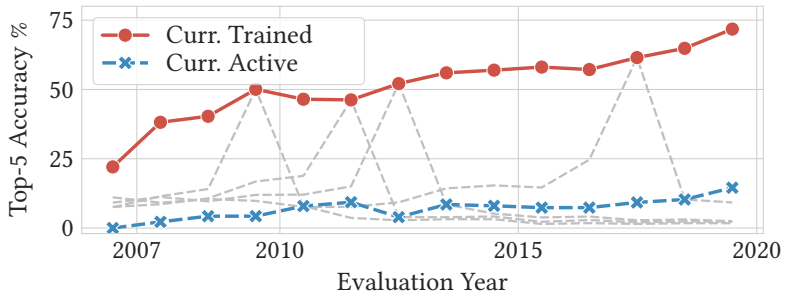

Fig. 10. Visualization of the currently trained vs. active composite model on CGLM-LANDMARK. The grey dashed lines are a subset of the models trained during the pipeline.

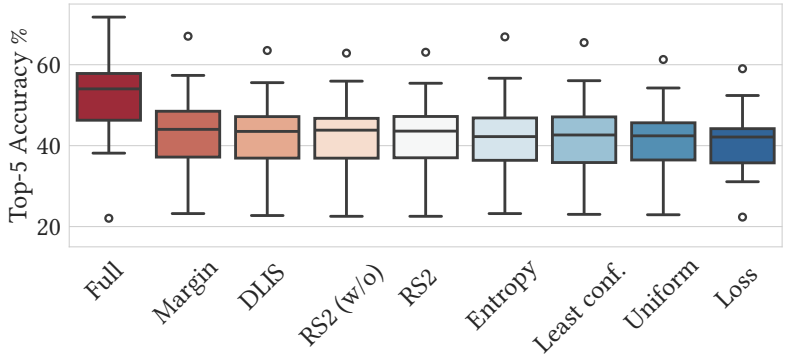

Fig. 11. Composite model accuracies for full data training and 50 % data selection on CGLM-LANDMARK.

the dataset as much as possible under the given budget, performs better than more sophisticated techniques like LEAST CONF. and ENTROPY.

On this dataset, for 25 % subsets, DLIS performs best (33.7 %), followed by RS2 (33.6 %). MARGIN (32.6 %) performs worse than RS2. For 12.5 % subsets, UNIFORM, RS2, and DLIS all reach around 23 % top-5 accuracy.

7.1.3 Takeaways. For YEARBOOK, where we have covariate shift, downsampling helps achieve near full-data performance on a 50 % budget. For CGLM-LANDMARK, where we have prior-probability shift, RS2, DLIS and MARGIN work well. This is motivating since these cheap sampling strategies do not require subject-specific knowledge. Future analyses might extend this to information retention [80] or more expensive downsamplers like CRAIG [63].

7.2 Triggering Policies

We explore triggering policies for full data training on YEARBOOK, using the setup from Section 7.1. We also explore the Kaggle ARXIV dataset to analyze a dataset with a different drift pattern and modality (text). We use the number of triggers instead of wall-clock time as a cost metric since we run the experiments on a shared machine. While all pipelines train on the same number of data points, fewer triggers are desirable due to system overhead per trigger and because the underlying assumption is that we cannot finetune on the fly due to costly deployment checks.

A plot of the cost-accuracy feasible set **F** (Section 3.1) for different triggering policies is shown in Figure 12. We use the currently active model because the currently trained model strongly favors fewer triggers: if we only trigger at the end, the model that has seen all data is by definition the currently trained model for all evaluations and would have very high accuracy. To fairly compare policies, we only consider the metrics after every pipeline triggered once since there is no active model before the first trigger. Otherwise, the missing initial values would skew the average. In general, the goal is to minimize the number of triggers while maximizing accuracy.

Time- and amount triggers. In Section 7.1, we trigger every year, which is the highest time resolution for YEARBOOK. Here, we explore triggering every 3 and 5 years, as well as every 500 and 1000 samples. Notably, triggering yearly is not optimal: Triggering every 3 years yields 26 instead of 75 triggers[4], but only a slightly lower average accuracy (92.8 % vs. 93.1 %). Triggering every 500 items performs similarly due to the even distribution of samples across years. When we trigger every 5 years, the performance drops to 92.4 % accuracy.

[4]As mentioned, we only consider metrics after *all pipelines* triggered once. While the yearly trigger overall fires 84 times, it fires 75 times after all triggers have fired once.

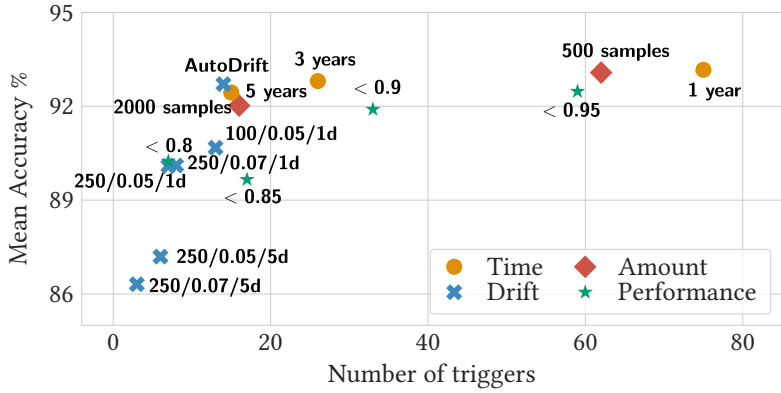

Fig. 12. Feasible set of triggering polices on YEARBOOK.

Performance-based triggers. These triggers fire when the model performance on a window drops below a threshold. For the first 3 500 samples we warm up and trigger at minimum every 3 years. We use windows of size 250 and test thresholds of 80 %, 85 %, 90 %, and 95 % accuracy. Generally, higher thresholds result in more frequent triggers and improved performance. Interestingly, the 80 % threshold performs better than 85 %. The 80 % threshold triggers slightly later, and the resulting model has a better performance than the model from the earlier 85 % trigger. Both models do not cross the threshold for some time, such that the overall average performance of 85 % is lower. *If labels are available*, performance-based triggers are a simple but well-performing triggering mechanism.

