MLOps

DevOps: Technical & management practices that aim to increase an organization's velocity in releasing high-quality software → CI, CD, monitoring, Infrastructure as code

MLOps: extension of DevOps to ML projects, set of tools and best practices for bringing ML into production

MLOps Features & Lifecycle

Design: Requirements Engineering (business objectives, ML use case prioritization, Data check)

Model dev: Data engineering (storage, versioning, labelling), Model engineering, Testing, Validation

Operations: Deployment, CI/CD pipelines, Monitoring (model/data drift, logging)

ML system design: Systematic approach to MLOps → considers an ML system holistically to ensure that all the components and their stakeholders can work together to satisfy the specified objectives

ML project lifecycle

1. **Planning & project setup:** requirements & goals, allocate resources, ethical implications
2. **Data collection & labelling:** training data, annotate with ground truth labels
3. **Training & debugging:** implement baseline, find SoTa model and reproduce, debug, improve model
4. **Deploying & testing:** setup production system, write tests, evaluate for biases, roll out

ML product archetypes

Software 2.0: take something traditional SW is good at and make it better with ML **Human in the loop:** complement humans with ML tools **Autonomous systems**

Infrastructure & Tooling

Distributed Training

Scenarios that require > 1 GPU: data batch / model might not fit on a single GPU

Data parallelism (split batches): distribute a single batch of data across GPUs, then average gradients across them (→ GPUs must have fast inter-connects)

Model parallelism: what takes up GPU memory? optimizer states (os), gradients (g), model params (p)

(1) Sharded Data Parallelism: ZeRO-3 → distribute p, g, os over the data-parallel GPU's
(2) Pipelined model-parallelism: Split model over several GPUs (DeepSpeed, FairScale) (3) Tensor parallelism: Matrix multiplication can be distributed over GPUs → 3D-parallelism combines these methods

Gradient checkpointing: saves strategically selected activations in the forward pass → only a fraction needs to be recomputed in the backward pass

FFCV: eliminate data bottlenecks (caching, data pre-loading, async data transfer)

Computation

GPU main factors: VRAM (amount of data fitting on GPU), Compute speed, Link speed (CPU-GPU)

Cloud vs On-prem: for high utilization, local HW is recommended (cheaper), for scaling up for peak usage, on-demand cloud is more flexible

Resource management

Typical workflow: (1) run an experiment (2) define machine + GPU, software setup, training dataset
→ possible solutions: SLURM, Docker + Kubernetes

Scheduler: when to run jobs
Orchestrator: where to run jobs
schedulers usually have orchestrating capacity and vice versa

Container: built packaged ENV **VM:** require the hypervisor to virtualize a HW stack

Experiment management

Tensorboard: solution for single projects **MLFlow:** OS SW for experiment and model management

Development & Troubleshooting

Strategy for DL troubleshooting (→ pessimistic)

! choose a metric to improve system works with a single metric (accuracy, precision, ...)

Start simple: choose simple baseline, use sensible defaults (adam optimizer, no regularization), normalize input (subtract mean and divide by std), simplify the problem (smaller dataset)

Implement & debug: (1) get your model to run: step through in a debugger → watch out for OOM, casting and shape (2) overfit a single batch: look for corrupted data, overregularization, broadcasting errs (3) Compare to a known result: keep iterating until model performs up to expectations

Evaluate: Bias-variance decomposition: breakdown of test error by src, test error = irreducible error + bias + variance + val. set overfitting. Distribution shift: use 2 val sets: one sampled from training distribution and one from test

Bias-variance with distribution shift: test error += distr. shift

High variance → overfitting; High bias → underfitting

Improve model/data: (1) underfitting: bigger model, reduce regularization, error analysis (2) overfitting: more training data, normalization (3) distribution shift: domain adaption techniques (4) re-balance dataset.

Tune hyperparameter: Tool: Weights & Biases. Grid search: try out all combinations of hyperparams → foreach combination train model & evaluate. Random search: same as grid search → often more efficient than grid search

Testing

Best practices: automate tests (CI/CD), test coverage
→ pytest, doctest (tests in docstrings), black, flake8 (linters)

Infrastructure Tests: Unit tests for training code. Goal: avoid bugs in training pipeline → add single batch or single epoch test, run frequently

Training tests: Integration tests between training and data system; Goal: make sure training is reproducible → pull fixed dataset (versioned) and run a full training run, check model performance consistency

Functionality tests: unit tests for prediction code; Goal: avoid regression in the code → load a pretrained model and test prediction on a few key examples

Evaluation tests: Integration tests between your training system and prediction; Goal: make sure model is ready to go into production → evaluate model on all metrics, datasets that you care about, slice-based evaluation, robustness tests: live data can contain noise, making it different from test → add noise to data

Shadow tests: integration tests between prediction and serving system → detect inconsistencies between offline and online model; How: run new model in prod, but don't return predictions to users

Labeling tests: Goal: catch poor quality labels before they corrupt your model; train and certify the labellers

Expectation tests: unit tests for data → great_expectations

Data Engineering

Data sources & storage

Building blocks: Filesystem, object storage, DB, warehouse, data lake

Object storage: API over the filesystem → S3, HDFS, Ceph

Databases: Persistent, fast, scalable

Warehouse: structured aggregation of data for analysis → ETL (extract, transform, load)

