THE BIG DICTIONARY OF MLOPS

HOPSWORKS

www.hopsworks.ai/mlops-dictionary

This dictionary/glossary covers terms from **MLOps, data engineering, and feature stores**, but does not cover terms from the broader ML (Machine Learning) algorithms and frameworks space. MLOps is the roadmap you follow to go from training models in notebooks to building production ML systems. MLOps is a set of principles and practices that encompass the entire ML System lifecycle, from ideation to data management, feature creation, model training, inference, observability, and operations. **MLOps is based on three principles: observability, automated testing, and versioning of ML artifacts.**

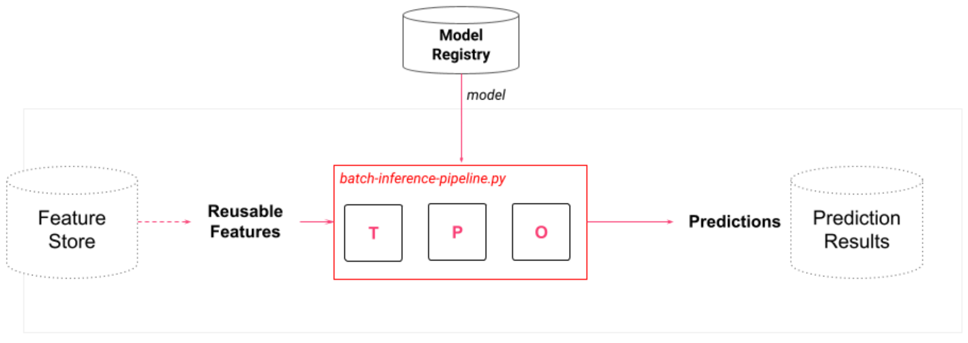
This version of the dictionary highlights the **25 most read and searched terms** since it was first published one year ago. The entries in this edition reflect current trends in the world of AI and MLOps and give us an idea of the direction in which the industry is leaning.

INDEX

B

- ### *Batch Inference Pipeline*

What is a batch inference pipeline?
A batch inference pipeline is a program that takes as input a batch of data and a model, and outputs predictions that are typically written to some sink, such as a database.



TPO = Transform features, Predict, Output.

Why are batch inference pipelines important?
Batch inference pipelines are important because they allow for efficient and scalable inference on large volumes of data using a trained model. Batch inference pipelines are typically run on a schedule (e.g., daily or hourly) and are used to drive dashboards and operational ML systems (that use the predictions for intelligent services).

Example of a batch inference application
A Spark program reads all of the new inference data that has arrived in the previous 24 hours as a DataFrame. The Spark program downloads the model from the model registry, broadcasts it to all executors, and then a map function calls predict on the model for each row in the DataFrame, returning the predictions as a DataFrame. The predictions DataFrame is then stored in a database from where it is consumed by a Dashboard or operational ML system.

C

- ### *Context Window for LLMs*

What is a context window for LLMs?
The context window of LLMs is the number of tokens the model can take as input when generating responses. For example, in GPT-3 the context window size is 2K (2000) and in GPT-4 it is a larger 32K. There is a trend and demand for increasingly larger context window sizes in LLMs. Larger context windows improve LLM performance and their usefulness across various applications.

Why is a large context window size important?
Larger context window sizes increase the ability to perform in-context learning in prompts. That is, you can provide more examples and/or larger examples as prompt inputs, enabling the LLM to give you a better answer. For example, a LLM could take an entire document as input, helping with comprehension of the full scope of an article. This ability enables LLM to produce more contextually relevant responses by leveraging a more comprehensive understanding of the input.

Another example would be providing the LLM with context information that was not available at the time the LLM was trained - if a user asked GPT-4 “who won the world cup in 2022?”, it doesn’t know the answer (it’s cut-off was before that date). However, a client can use the query to find a relevant document about the 2022 world cup (e.g., using similarity search in a vector database) and add that document to the prompt (e.g., it might be the wikipedia article on the 2022 world cup). The LLM can now answer the query (“Argentina” is the correct answer) as the answer was in the document included in the prompt.

Challenges in increasing the context window size
The costs increase of larger context windows appear to increase quadratically as the number of tokens is increased, e.g., going from 2K to 4K with GPT-3 to GPT-3.5 was not twice as computationally expensive, but 4x the computationally cost. **Research is ongoing on decreasing the cost**.

Another challenge is that, based on **Liu et al**, it appears that adding relevant context at the beginning or the end of a prompt improves the performance of LLMs, compared to including the relevant context in the middle of the prompt. It is unclear how this observation will affect larger context windows.

F

- ### *Feature*

What are features in Machine Learning?
A feature is a measurable property of some data-sample that is used as input for a ML model for training and serving. A feature should have predictive power for the model it is being used in.

What questions do I need to ask about whether it is ok to use a particular feature in my model or not?
Predictive power is a necessary but not a sufficient condition for including a feature in a model. The feature

- should have predictive power for your model,
- should be feasible for use in the model (i.e., you are able to compute the feature and use it when needed - online or offline),
- should not be redundant (e.g., highly correlated with an existing feature),
- should not be cost-prohibitive (i.e., using the feature means the model will not generate a ROI), and
- should not be prohibited from use or unethical to use.