Drift-based triggering. The previous triggers rely on prior knowledge: we configure amount- and time triggers based on our experience on when drift occurs and how many samples there are. They also assume a constant drift frequency. This does not reflect reality where trend seasonality might be irregular [60]. Performance triggers require labels as well as expected model performance. Drift-based policies do not require this prior knowledge, as they use information from the data itself. We perform the same warm up as for performance triggers. We test MMD (using alibi-detect [109]) on embeddings without PCA, use threshold-based triggering, and sweep across detection intervals (100, 250, 500), thresholds (0.05, 0.07, 0.09), and window sizes (1 day, 5 days) of which we show a subset in Figure 12. We also test MODYN's automatic threshold mechanism that triggers when the drift score is in the top 5 % of the 15 previously observed scores (`AutoDrift`).

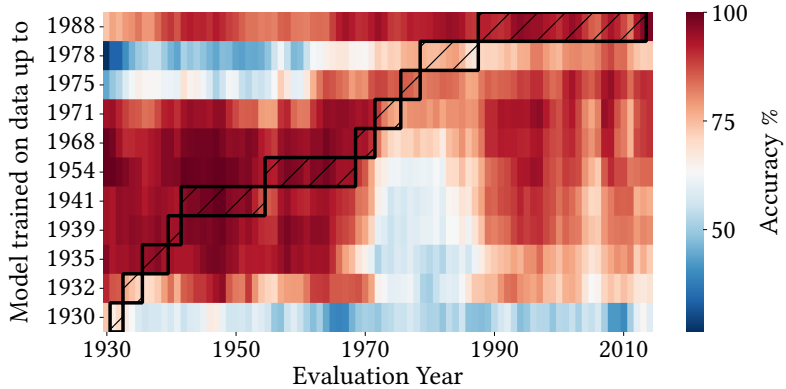

Fig. 13. The drift MMD (250/0.05/1d) triggering policy on YEARBOOK. The black boxes indicate when a model is active, i.e., the time during which it would be used for inference.

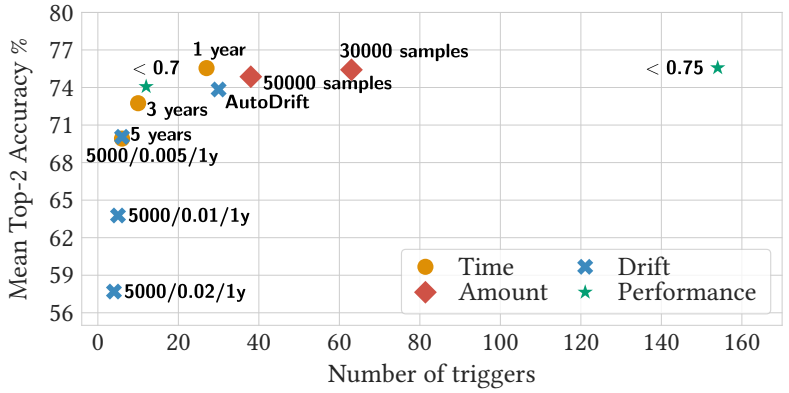

Fig. 14. Feasible set of triggering polices on Kaggle ARXIV.

On YEARBOOK, the drift policies trigger not as often. A detection interval of 250 samples, with a threshold of 0.05 and 1 day window performs well, as it only triggers 8 times while still having an average accuracy of 90.1 %. In Figure 13 we show how the policy navigates around the drift area: consider the model trained up to 1954. Shortly before the model's performance degrades in the 1970s, a trigger is fired (end of black box) and the model up to 1968 is trained (finetuned on the data seen since 1954). Note that the drift policy does not have information about future model performance and just uses the information from the data itself to make these decisions. The other configurations perform slightly worse as they are less sensitive. For example, increasing the window to 5 days decreases the number of triggers to 3 with an accuracy of 88 %. A larger window size smoothens the drift scores, as the new data needs to be significantly different from the data in the larger window.

The `AutoDrift` policy performs well, as it triggers 14 times with an average accuracy of 92.7 %. Importantly, this policy does not require information on the drift metric magnitude. It uses a simple outlier detection mechanism, making drift detection more user-friendly. Overall, these results are promising as the drift policies successfully navigate around YEARBOOK's drift area without using prior information on the dataset.

Kaggle ARXIV dataset. The task of this large (~ 2 M samples from 1990 to 2024) textual dataset is to classify paper titles from arXiv into 172 categories. Textual data uses embeddings for drift detection. The dataset has a different drift pattern, as performance slowly degrades over time. We train a DistilBERT model [91] with AdamW, learning rate 0.00002, and 5 epochs per trigger. We evaluate each pipeline using 6 month tumbling windows, and warm up the drift and performance triggers for 20 k samples.

We show the cost-accuracy scatter in Figure 14. As more papers are submitted each year, the data density increases, and amount triggers fire more frequently than time triggers on this dataset. Performance-triggers strongly depend on the threshold, as the 75 % threshold triggers 154 times while 70 % triggers only 12 times. The drift trigger with a threshold of 0.0005 and 1 year windows almost matches the 5 year trigger with 6 triggers and 70 % top-2 accuracy. `AutoDrift` again performs well with 30 triggers and 73.8 % top-2 accuracy, without the need to configure performance- or drift thresholds manually. Overall, for both YEARBOOK and Kaggle ARXIV, data-centric triggering can reduce pipeline cost.

7.3 Training Throughput

In this experiment, we train a model and evaluate the training throughput for different parameters to show how MODYN's optimizations impact training throughput.

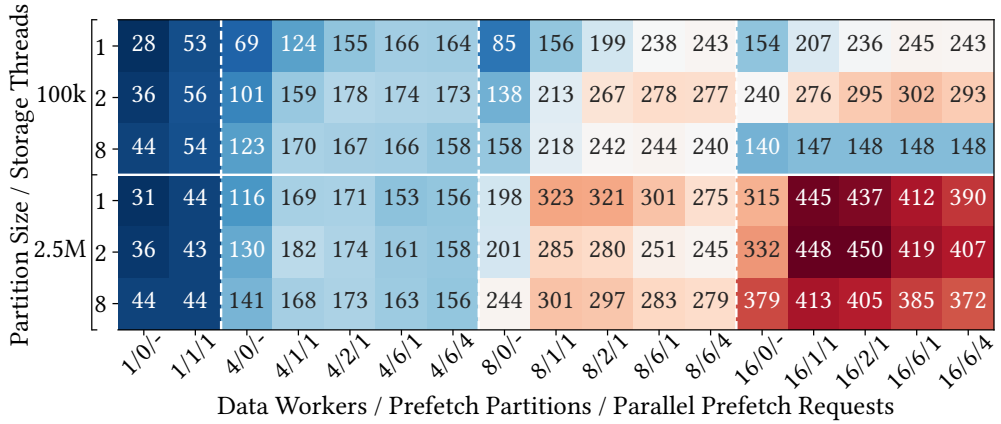

Fig. 15. Throughput (x1000) for Criteo (Section 7.3.1). The first three rows show the results for partitions with 100 k samples, and the last three rows for partitions with 2.5 M samples. For each partition, we show results for 1, 2, and 8 threads at storage.