Lake: unstructured aggregation of data from multiple sources (databases, logs, ...) → ELT (extract, load, transform)

Data Processing

DAG (Directed Acyclic Graph)

Hadoop / Spark: Parallel, distributed big data processing

Airflow: Workflow management tool. Workflow as DAG

Feature Store (Tecton, Feast, FeatureForm)

ML-specific data system that: (1) runs data pipelines that transform raw data into feature values
(2) stores and manages the feature data itself
(3) serves feature data consistently for training and inference

Data exploration (Pandas)

Initial understanding of data before training a model that typically involves visual-profiling → check for anomalies or missing values

Data Versioning

Level 0: unversioned (filesystem)
L1: versioned via snapshot at training time
L2: versioned as a mix of assets & code (Git LFS)
L3: specialized data versioning solution (Oxen, DVC)

Data Privacy

Federated Learning: training a global model from data on a local device, without having access to the data
Differential Privacy: aggregating data such that individual points cannot be identified

Training data & Feature Engineering

Data Labeling

Cost-effective and high quality data labeling is key to successful model development. Labeling SW: Label Studio, Diffgram, Prodigy
Hand labeling: Expensive, non-private, slow, non-adaptive
Self-supervised learning: model is trained on a pretext task using the data itself to generate supervisory signals, rather than relying on external labels
Semi-supervised learning: small part of training data has labels, most data is unlabeled
Weakly supervised learning: small part of train data has labels, the remainder has "weak" labels, obtained from heuristics
Active learning: Learning Algo can interactively query a human to label new data points
Transfer learning: model designed for one task is reused on a different task

Sampling & Class imbalance

Convenience sampling: selection based on availability
Snowball sampling: future samples are selected based on existing samples
Judgement sampling: experts decide what to include
Quota sampling: quotas for certain slices of data
Stratified sampling: divide population by subgroups
Class imbalance: not enough signal to learn about rare classes → sampling biases, labeling errors
↳ Resampling: undersampling: remove samples from majority classes (overfitting); oversampling: add more samples to minority class (→ loss of information)

Data augmentation & Synthetic Data

Increase size and diversity of data without actually collecting new data → torchvision

Feature Engineering

Process of creating new features or modifying existing
Handling missing values: - remove column with too many missing values; - row deletion (bad when many examples have missing fields); - imputation (fill missing values)
Best practices: use features that generalize well

Deployment

Model Prediction

Batch Prediction: Periodically run model on new data and cache results in a DB.
Online Prediction: on-demand prediction
↳ Model-in-service: Web server loads model and calls it to make predictions
Model-as-service: model hosted on its own server → TorchServe, Tensorflow serving, MLServer

Deployment

Tools: Google Cloud, AzureML
Model store: save artifacts (params, data, dependencies) → MLFlow, Neptune, ClearML
Optimizations: model compression, pruning

Edge deployment (TFLite, CoreML, Pytorch mobile)

Edge computing: works without internet, don't worry about latency, fewer privacy concerns
Edge deployment: Send model weights to client device, client loads the model and interacts directly

Monitoring & Continual Learning

Data drift: data distribution changes → different users, different usage patterns; Instantaneous drift: model deployed in new domain, bug in pipeline; Gradual shift: user preferences change over time
Model shift "Concept drift": same input, expecting different output. Can be cyclic and seasonal.
What to monitor: model metrics, business metrics, model inputs & outputs, system performance
Measuring distribution change: ① select a reference window of "good" data ② select measurement window ③ compare using a distance metric → rule-based vs statistical (Alibi-detect)
Monitoring tools: NannyML, deepchecks, new relic

Retraining

→ when there is data/model drift or new data is available
Periodic retraining: ① Logging: log everything ② Curation: sample at random to give max data points ③ Trigger: retrain ④ dataset formation: use rolling window of data ⑤ offline testing, then online testing
On what data to retrain:
- retrain on all available data, sliding window, continual learning (→ finetune on new data)
↳ ⚠ catastrophic forgetting of previously learned

Continual Learning

Replay methods: maintain subset of samples from previous task and reuse as additional inputs in fut.
Regularization: add regularization term to loss func. consolidating previous knowledge when learning on new data

Foundation Models

Model that is trained on broad data at scale, is designed for generality of output → can be adapted to various downstream tasks

Large Language Model (LLM)

success recipe: tokenization, embedding creation, transformer
Tokenization: Text is split up → indices into vocabulary
Transformer architectures: ① Decoder only (GPT): generates tokens to continue given input ② Encoder only (BERT): learns text representation that can support various NLP tasks
Few-shot learning: using prompts that include instructions with a few examples, one can address almost any NLP task
Limitations: lack state/memory, stochastic/probabilistic, stale information, huge, hallucination

LLMOps

RAG (Retrieval Augmented Generation): Enhance LLM by giving them access to information from DB

ML Roles and Teams

Economics: AI reduces cost of prediction → use ML when impact is high and feasibility is high
Cost drivers: data availability, model accuracy, difficulty