How important is it to select or create good features?
Features matter because they directly impact the accuracy and performance of machine learning models. Choosing the right set of features is critical for building effective models, and feature engineering (now often called data-centric AI) is an iterative process of adding and removing features to find the best model given the available data and the available resources for training and inference.

Example of features
In a model that tries to predict fraud for credit card transactions, the features might include the transaction amount and location for the current transactions as well as the number and location of transactions in recent windows of time (the last 5 minutes, 30 minutes, 1 hour, 6 hours). These features can help the model identify patterns such as chain attacks and geographic attacks that are indicative of fraudulent behavior.

- ### *Feature Pipeline*

What is a feature pipeline in machine learning?
A feature pipeline is a program that orchestrates the execution of a dataflow graph of feature functions (transformations on input data to create unencoded feature data), where the computed features are written to one or more feature groups. A feature pipeline can also include reading the input data from data sources, data validation, and any other steps needed when computing features.

Why do I need a feature pipeline?
Feature pipelines are needed to enable features to be computed on a schedule, or if they are streaming feature pipelines, run 24x7. The feature pipeline encapsulates the logic for computing features in feature groups, defines the data validation logic, and writes the features to feature groups. A batch feature pipeline needs to be run on a schedule by an orchestration engine, such as Airflow, Dagster, or, for simple cron-based scheduling, Modal.

What are the data sources for feature pipelines?
Feature pipelines read their input data from data sources such as data warehouses, message buses, databases, object stores, and Http APIs. The data sources can provide input live data, during scheduled executions, or historical data when backfilling feature groups. Feature pipelines should scale to handle the largest expected input volume size.

An example of the steps in feature pipeline might include:

1. **Data ingestion**: Raw data is read from various data sources for processing.
2. **Data/feature validation**: The raw data and/or feature data is validated to ensure that it is accurate, complete, and consistent.
3. **Feature extraction/transformation**: Relevant features are extracted from the raw data and transformed into a format that is optimized for machine learning models using techniques such as filtering, aggregation, dimensionality reduction (embeddings, PCA), binning, and feature crossing.
4. **Feature storage**: The features are stored in feature groups in the feature store for access training and inference pipelines.

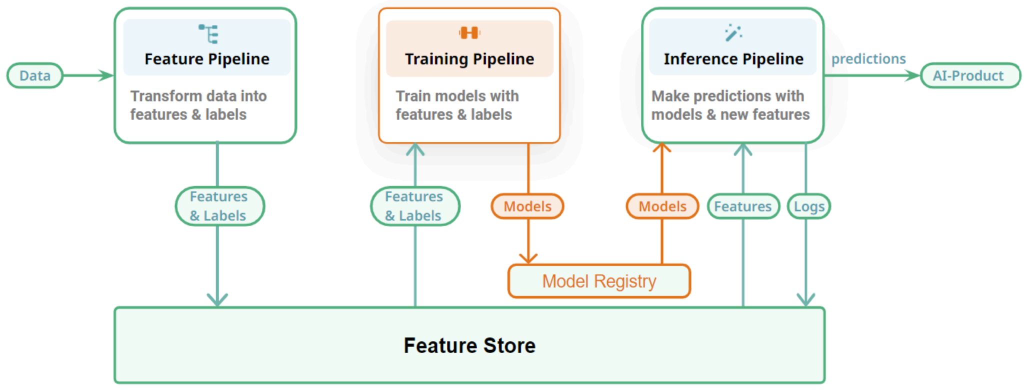
● *Feature Store*

What is a feature store?
A feature store is a data platform that supports the development and operation of machine learning systems by managing the storage and efficient querying of feature data. Machine learning systems can be real-time, batch or stream processing systems, and the feature store is a general purpose data platform that supports a multitude of write and read workloads, including batch and streaming writes, to batch and point read queries, and even approximate nearest neighbor search. Feature stores also provide compute support to ML pipelines that create and use features, including ensuring the consistent computation of features in different (offline and online) ML pipelines.

What is a feature and why do I need a specialized store for them?
A feature is a measure property of some entity that has predictive power for a machine learning model. Feature data is used to train ML models, and make predictions in batch ML systems and online ML systems. Features can be computed either when they are needed or in advance and used later for training and inference. Some of the advantages of storing features is that they can be easily discovered and reused in different models, reducing the cost and time required to build new machine learning

systems. For **real-time ML** systems, the feature store provides history and context to (stateless) online models. Online models tend to have no local state, but the feature store can enrich the set of features available to the model by providing, for example, historical feature data about users (retrieved with the user's ID) as well as contextual data, such as what's trending. The feature store also reduces the time required to make online predictions, as these features do not need to be computed on-demand, - they are precomputed.

How does the feature store relate to MLOps and ML systems?



In a MLOps platform, the feature store is the glue that ties together different ML pipelines to make a complete ML system:

- feature pipelines compute features and then write those features (and labels/targets) to it;
- training pipelines read features (and labels/targets) from it;
- Inference pipelines can read precomputed features from it.

The main goals of MLOps are to decrease model iteration time, improve **model performance**, ensure governance of **ML assets** (feature, models), and improve collaboration. By decomposing your ML system into separate **feature, training, and inference (FTI) pipelines**, your system will be more modular with 3 pipelines that can be independently developed, tested, and operated. This architecture will scale from one developer to teams that take responsibility for the different ML pipelines: data engineers and data scientists typically build and operate feature pipelines; data scientists build and operate training pipelines, while ML engineers build and operate inference pipelines. The feature store enables the FTI pipeline architecture, enabling improved communication within and between data, ML, and operations teams.