Setup. We configure the Postgres storage instance to use 96 maximum parallel workers, with 2 maximum workers per gather. All components are deployed on the same machine, to avoid measuring network bandwidth instead of Modyn throughput. We run all measurements three times and report the average results.

Workloads. We consider two workloads. In the first workload, we train a DLRM recommendation model [69] on the Criteo 1TB click stream dataset [23], which provides user data over 24 days, with roughly 180 million samples per day. Given categorical and numerical features, the task is to predict whether a user will click on a suggestion. We use this scenario because the high number of samples with thousands of samples per file stress-tests Modyn's data-retrieval implementation, in comparison to simpler scenarios such as vision models. We use NVIDIA's DLRM implementation [72] and follow their "small" setup with a batch size of 65 536. At the storage, we use Modyn's `BinaryFileWrapper`, i.e., the 160 B samples are stored in a fixed row size binary file format, distributed across files containing ca. 180 000 samples each. The bytes parser function at the trainer creates input tensors directly from a `memoryview` to avoid unnecessary copies. The second workload trains a ResNet50 [38] on CGLM, as in Section 7.1.2. We use Modyn's `SingleSampleFileWrapper`, i.e., each sample is stored in one JPEG file. The bytes parser function converts the data to an RGB `PIL.Image` on which the dataset applies image augmentations (e.g., resize and crop) to generate a tensor.

Throughput measurement. The size of each partition, as discussed in Section 5.2, directly dictates the total number of partitions within the trigger training set. Every worker gets an equal share of each partition. Note that we do *not* synchronize CUDA after each batch, i.e., we allow PyTorch to perform computation while the next batch is being fetched. We do not shuffle for this benchmark. We measure the time from the start of the training loop to the last model update and obtain the throughput by dividing the time by the total number of samples in the trigger.

7.3.1 Criteo Throughput. In Figure 15 we show the throughput of training in the Criteo workload. We test both a partition size of 100 k (≈ 1.53 batches per partition) and 2.5 M samples (≈ 38.15 batches per partition). We first discuss the results for a single thread at the storage, i.e., the top row per partition size.

Data loader workers. Using one data loader worker and no prefetching, there is no difference between the small and big partitions. When enabling prefetching of one partition, i.e., loading the next partition into a buffer before its batches are requested, the throughput increases by 1.89x and

1.42x for small and large partitions, respectively. Note that *prefetching a partition* means that each worker prefetches its share of a partition. The smaller partitions benefit more from prefetching.

Increasing the number of workers generally increases throughput. For example, for the large partitions with one prefetched partition, using four workers improves throughput by 3.84x, using eight workers by 7.34x, and using 16 workers by 1.11x, compared to a single worker. This increase is explained by the ability to fetch the keys and data from selector and storage in parallel, and the parallelization of the bytes-to-tensor transformation.

Notably, in contrast to the single worker scenario, the larger partition size has higher throughput with multiple workers than the smaller partition size. For example, for 16 workers and with prefetching one partition (16/1/1), the larger partition setting has 2.15x higher throughput than the smaller partition setting. This is because for the small partitions and 16 workers, a partition does not even cover 10 % of a batch. For larger partitions, the workers have ~ 2.5 batches per partition, which is sufficient to saturate the GPU. More workers favor larger partition sizes.

Additional prefetching. We can both prefetch more partitions and request more partitions in parallel. For the single threaded storage and the smaller partitions, increasing the number of prefetched partitions–while keeping one parallel request–increases throughput, especially for higher number of workers (e.g. 4/6/1, 8/6/1). However, there are diminishing returns to increasing the number of prefetch partitions. For example, for four workers, going from 1 (4/1/1) to 2 (4/2/1) prefetched partitions increases throughput by 1.25x, but going from 2 (4/2/1) to 6 (4/6/1) only increases throughput by 1.07x. As soon as we fill up the buffer faster than data is consumed, there is no benefit from further prefetching data. When using more workers, the benefit of prefetching more partitions is higher because fixed size partitions are distributed across all workers. Prefetching one partition with four workers prefetches the same amount of samples as eight workers that prefetch 2 partitions.

Using more parallel prefetch requests does not improve throughput. This is explained by the fact that MODYN's components have upper limits of load they can handle: Postgres has a maximum number of worker threads, the number of gRPC worker threads is limited, and the disk holding the databases and dataset has limited bandwidth. Many parallel requests overload the system.

Multi-threaded storage. The data retrieval at the storage can use multiple threads (Section 5.3.1). We find that using 2 threads increases throughput, but using 8 threads overloads the system and may lead to worse performance. The throughput increases are higher for smaller number of workers. For example, for the setting of one worker and no prefetching (1/0/-), on the small partitions, parallelism increases throughput by 1.29x and 1.57x for 2 and 8 threads, respectively. For 16 workers (16/0/-), increasing the storage threads from 2 to 8 decreases performance to 0.58x.

The reason for the performance decrease with 8 threads is that, while we parallelize data retrieval, there is a limit on the number of parallel Postgres workers. If 16 workers send a request that gets split upon 8 threads, and each thread emits one query that executes with 2 workers in parallel, we need 256 Postgres workers, amplified with increasing parallel prefetch requests. Nevertheless, in the following, we show that we reach sufficiently high training throughput.

Comparison to local training. We compare MODYN to local training to quantify its overhead. For this, we read data sequentially from 90 binary files containing 30 M samples. Each dataloader worker is assigned a share of the files. Note that this not only removes the communication and gRPC overhead, but also removes the sample-level data selection. MODYN loads each sample individually by key, but the local approach loads entire files sequentially and emits all samples in them.