What problems does a feature store solve?
The feature store solves many of the challenges that you typically face when you (1) deploy models to production, and (2) scale the number of models you deploy to production, and (3) scale the size of your ML teams, including:

- Support for **collaborative development** of ML systems based on centralized, governed access to feature data, along with a new unified architecture for ML systems as feature, training and inference pipelines;
- **Manage incremental datasets** of feature data. You should be able to easily add new, update existing, and delete feature data using DataFrames. Feature data should be transparently and consistently replicated between the offline and online stores;
- **Backfill feature data** from data sources using a feature pipeline and **backfill training data** using a training pipeline;
- Provides **history and context** to stateless interactive (online) ML applications;
- **Feature reuse** is made easy by enabling developers to select existing features and reuse them for training and inference in a ML model;
- Support for **diverse feature computation frameworks - including batch, streaming, and request-time computation**. This enables ML systems to be built based on their **feature freshness** requirements;
- **Validate feature data written** and **monitor new feature data for drift**;
- A **taxonomy for data transformations for machine learning** based on the type of feature computed (a) reusable features are computed by model-independent transformations, (b) features specific to one model are computed by model-dependent transformations, and (c) features computed with request-time data are on-demand transformations. The feature store provides abstractions to prevent skew between data transformations performed in more than one ML pipeline.
- A **point-in-time consistent query engine** to create training data from historical time-series feature data, potentially spread over many tables, without future **data leakage;**
- A **query engine to retrieve and join precomputed features at low latency** for online inference using an entity key;
- A **query engine to find similar feature values using embedding vectors**.

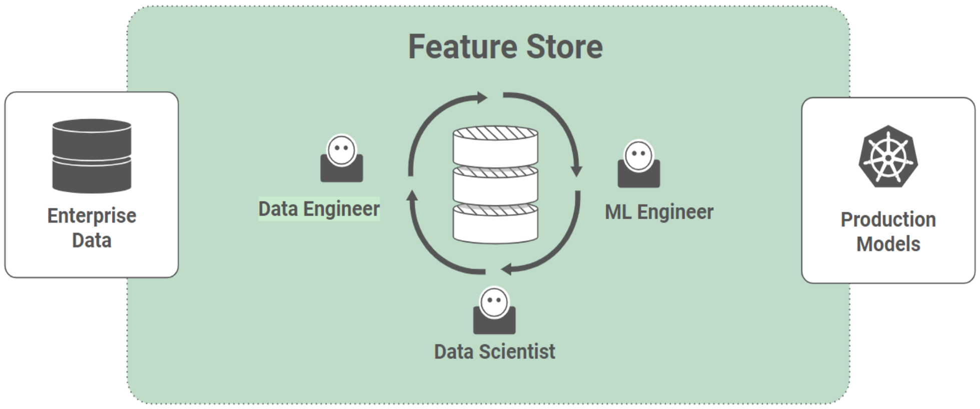
The table below shows you how the feature store can help you with common ML deployment scenarios.

	Batch ML	Real-Time ML	Enterprise ML at Scale
Collaborative Development	❎	❎	✅
Incremental Datasets	✅	✅	✅
Backfill feature data and training data	✅	✅	✅
Point-in-Time Consistent Training Data	✅	✅	✅
History and Context for Online Models	❎	✅	✅
Feature Reuse	❎	❎	✅
Feature validation/monitoring	✅	✅	✅
Multiple Computation Frameworks	❎	✅	✅
Taxonomy of Data Transformations	✅	✅	✅
Low latency retrieval of precomputed features	❎	✅	✅
Similarity search	❎	✅	✅

For just putting ML in production, the feature store helps with managing incremental datasets, feature validation and monitoring, where to perform data transformations, and how to create point-in-time consistent training data. Real-Time ML extends the production ML scenario with the need for history and context information for stateless online models, low latency retrieval of precomputed features, online similarity search, and the need for either stream processing or on-demand feature computation. For the ML at large scale, there is also the challenge of enabling collaboration between teams of data engineers, data scientists, and ML engineers, as well as the reuse of features in many models.

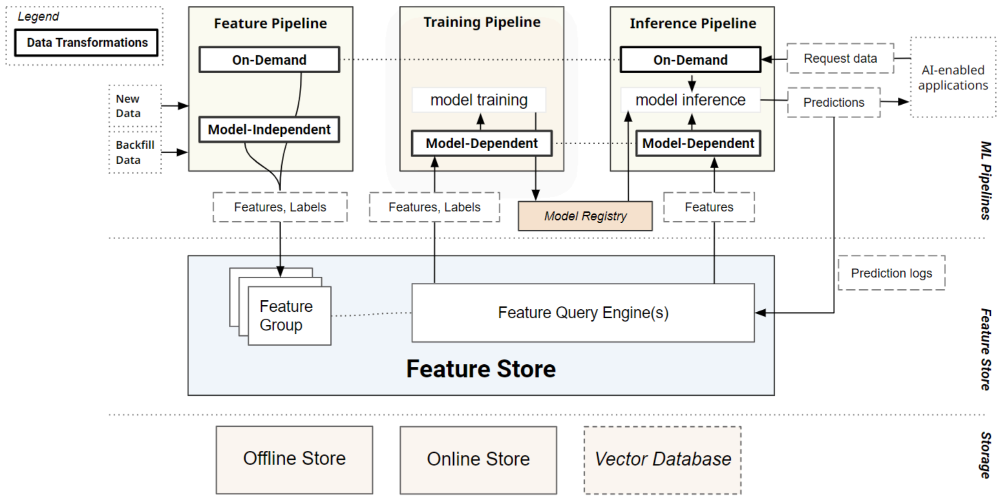
Collaborative Development
Feature stores are the key data layer in a MLOps platform. The main goals of MLOps are to decrease model iteration time, improve model performance, ensure governance of ML assets (feature, models), and improve collaboration. The feature store enables different teams to take responsibility for the different ML pipelines: data engineers and data scientists typically build and operate feature pipelines; data scientists build and operate training pipelines, while ML engineers build and operate inference pipelines.

Cross-Team Collaboration with a Feature Store



They enable the sharing of ML assets and improved communication within and between teams. Whether teams are building batch machine learning systems or real-time machine learning systems, they can use shared language around feature, training, and inference pipelines to describe their responsibilities and interfaces.

A more detailed Feature Store Architecture is shown in the figure below.



Its historical feature data is stored in an offline store (typically a columnar data store), its most recent feature data that is used by online models in an online store (typically a row-oriented database or key-value store), and if indexed embeddings are supported, they are stored in a vector database. Some feature stores provide the storage layer as part of the platform, some have partial or full pluggable storage layers.

The machine learning pipelines (feature pipelines, training pipelines, and inference pipelines) read and write features/labels from/to the feature store, and prediction logs are typically also stored there to support feature/model monitoring and debugging. Different data transformations (model-independent, model-dependent, and on-demand) are performed in the different ML pipelines, see the Taxonomy of Data Transformations for more details.

Incremental Datasets

Feature pipelines keep producing feature data as long as your ML system is running. Without a feature store, it is non-trivial to manage the mutable datasets updated by feature pipelines - as the datasets are stored in the different offline/online/vector-db stores. Each store has its own drivers, authentication and authorization support, and the synchronization of updates across all stores is challenging.

Feature stores make the management of mutable datasets of features, called feature groups, easy by providing CRUD (create/read/update/delete) APIs. The following code snippet shows how to append, update & delete feature data in a feature group using a Pandas DataFrame in Hopsworks. The updates are transparently synchronized across all of the underlying stores - the offline/online/vector-db stores.

Python
```python
df = # read from data source, then perform feature
engineering
fg =
fs.get_or_create_feature_group(name="query_terms_yearly",
                              version=1,
                              description="Count of search
term by year",
                              primary_key=['year',
'search_term'],
                              partition_key=['year'],
                              online_enabled=True
                              )
fg.insert(df) # insert or update
fg.commit_delete_record(df) # delete
```

We can also update the same feature group using a stream processing client (streaming feature pipeline). The following code snippet uses PySpark streaming to update a feature group in Hopsworks. It computes the average amount of money spent on a credit card, for all transactions on the credit card, every 10 minutes. It reads its input data as events from a Kafka cluster.

Python

```
df_read =
spark.readStream.format("kafka")...option("subscribe",
KAFKA_TOPIC_NAME).load()

# Deserialize data from Kafka and create streaming query
df_deser = df_read.selectExpr(....).select(...)

# 10 minute window
windowed10mSignalDF = df_deser \
    .selectExpr(...)\
    .withWatermark(...) \
    .groupBy(window("datetime", "10 minutes"),
"cc_num").agg(avg("amount")) \
    .select(...)

card_transactions_10m_agg
=fs.get_feature_group("card_transactions_10m_agg",
version=1)
query_10m =
card_transactions_10m_agg.insert_stream(window10mSignalDF)
```

Some feature stores also support defining columns as embeddings that are indexed for similarity search. The following code snippet writes a DataFrame to a feature group in Hopsworks, and indexes the “embedding_body” column in the vector database. You need to create the vector embedding using a model, add it as a column to the DataFrame, and then write the DataFrame to Hopsworks.

Python

```
from hsfs import embedding
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('all-MiniLM-L6-v2')

df = # read from data source, then perform feature
engineering
```

```
embeddings_body = model.encode(df["Article"])
df["embedding_body"] = pd.Series(embeddings_body.tolist())

emb = embedding.EmbeddingIndex()
emb.add_embedding("embedding_body",
len(df["embedding_body"][0]))

news_fg = fs.get_or_create_feature_group(
    name="news_fg",
    embedding_index=emb,
    primary_key=["id"],
    version=1,
    online_enabled=True
)
news_fg.insert(df)
```