The results are shown in Figure 16a. For each number of dataloader workers, we compare the best throughput we measure for Figure 15 against the local throughput. MODYN reaches 98 %, 85.4 %, 77.8 %, and 71 % of the optimal local performance for 1, 4, 8, and 16 workers. Despite having a

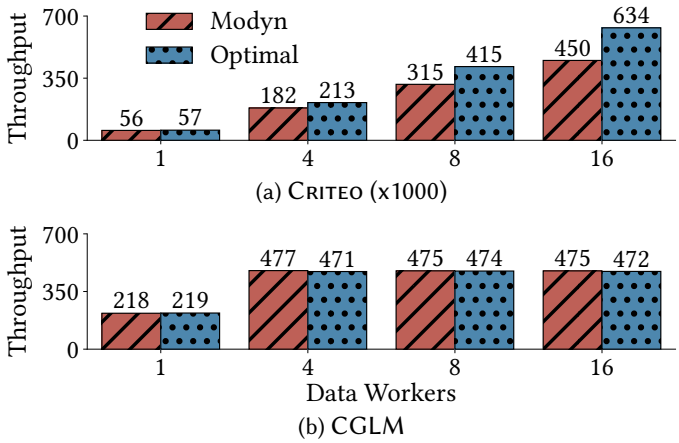

Fig. 16. Modyn throughput vs. optimal throughput when loading data sequentially locally.

much more involved data retrieval process, Modyn reaches over 70 % of optimal throughput for the challenging recommendation system case.

7.3.2 CGLM Throughput. Figure 16b compares Modyn to the optimal local throughput. As soon as 4 workers are used, the throughput stagnates at around 475 samples/s. Modyn basically reaches the optimal local throughput for all configurations. This is because computer vision workloads like CGLM (or yearbook) are *compute-bound*, while training a recommendation systems model is *memory-bound* [1, 21, 66, 124]. Four workers, with Modyn's C++ storage and selector implementations, supply the model with enough data.

8 CONCLUSION AND FUTURE WORK

We present the data-centric Modyn orchestrator for ML pipelines on growing datasets, together with an ecosystem of tooling, benchmarks, and concepts to fairly compare ML pipelines. Modyn implements various triggering and data selection policies and optimizes the system infrastructure under the hood for high-throughput sample-level data selection. For future work from an ML perspective, it is interesting to extend our analyses across more benchmark datasets, explore more presampling policies, and consider metrics such as information retention [16, 80]. Future work might also use Modyn and the ideas on comparing pipelines (Section 3) to find optimal pipeline configurations on benchmarks with an AutoML approach [40, 89], and extend Modyn to the unsupervised case and train generative large language models [111]. Due to the right to data deletion in regulations such as GDPR and CCPA [29, 100], support for data deletion (dynamic instead of just growing datasets) also is an interesting feature [15, 112].

From a systems and database perspective, additional research opportunities arise. For example, some selection policies require to store huge embeddings over time [82] which is a data management challenge in itself. It is also not yet clear how to optimally compress and store multiple model versions over time [103, 104]. Last, since Modyn is a centralized system, it can be leveraged for provenance analyses, such as understanding why retraining and selection decisions were made [19, 68, 77, 118]. Modyn provides a rich environment for such research on different parts of the training pipeline.

ACKNOWLEDGMENTS

Maximilian Böther is supported by the Swiss National Science Foundation (Project Number 200021_204620). Ties Robroek is supported by the Independent Research Fund Denmark's Sapere Aude program under grant agreement number 0171-00061B. We thank Francesco Deaglio, Jingyi

Zhu, Robin Oester, and Foteini Strati for their contributions to Modyn's codebase. We also thank the anonymous reviewers for their helpful comments.

REFERENCES

[1] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair. 2021. Accelerating recommendation system training by leveraging popular choices. *Proceedings of the VLDB Endowment* 15, 1 (2021). https://doi.org/10.14778/3485450.3485462

[2] Gabriel J. Aguiar and Alberto Cano. 2024. A comprehensive analysis of concept drift locality in data streams. *Knowledge-Based Systems* 289 (2024). https://doi.org/10.1016/j.knosys.2024.111535

[3] Rahaf Aljundi, Eugene Belilovsky, Tinne Tuytelaars, Laurent Charlin, Massimo Caccia, Min Lin, and Lucas Page-Caccia. 2019. Online Continual Learning with Maximal Interfered Retrieval. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

[4] Rahaf Aljundi, Min Lin, Baptiste Goujaud, and Yoshua Bengio. 2019. Gradient based sample selection for online continual learning. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

[5] Amazon. 2023. Amazon SageMaker. https://docs.aws.amazon.com/sagemaker/index.html.

[6] arXiv.org submitters. 2024. arXiv Kaggle Dataset. https://doi.org/10.34740/KAGGLE/DSV/7548853

[7] Jihwan Bang, Heesu Kim, YoungJoon Yoo, Jung-Woo Ha, and Jonghyun Choi. 2021. Rainbow Memory: Continual Learning with a Memory of Diverse Samples. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. https://doi.org/10.1109/cvpr46437.2021.00812

[8] Roberto Souto Maior Barros and Silas Garrido T. Carvalho Santos. 2018. A large-scale comparison of concept drift detectors. *Information Sciences* 451–452 (2018). https://doi.org/10.1016/j.ins.2018.04.014

[9] Michael Bayer. 2012. SQLAlchemy. In *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*. aosabook.org. http://aosabook.org/en/sqlalchemy.html

[10] Denis Baylor, Kevin Haas, Konstantinos Katsiapis, Sammy Leong, Rose Liu, Clemens Mewald, Hui Miao, Neoklis Polyzotis, Mitchell Trott, and Martin Zinkevich. 2019. Continuous Training for Production ML in the TensorFlow Extended (TFX) Platform. In *Proceedings of the USENIX Conference on Operational Machine Learning (OpML)*.

[11] BentoML. 2023. BentoML: Github Organization. https://github.com/bentoml/. Accessed: 2023-11-28.

[12] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. 2022. Ekya: Continuous Learning of Video Analytics Models on Edge Compute Servers. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[13] Lukas Biewald. 2020. Experiment Tracking with Weights and Biases. https://www.wandb.com/.