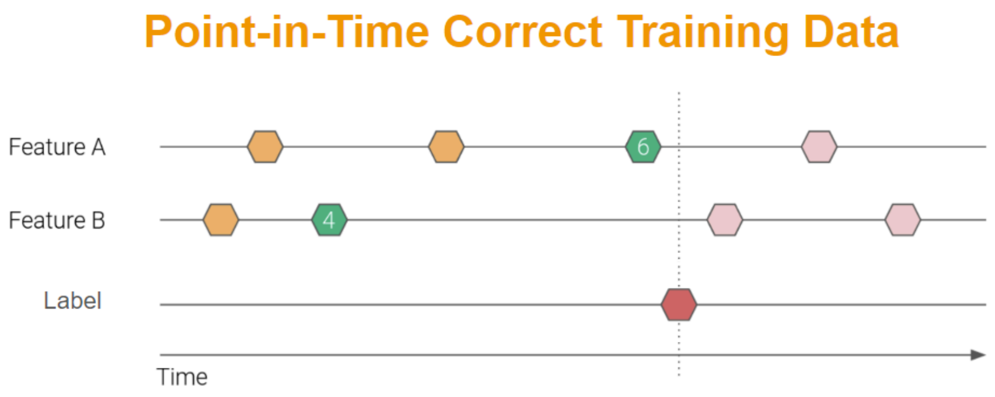
Backfill feature data and Training Data
Backfilling is the process of recomputing datasets from raw, historical data. When you backfill feature data, backfilling involves running a feature pipeline with historical data to populate the feature store. This requires users to provide a start_time and an end_time for the range of data that is to be backfilled, and the data source needs to support timestamps, e.g., Type 2 slowly changing dimensions in a data warehouse table.

The same feature pipeline used to backfill features should also process “live” data. You just point the feature pipeline at the data source and the range of data to backfill (e.g., backfill the daily partitions with all users for the last 180 days). Both batch and streaming feature pipelines should be able to backfill features. Backfilling features is important because you may have existing historical data that can be leveraged to create training data for a model. If you couldn’t backfill features, you could start logging features in your production system and wait until sufficient data has been collected before you start training your model.

Point-in-Time Correct Training Data
If you want to create training data from time-series feature data without any future data leakage, you will need to perform a temporal join, sometimes called a point-in-time correct join.

For example, in the figure below, we can see that for the (red) label value, the correct feature values for Feature A and Feature B are 4 and 6, respectively. Data leakage would occur if we included feature values that are either the pink (future data leakage) or orange values (stale feature data). If you do not create point-in-time correct training data, your model may perform poorly and it will be very difficult to discover the root cause of the poor performance.

If your offline store supports AsOf Joins, feature retrieval involves joining Feature A and Feature B from their respective tables AsOf the timestamp value for each row in the Label table. The SQL query to create training data is an “AS OF LEFT JOIN”, as this query enforces the invariant that for every row in your Label table, there should be a row in your training dataset, and if there are missing feature values for a join, we should include NULL values (we can later impute missing values in model-dependent transformations). If your offline store does not support AsOf Joins, you can write alternative windowing code using state tables.



As both AsOf Left joins and window tables result in complex SQL queries, many feature stores provide domain-specific language (DSL) support for executing the temporal query. The following code snippet, in Hopsworks, creates point-in-time-consistent training data by first creating a feature view. The code starts by (1) selecting the columns to use as features and label(s) to use for the model, then (2) creates a feature view with the selected columns, defining the label column(s), and (3) uses the feature view object to create a point-in-time correct snapshot of training data.

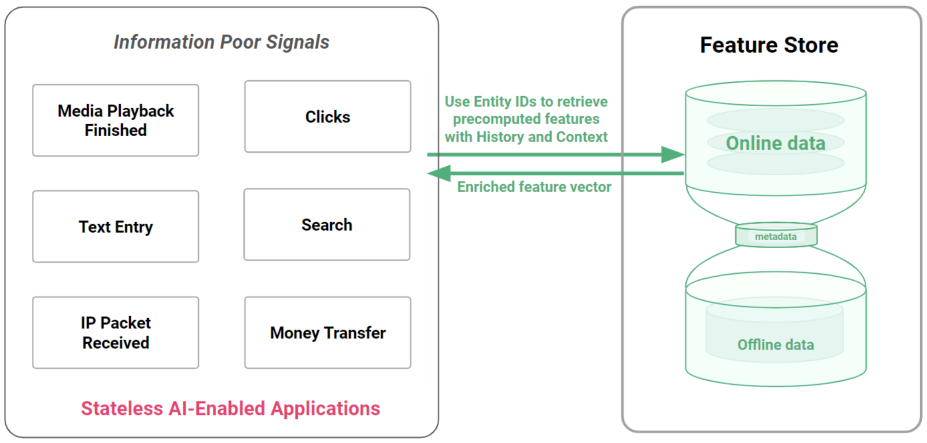
Python
```python
fg_loans = fs.get_feature_group(name="loans", version=1)
fg_applicants = fs.get_feature_group(name="applicants",
version=1)
select= fg_loans.select_except(["issue_d", "id"]).join(\
            fg_applicants.select_except(["earliest_cr_line",
"id"]))
```

```
fv = fs.create_feature_view(name="loans_approvals",
            version=1,
            description="Loan applicant data",
            labels=["loan_status"],
            query=select
            )
X_train, X_test, y_train, y_test =
fv.train_test_split(test_size=0.2)
#....
model.fit(X_train, y_train)
```

The following code snippet, in Hopsworks, uses the feature view we just defined to create point-in-time consistent batch inference data. The model makes predictions using the DataFrame *df* containing the batch inference data.