[14] Maximilian Böther, Foteini Strati, Viktor Gsteiger, and Ana Klimovic. 2023. Towards A Platform and Benchmark Suite for Model Training on Dynamic Datasets. In *Proceedings of the Workshop on Machine Learning and Systems (EuroMLSys)*. https://doi.org/10.1145/3578356.3592585

[15] Lucas Bourtoule, Varun Chandrasekaran, Christopher A. Choquette-Choo, Hengrui Jia, Adelin Travers, Baiwu Zhang, David Lie, and Nicolas Papernot. 2021. Machine Unlearning. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. https://doi.org/10.1109/sp40001.2021.00019

[16] Zhipeng Cai, Ozan Sener, and Vladlen Koltun. 2021. Online Continual Learning with Natural Distribution Shifts: An Empirical Study with Visual Data. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. https://doi.org/10.1109/iccv48922.2021.00817

[17] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[18] Gert Cauwenberghs and Tomaso A. Poggio. 2000. Incremental and Decremental Support Vector Machine Learning. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

[19] Adriane Chapman, Paolo Missier, Giulia Simonelli, and Riccardo Torlone. 2020. Capturing and querying fine-grained provenance of preprocessing pipelines in data science. *Proceedings of the VLDB Endowment* 14, 4 (2020). https://doi.org/10.14778/3436905.3436911

[20] Andrew Chen, Andy Chow, Aaron Davidson, Arjun DCunha, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Clemens Mewald, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Avesh Singh, Fen Xie, Matei Zaharia, Richard Zang, Juntai Zheng, and Corey Zumar. 2020. Developments in MLflow: A System to Accelerate the Machine Learning Lifecycle. In *Proceedings of the International Workshop on Data Management for End-to-End Machine Learning (DEEM)*. https://doi.org/10.1145/3399579.3399867

[21] Runxiang Cheng, Chris Cai, Selman Yilmaz, Rahul Mitra, Malay Bag, Mrinmoy Ghosh, and Tianyin Xu. 2023. Towards GPU Memory Efficiency for Distributed Training at Scale. In *Proceedings of the Symposium on Cloud Computing (SoCC)*. https://doi.org/10.1145/3620678.3624661

[22] Cody Coleman, Christopher Yeh, Stephen Mussmann, Baharan Mirzasoleiman, Peter Bailis, Percy Liang, Jure Leskovec, and Matei Zaharia. 2020. Selection via Proxy: Efficient Data Selection for Deep Learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*.
[23] Criteo. 2013. Download Terabyte Click Logs. https://labs.criteo.com/2013/12/download-terabyte-click-logs/.
[24] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.
[25] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Tilmann Rabl, and Volker Markl. 2019. Continuous Deployment of Machine Learning Pipelines. https://doi.org/10.5441/002/EDBT.2019.35
[26] Tom Diethe, Tom Borchert, Eno Thereska, Borja Balle, and Neil Lawrence. 2018. Continual Learning in Practice. In *Proceedings of the Workshop on Continual Learning at NeurIPS*. https://doi.org/10.48550/ARXIV.1903.05202
[27] Gregory Ditzler and Robi Polikar. 2011. Hellinger distance based drift detection for nonstationary environments. In *Proceedings of the Symposium on Computational Intelligence in Dynamic and Uncertain Environments (CIDUE)*, Vol. 3741. https://doi.org/10.1109/cidue.2011.5948491
[28] Alex Egg. 2021. Online Learning for Recommendations at Grubhub. In *Proceedings of the Conference on Recommender Systems (RecSys)*. https://doi.org/10.1145/3460231.3474599
[29] European Union. 2016. Art. 17 GDPR: Right to erasure ('right to be forgotten'). https://gdpr.eu/article-17-right-to-be-forgotten/.
[30] Evidently AI. 2024. Evidently: Collaborative AI observability platform. https://www.evidentlyai.com/. Accessed: 2024-06-26.
[31] Clement Farabet and Nicolas Koumchatzky. 2020. Presentation: Inside NVIDIA's AI Infrastructure for Self-driving Cars. In *Presentations of the USENIX Conference on Operational Machine Learning (OpML)*. https://www.usenix.org/conference/opml20/presentation/farabet
[32] Yasir Ghunaim, Adel Bibi, Kumail Alhamoud, Motasem Alfarra, Hasan Abed Al Kader Hammoud, Ameya Prabhu, Philip H.S. Torr, and Bernard Ghanem. 2023. Real-Time Evaluation in Online Continual Learning: A New Hope. In *Proceedings of te IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. https://doi.org/10.1109/cvpr52729.2023.01144
[33] Stefan Grafberger, Paul Groth, and Sebastian Schelter. 2023. Automating and Optimizing Data-Centric What-If Analyses on Native Machine Learning Pipelines. *Proceedings of the ACM on Management of Data (SIGMOD)* 1, 2 (2023). https://doi.org/10.1145/3589273
[34] Arthur Gretton, Karsten M. Borgwardt, Malte J. Rasch, Bernhard Schölkopf, and Alexander Smola. 2012. A Kernel Two-Sample Test. *Journal of Machine Learning Research* 13, 25 (2012).
[35] Hasan Abed Al Kader Hammoud, Ameya Prabhu, Ser-Nam Lim, Philip H. S. Torr, Adel Bibi, and Bernard Ghanem. 2023. Rapid Adaptation in Online Continual Learning: Are We Evaluating It Right?. In *Proceedings of the International Conference on Computer Vision (ICCV)*. https://doi.org/10.1109/iccv51070.2023.01728
[36] Wei Hao, Daniel Mendoza, Rafael da Silva, Deepak Narayanan, and Amar Phanishayee. 2024. MGit: A Model Versioning and Management System. In *Proceedings of the International Conference on Machine Learning (ICML)*. https://doi.org/10.48550/ARXIV.2307.07507
[37] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2018.00059
[38] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. https://doi.org/10.1109/cvpr.2016.90
[39] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, and Joaquin Quiñonero Candela. 2014. Practical Lessons from Predicting Clicks on Ads at Facebook. In *Proceedings of the International Workshop on Data Mining for Online Advertising (ADKDD)*. https://doi.org/10.1145/2648584.2648589
[40] Xin He, Kaiyong Zhao, and Xiaowen Chu. 2021. AutoML: A survey of the state-of-the-art. *Knowledge-Based Systems* 212 (2021), 106622. https://doi.org/10.1016/j.knosys.2020.106622
[41] Hopsworks AB. 2024. Hopsworks Feature Monitoring. https://www.hopsworks.ai/dictionary/feature-monitoring.
[42] Chip Huyen. 2020. Machine learning is going real-time. https://huyenchip.com/2020/12/27/real-time-machine-learning.html.
[43] Chip Huyen. 2022. *Designing Machine Learning Systems*. O'Reilly Media, Inc.
[44] Chip Huyen. 2022. Real-time machine learning: challenges and solutions. https://huyenchip.com/2022/01/02/real-time-machine-learning-challenges-and-solutions.html.