Python
```
fv = fs.get_feature_view(name="loans_approvals",
version=fv_version)
df = fv.get_batch_data(start_time="2023-12-23 00:00",
end_time=NOW)

predictions_df = model.predict(df)
```

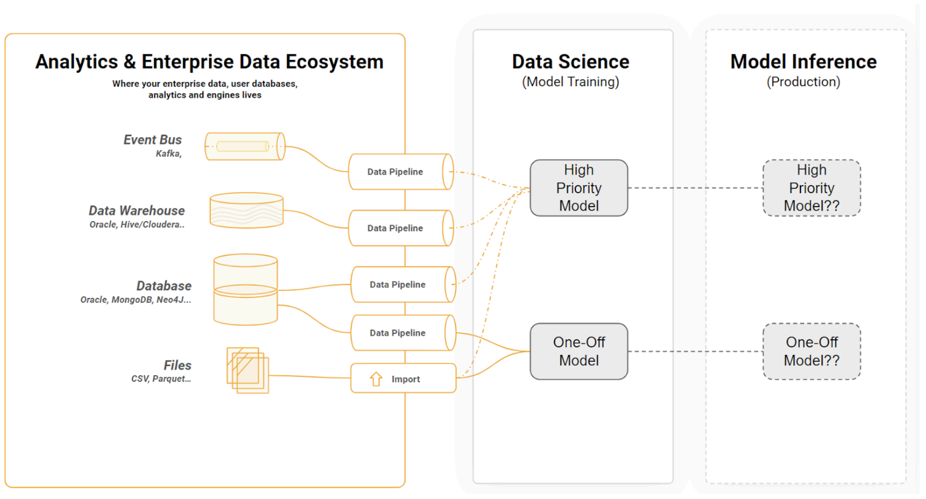
History and Context for Online Models



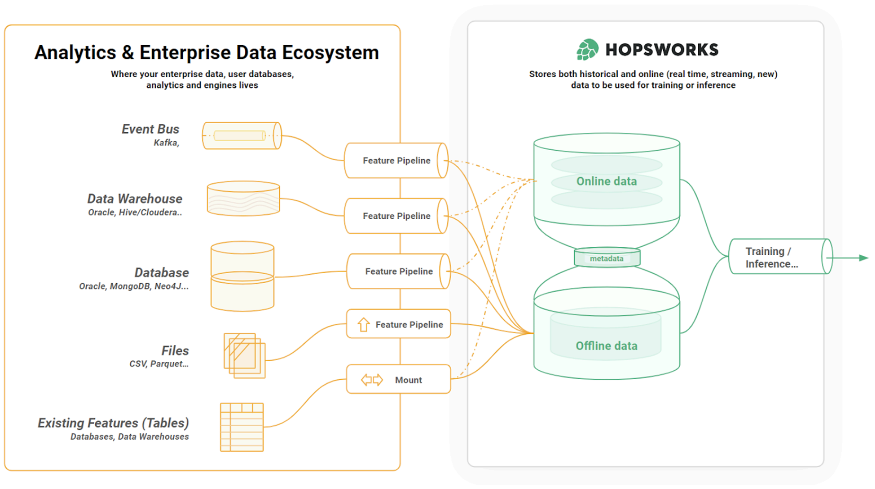
Online models are often hosted in model-serving infrastructure or stateless (AI-enabled) applications. In many user-facing applications, the actions taken by users are “information poor”, but we would still like to use a trained model to make an intelligent decision. For example, in Tiktok, a user click contains a limited amount of information - you could not build the world’s best real-time recommendation system using just a single user click as an input feature.

The solution is to use the user’s ID to retrieve precomputed features from the online store containing the user's personal history as well as context features (such as what videos or searches are trending). The precomputed features returned enrich any features that can be computed from the user input to build a rich feature vector that can be used to train complex ML models. For example, in Tiktok, you can retrieve precomputed features about the 10 most recent videos you looked at - their category, how long you engaged for, what’s trending, what your friends are looking at, and so on.
In many examples of online models, the entity is a simple user or product or booking. However, often you will need more complex data models, and it is beneficial if your online store supports multi-part primary keys (see Uber talk).

Feature Reuse
A common problem faced by organizations when they build their first ML models is that there is a lot of bespoke tooling, extracting data from existing backend systems so that it can be used to train a ML model. Then, when it comes to productionizing the ML model, more data pipelines are needed to continually extract new data and compute features so that the model can make continual predictions on the new feature data.

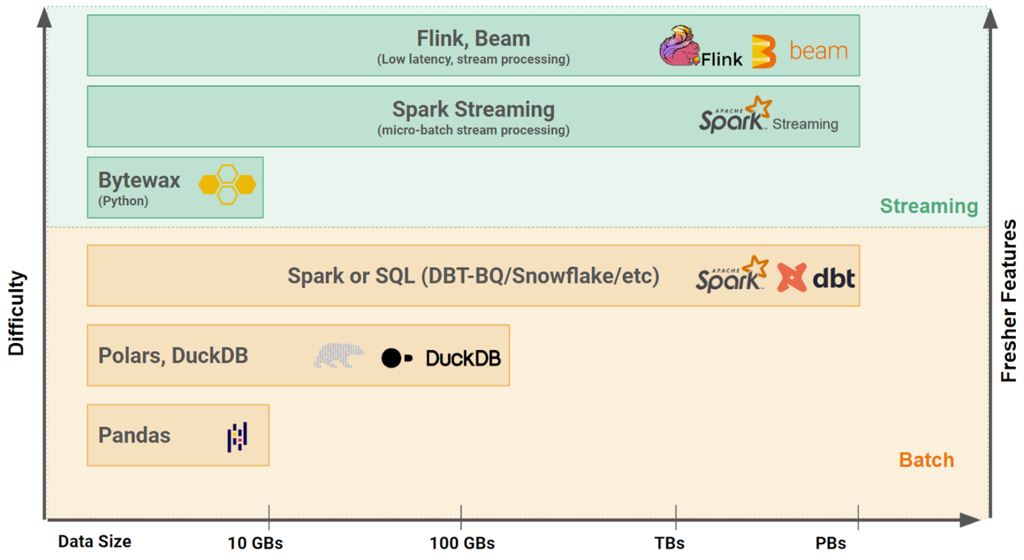


However, after the first set of pipelines have been written for the first model, organizations soon notice that one or more features used in an earlier model are needed in a new model. Meta reported that in their feature store “most features are used by many models”, and that the most popular 100 features are reused in over 100 different models. However, for expediency, developers typically rewrite the data pipelines for the new model. Now you have different models re-computing the same feature(s) with different pipelines. This leads to waste, and a less maintainable (non-DRY) code base.


Analytics & Enterprise Data Ecosystem
Where your enterprise data, user databases, analytics and engines lives
Event Bus
Kafka,
Data Warehouse
Oracle, Hive/Cloudera..
Database
Oracle, MongoDB, Neo4J...
Files
CSV, Parquet...
Existing Features (Tables)
Databases, Data Warehouses
Feature Pipeline
Feature Pipeline
Feature Pipeline
Feature Pipeline
Mount
HOPSWORKS
Stores both historical and online (real time, streaming, new) data to be used for training or inference
Online data
metadata
Offline data
Training / Inference...

The benefits of feature reuse with a feature store include higher quality features through increased usage and scrutiny, reduced storage costs - and less feature pipelines. In fact, the feature store decouples the number of models you run in production from the number of feature pipelines you have to maintain. Without a feature store, you typically write at least one feature pipeline per model. With a (large enough) feature store, you may not need to write any feature pipeline for your model if the features you need are already available there.

Multiple Feature Computation Models



The feature pipeline typically does not need GPUs, may be a batch program or streaming program, and may process small amounts of data with Pandas or Polars or large amounts of data with a framework such as Spark or DBT/SQL. Streaming feature pipelines can be implemented in Python (Bytewax) or more commonly in distributed frameworks such as PySpark, with its micro-batch computation model, or Flink/Beam with their lower latency per-event computation model.

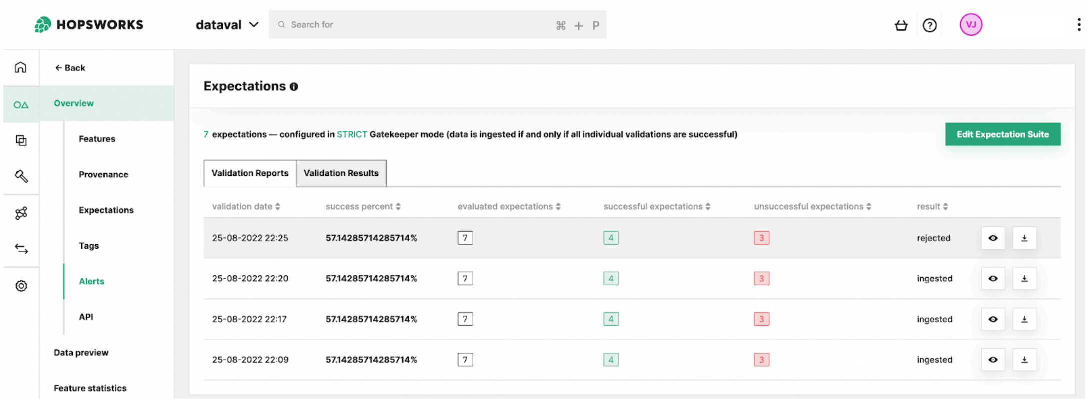
The training pipeline is typically a Python program, as most ML frameworks are written in Python. It reads features and labels as input, trains a model and outputs the trained model (typically to a model registry).

An inference pipeline then downloads a trained model and reads features as input (some may be computed from the user's request, but most will be read as precomputed features from the feature store). Finally, it uses the features as input to the model to make predictions that are either returned to the client who requested them or stored in some data store (often called an inference store) for later retrieval.

Validate Feature Data and Monitor for Drift
Garbage-in, garbage out is a well known adage in the data world. Feature stores can provide support for validating feature data in feature pipelines. The following code snippet uses the Great Expectations library to define a data validation rule that is applied when feature data is written to a feature group in Hopsworks.

Python
```
df = # read from data source, then perform feature
engineering


# define data validation rules in Great Expectations
ge_suite = ge.core.ExpectationSuite(
    expectation_suite_name="expectation_suite_101"
    )

ge_suite.add_expectation(
    ExpectationConfiguration(

expectation_type="expect_column_values_to_not_be_null",
        kwargs={"column":"'search_term'"}
    )
)

fg =
fs.get_or_create_feature_group(name="query_terms_yearly",
                              version=1,
                              description="Count of search
term by year",
                              primary_key=['year',
'search_term'],
                              partition_key=['year'],
                              online_enabled=True,
                              expectation_suite=ge_suite
                              )
fg.insert(df) # data validation rules executed in client
before insertion
```

The data validation results can then be viewed in the feature store, as shown below. In Hopsworks, you can trigger alerts if data validation fails, and you can decide whether to allow the insertion or fail the insertion of data, if data validation fails.