[45] Ruoxi Jia, Fan Wu, Xuehui Sun, Jiacen Xu, David Dao, Bhavya Kailkhura, Ce Zhang, Bo Li, and Dawn Song. 2021. Scalability vs. Utility: Do We Have to Sacrifice One for the Other in Data Importance Quantification?. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. https://doi.org/10.1109/cvpr46437.2021.00814

[46] Jaykumar Kasundra, Claudia Schulz, Melicaalsadat Mirsafian, and Stavroula Skylaki. 2023. A Framework for Monitoring and Retraining Language Models in Real-World Applications. *arXiv preprint* (2023). https://doi.org/10.48550/arXiv.2311.09930

[47] Angelos Katharopoulos and François Fleuret. 2018. Not All Samples Are Created Equal: Deep Learning with Importance Sampling. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[48] KrishnaTeja Killamsetty, Durga Sivasubramanian, Ganesh Ramakrishnan, Abir De, and Rishabh K. Iyer. 2021. GRADMATCH: Gradient Matching based Data Subset Selection for Efficient Deep Model Training. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[49] Andreas Kirsch. 2023. Does 'Deep Learning on a Data Diet' reproduce? Overall yes, but GraNd at Initialization does not. *Transactions on Machine Learning Research* (2023).

[50] Hyunseo Koh, Dahyun Kim, Jung-Woo Ha, and Jonghyun Choi. 2022. Online Continual Learning on Class Incremental Blurry Task Configuration with Anytime Inference. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[51] Alex Krizhevsky and Geoffrey Hinton. 2009. *Learning multiple layers of features from tiny images*. Technical Report. University of Toronto, Toronto, Ontario. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[52] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. 2022. Plumber: Diagnosing and Removing Performance Bottlenecks in Machine Learning Data Pipelines. In *Proceedings of the Conference on Machine Learning and Systems (MLSys)*.

[53] Angeliki Lazaridou, Adhiguna Kuncoro, Elena Gribovskaya, Devang Agrawal, Adam Liska, Tayfun Terzi, Mai Gimenez, Cyprien de Masson d'Autume, Tomás Kociský, Sebastian Ruder, Dani Yogatama, Kris Cao, Susannah Young, and Phil Blunsom. 2021. Mind the Gap: Assessing Temporal Generalization in Neural Language Models. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

[54] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998). https://doi.org/10.1109/5.726791

[55] Aodong Li, Alex Boyd, Padhraic Smyth, and Stephan Mandt. 2021. Detecting and Adapting to Irregular Distribution Shifts in Bayesian Online Learning. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

[56] Hanmo Liu, Shimin Di, and Lei Chen. 2023. Incremental Tabular Learning on Heterogeneous Feature Space. *Proceedings of the International Conference on Management of Data (SIGMOD)* 1, 1 (2023). https://doi.org/10.1145/3588698

[57] David Lopez-Paz and Maxime Oquab. 2017. Revisiting Classifier Two-Sample Tests. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[58] David Lopez-Paz and Marc'Aurelio Ranzato. 2017. Gradient Episodic Memory for Continual Learning. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*.

[59] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, Joao Gama, and Guangquan Zhang. 2018. Learning under Concept Drift: A Review. *IEEE Transactions on Knowledge and Data Engineering* (2018). https://doi.org/10.1109/tkde.2018.2876857

[60] Ananth Mahadevan and Michael Mathioudakis. 2023. Cost-Effective Retraining of Machine Learning Models. *arXiv preprint* (2023). https://doi.org/10.48550/arXiv.2310.04216

[61] Kiran Kumar Matam, Hani Ramezani, Fan Wang, Zeliang Chen, Yue Dong, Maomao Ding, Zhiwei Zhao, Zhengyu Zhang, Ellie Wen, and Assaf Eisenman. 2024. QuickUpdate: a Real-Time Personalization System for Large-Scale Recommendation Models. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[62] Sören Mindermann, Jan Markus Brauner, Muhammed Razzak, Mrinank Sharma, Andreas Kirsch, Winnie Xu, Benedikt Höltgen, Aidan N. Gomez, Adrien Morisot, Sebastian Farquhar, and Yarin Gal. 2022. Prioritized Training on Points that are Learnable, Worth Learning, and not yet Learnt. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[63] Baharan Mirzasoleiman, Jeff A. Bilmes, and Jure Leskovec. 2020. Coresets for Data-efficient Training of Machine Learning Models. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[64] Rishabh Misra. 2022. News Category Dataset. *arXiv* (2022).

[65] Akshay Naresh Modi, Chiu Yuen Koo, Chuan Yu Foo, Clemens Mewald, Denis M. Baylor, Eric Breck, Heng-Tze Cheng, Jarek Wilkiewicz, Levent Koc, Lukasz Lew, Martin A. Zinkevich, Martin Wicke, Mustafa Ispir, Neoklis Polyzotis, Noah Fiedel, Salem Elie Haykal, Steven Whang, Sudip Roy, Sukriti Ramesh, Vihan Jain, Xin Zhang, and Zakaria Haque. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*. https://doi.org/10.1145/3097983.3098021

[66] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. 2022. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/3470496.3533727

[67] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. 2021. tf.data: a machine learning data processing framework. *Proceedings of the VLDB Endowment* 14, 12 (2021). https://doi.org/10.14778/3476311.3476374

[68] Mohammad Hossein Namaki, Avrilia Floratou, Fotis Psallidas, Subru Krishnan, Ashvin Agrawal, Yinghui Wu, Yiwen Zhu, and Markus Weimer. 2020. Vamsa: Automated Provenance Tracking in Data Science Scripts. In *Proceedings of the International Conference on Knowledge Discovery & Data Mining (KDD)*. https://doi.org/10.1145/3394486.3403205

[69] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. (2019). https://doi.org/10.48550/ARXIV.1906.00091

[70] Neptune. 2023. Neptune.ai ML Metadata Store. https://neptune.ai/.

[71] NVIDIA. 2023. NVIDIA Triton Inference Server. https://developer.nvidia.com/nvidia-triton-inference-server. Accessed: 2023-11-28.

[72] NVIDIA. 2024. NVIDIA DLRM Example Implementation. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Recommendation/DLRM. Accessed: 2024-06-26.

[73] Patrik Okanovic, Roger Waleffe, Vasilis Mageirakos, Konstantinos E. Nikolakakis, Amin Karbasi, Dionysis Kalogerias, Nezihe Merve Gürel, and Theodoros Rekatsinas. 2023. Repeated Random Sampling for Minimizing the Time-to-Accuracy of Learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[74] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. In *Proceedings of the Workshop on ML Systems at NeurIPS*. https://doi.org/10.48550/arXiv.1712.06139

[75] Andrei Paleyes, Raoul-Gabriel Urma, and Neil D. Lawrence. 2022. Challenges in Deploying Machine Learning: A Survey of Case Studies. *Comput. Surveys* 55, 6 (2022). https://doi.org/10.1145/3533378

[76] Mansheej Paul, Surya Ganguli, and Gintare Karolina Dziugaite. 2021. Deep Learning on a Data Diet: Finding Important Examples Early in Training. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*.

[77] Débora Pina, Adriane Chapman, Daniel De Oliveira, and Marta Mattoso. 2023. Deep Learning Provenance Data Integration: a Practical Approach. In *Proceedings of the ACM Web Conference (WWW)*. https://doi.org/10.1145/3543873.3587561

[78] Robi Polikar, Lalita Upda, Satish S. Upda, and Vasant Honavar. 2001. Learn++: an incremental learning algorithm for supervised neural networks. *IEEE Transactions on Systems, Man and Cybernetics, Part C* 31, 4 (2001). https://doi.org/10.1109/5326.983933

[79] Omead Pooladzandi, David Davini, and Baharan Mirzasoleiman. 2022. Adaptive Second Order Coresets for Data-efficient Machine Learning. In *Proceedings of the International Conference on Machine Learning (ICML)*. https://proceedings.mlr.press/v162/pooladzandi22a.html

[80] Ameya Prabhu, Zhipeng Cai, Puneet Dokania, Philip Torr, Vladlen Koltun, and Ozan Sener. 2023. Online Continual Learning Without the Storage Constraint. *arXiv preprint* (2023). https://doi.org/10.48550/arXiv.2305.09253

[81] Ameya Prabhu, Philip H. S. Torr, and Puneet K. Dokania. 2020. GDumb: A Simple Approach that Questions Our Progress in Continual Learning. In *Proceedings of the European Conference on Computer Vision (ECCV)*. https://doi.org/10.1007/978-3-030-58536-5_31

[82] Garima Pruthi, Frederick Liu, Satyen Kale, and Mukund Sundararajan. 2020. Estimating Training Data Influence by Tracing Gradient Descent. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*.

[83] Pydantic Contributors. 2024. Pydantic Documentation. https://docs.pydantic.dev/latest/. Accessed: 2024-07-07.

[84] PyTorch Serve Contributors. 2020. TorchServe: Docs. https://pytorch.org/serve/. Accessed: 2023-11-28.

[85] Stephan Rabanser, Stephan Günnemann, and Zachary C. Lipton. 2019. Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

[86] Srikumar Ramalingam, Daniel Glasner, Kaushal Patel, Raviteja Vemulapalli, Sadeep Jayasumana, and Sanjiv Kumar. 2021. Less is more: Selecting informative and diverse subsets with balancing constraints. (2021). https://doi.org/10.48550/arXiv.2104.12835

[87] Elias Ramzi, Nicolas Audebert, Clément Rambour, André Araújo, Xavier Bitot, and Nicolas Thome. 2023. Optimization of Rank Losses for Image Retrieval. *CoRR* abs/2309.08250 (2023). https://doi.org/10.48550/ARXIV.2309.08250

[88] Sergey Redyuk, Zoi Kaoudi, Volker Markl, and Sebastian Schelter. 2021. Automating Data Quality Validation for Dynamic Data Ingestion. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. https://doi.org/10.5441/002/EDBT.2021.07

[89] Sergey Redyuk, Zoi Kaoudi, Sebastian Schelter, and Volker Markl. 2024. Assisted design of data science pipelines. *The VLDB Journal* 33, 4 (2024). https://doi.org/10.1007/s00778-024-00835-2

[90] Gordon J. Ross, Niall M. Adams, Dimitris K. Tasoulis, and David J. Hand. 2012. Exponentially weighted moving average charts for detecting concept drift. *Pattern Recognition Letters* 33, 2 (2012). https://doi.org/10.1016/j.patrec.2011.08.019

[91] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2020. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. In *Proceedings of the Workshop on Energy Efficient Machine Learning and Cognitive Computing at NeurIPS*.

[92] Sebastian Schelter, Stefan Grafberger, Shubha Guha, Bojan Karlas, and Ce Zhang. 2023. Proactively Screening Machine Learning Pipelines with ARGUSEYES. In *Companion of the International Conference on Management of Data (SIGMOD)*. ACM. https://doi.org/10.1145/3555041.3589682

[93] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating large-scale data quality verification. *Proceedings of the VLDB Endowment* 11, 12 (2018). https://doi.org/10.14778/3229863.3229867

[94] Sebastian Schelter, Tammo Rukat, and Felix Biessmann. 2020. Learning to Validate the Predictions of Black Box Classifiers on Unseen Data. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. https://doi.org/10.1145/3318464.3380604

[95] Shreya Shankar, Rolando Garcia, Joseph M. Hellerstein, and Aditya G. Parameswaran. 2024. "We Have No Idea How Models will Behave in Production until Production": How Engineers Operationalize Machine Learning. In *Proceedings of Conference on Computer-Supported Cooperative Work and Social Computing (CSCW)*. https://doi.org/10.1145/3653697

[96] Shreya Shankar, Bernease Herman, and Aditya G. Parameswaran. 2022. Rethinking Streaming Machine Learning Evaluation. In *Proceedings of the ML Evaluation Standards Workshop at ICLR*. https://doi.org/10.48550/arXiv.2205.11473

[97] Shreya Shankar and Aditya G. Parameswaran. 2022. Towards Observability for Production Machine Learning Pipelines. *Proceedings of the VLDB Endowment* 15, 13 (2022). https://doi.org/10.14778/3565838.3565853

[98] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. 2022. Ekko: A Large-Scale Deep Learning Recommender System with Low-Latency Model Update. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[99] Maciej Sobczak and GitHub Contributors. 2023. SOCI - The C++ Database Access Library. https://github.com/SOCI/soci. Accessed: 2023-11-28.

[100] State of California, USA. 2018. Section 1798.130 CCPA. https://ccpa-info.com/california-consumer-privacy-act-full-text/.

[101] Monika Steidl, Michael Felderer, and Rudolf Ramler. 2023. The pipeline for the continuous development of artificial intelligence models—Current state of research and practice. *Journal of Systems and Software* 199 (2023). https://doi.org/10.1016/j.jss.2023.111615

[102] Michael Stonebraker and Lawrence A. Rowe. 1986. The design of POSTGRES. *ACM SIGMOD Record* 15, 2 (1986). https://doi.org/10.1145/16856.16888

[103] Nils Strassenburg, Dominic Kupfer, Julia Kowal, and Tilmann Rabl. 2023. Efficient Multi-Model Management. In *Proceedings of the International Conference on Extending Database Technology, (EDBT)*. https://doi.org/10.48786/edbt.2023.37

[104] Nils Strassenburg, Ilin Tolovski, and Tilmann Rabl. 2022. Efficiently Managing Deep Learning Models in a Distributed Environment. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. https://doi.org/10.48786/EDBT.2022.12

[105] Ashraf Tahmasbi, Ellango Jothimurugesan, Srikanta Tirthapura, and Phillip B. Gibbons. 2021. DriftSurf: Stable-State / Reactive-State Learning under Concept Drift. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[106] Tesla. 2019. Tesla Autonomy Day. https://www.youtube.com/watch?v=Ucp0TTmvqOE&t=6678s.

[107] Huangshi Tian, Minchen Yu, and Wei Wang. 2018. Continuum: A Platform for Cost-Aware, Low-Latency Continual Learning. In *Proceedings of the Symposium on Cloud Computing (SoCC)*. https://doi.org/10.1145/3267809.3267817

[108] Josh Tobin. 2021. Toward continual learning systems. https://gantry.io/blog/toward-continual-learning-systems/.

[109] Arnaud Van Looveren, Janis Klaise, Giovanni Vacanti, Oliver Cobb, Ashley Scillitoe, Robert Samoilescu, and Alex Athorne. 2019. Alibi Detect: Algorithms for outlier, adversarial and drift detection. https://github.com/SeldonIO/alibi-detect

[110] Daniel Vela, Andrew Sharp, Richard Zhang, Trang Nguyen, An Hoang, and Oleg S. Pianykh. 2022. Temporal quality degradation in AI models. *Scientific Reports* 12, 1 (2022). https://doi.org/10.1038/s41598-022-15245-z

[111] Zige Wang, Wanjun Zhong, Yufei Wang, Qi Zhu, Fei Mi, Baojun Wang, Lifeng Shang, Xin Jiang, and Qun Liu. 2023. Data Management For Large Language Models: A Survey. *arXiv preprint* (2023). https://doi.org/10.48550/ARXIV.2312.01700

[112] Alexander Warnecke, Lukas Pirch, Christian Wressnegger, and Konrad Rieck. 2023. Machine Unlearning of Features and Labels. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*.

[113] Elias Werner, Nishant Kumar, Matthias Lieber, Sunna Torge, Stefan Gumhold, and Wolfgang Nagel. 2024. Towards Computational Performance Engineering for Unsupervised Concept Drift Detection: Complexities, Benchmarking, Performance Analysis. In *Proceedings of the International Conference on Data Science, Technology and Applications (DATA)*. https://doi.org/10.5220/0012758600003756

[114] Tobias Weyand, Andre Araujo, Bingyi Cao, and Jack Sim. 2020. Google Landmarks Dataset v2 – A Large-Scale Benchmark for Instance-Level Recognition and Retrieval. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. https://doi.org/10.1109/cvpr42600.2020.00265

[115] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. https://doi.org/10.18653/v1/2020.emnlp-demos.6

[116] Xingjiao Wu, Luwei Xiao, Yixuan Sun, Junhang Zhang, Tianlong Ma, and Liang He. 2022. A survey of human-in-the-loop for machine learning. *Future Generation Computer Systems* 135 (2022). https://doi.org/10.1016/j.future.2022.05.014

[117] Yue Wu, Yinpeng Chen, Lijuan Wang, Yuancheng Ye, Zicheng Liu, Yandong Guo, and Yun Fu. 2019. Large Scale Incremental Learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. https://doi.org/10.1109/cvpr.2019.00046

[118] Yinjun Wu, Val Tannen, and Susan B. Davidson. 2020. PrIU: A Provenance-Based Approach for Incrementally Updating Regression Models. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. https://doi.org/10.1145/3318464.3380571

[119] Doris Xin, Hui Miao, Aditya Parameswaran, and Neoklis Polyzotis. 2021. Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. https://doi.org/10.1145/3448016.3457566

[120] Chen Yang, Jin Chen, Qian Yu, Xiangdong Wu, Kui Ma, Zihao Zhao, Zhiwei Fang, Wenlong Chen, Chaosheng Fan, Jie He, Changping Peng, Zhangang Lin, and Jingping Shao. 2023. An Incremental Update Framework for Online Recommenders with Data-Driven Prior. In *Proceedings of the International Conference on Information and Knowledge Management (CKIM)*. https://doi.org/10.1145/3583780.3615456

[121] Huaxiu Yao, Caroline Choi, Bochuan Cao, Yoonho Lee, Pang Wei Koh, and Chelsea Finn. 2022. Wild-Time: A Benchmark of in-the-Wild Distribution Shift over Time. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS) (Benchmark Track)*.

[122] Liheng Yuan, Heng Li, Beihao Xia, Cuiying Gao, Mingyue Liu, Wei Yuan, and Xinge You. 2022. Recent Advances in Concept Drift Adaptation Methods for Deep Learning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. https://doi.org/10.24963/ijcai.2022/788

[123] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65. https://doi.org/10.1145/2934664

[124] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. 2022. Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/3470496.3533044