Expectations

7 expectations — configured in STRICT Gatekeeper mode (data is ingested if and only if all individual validations are successful)

validation date	success percent	evaluated expectations	successful expectations	unsuccessful expectations	result
25-08-2022 22:25	57.142857142857 14%	7	4	3	rejected
25-08-2022 22:20	57.142857142857 14%	7	4	3	ingested
25-08-2022 22:17	57.142857142857 14%	7	4	3	ingested
25-08-2022 22:09	57.142857142857 14%	7	4	3	ingested

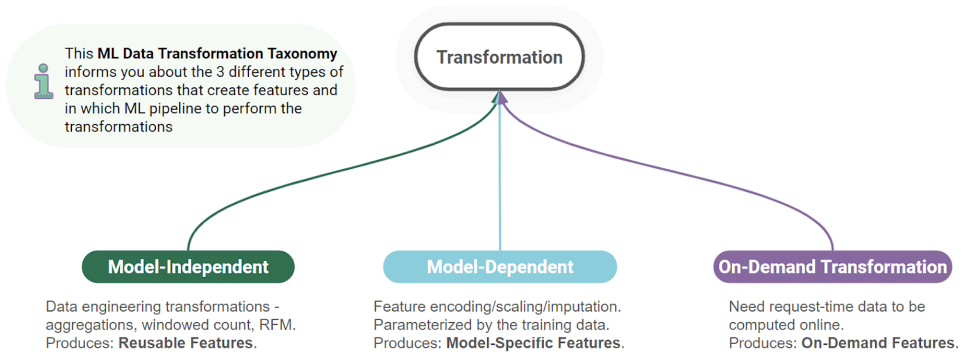
Feature monitoring is another useful capability provided by many feature stores. Whether you build a batch ML system or an online ML system, you should be able to monitor inference data for the system's model to see if it is statistically significantly different from the model's training data (*data drift*). If it is, you should alert users and ideally kick-off the retraining of the model using more recent training data.

Here is an example code snippet from Hopsworks for defining a feature monitoring rule for the feature "amount" in the model's prediction log (available for both batch and online ML systems). A job is run once per day to compare inference data for the last week for the *amount* feature, and if its mean value deviates more than 50% from the mean observed in the model's training data, *data drift* is flagged and alerts are triggered.

```python
# Compute statistics on a prediction log as a detection
window
fg_mon = pred_log.create_feature_monitoring("name",
    feature_name = "amount", job_frequency = "DAILY")
    .with_detection_window(row_percentage=0.8, time_offset
="1w")

# Compare feature statistics with a reference window - e.g.,
training data
fg_mon.with_reference_training_dataset(version=1).compare_on
(
    metric = "mean", threshold=50)
```

Taxonomy of Data Transformations

When data scientists and data engineers talk about data transformations, they are not talking about the same thing. This can cause problems in communication, but also in the bigger problem of feature reuse in feature stores. There are 3 different types of data transformations, and they belong in different ML pipelines.


This ML Data Transformation Taxonomy informs you about the 3 different types of transformations that create features and in which ML pipeline to perform the transformations
Transformation
Model-Independent
Model-Dependent
On-Demand Transformation
Data engineering transformations - aggregations, windowed count, RFM. Produces: Reusable Features.
Feature encoding/scaling/imputation. Parameterized by the training data. Produces: Model-Specific Features.
Need request-time data to be computed online. Produces: On-Demand Features.

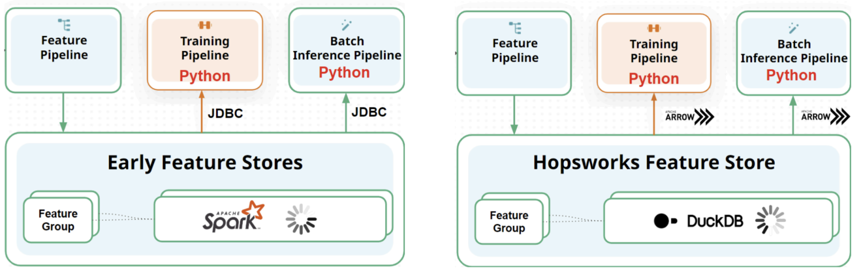
Data transformations, as understood by data engineers, is a catch-all term that covers data cleansing, aggregations, and any changes to your data to make it consumable by BI or ML. These data transformations are called model-independent transformations as they **produce features that are reusable by many models**.

In data science, data transformations are a more specific term that refers to encoding a variable (categorical or numerical) into a numerical format, scaling a numerical variable, or imputing a value for a variable, with the goal of improving the performance of your ML model training. These data transformations are called model-dependent transformations and they are **specific to one model**.

Finally, there are data transformations that can only be performed at runtime for online models as they require parameters only available in the prediction request. These data transformations are called on-demand transformations, but they may *also be needed in feature pipelines* if you want to backfill feature data from historical data.

The feature store architecture diagram from earlier shows that model-independent transformations are only performed in feature pipelines (whether batch or streaming pipelines). However, model-dependent transformations are performed in both training and inference pipelines, and on-demand transformations can be applied in both feature and online inference pipelines. You need to ensure that equivalent transformations are performed in both pipelines - if there is skew between the transformations, you will have model performance bugs that will be very hard to identify and debug. Feature stores help prevent this problem of online-offline skew. For example, model-dependent transformations can be performed in scikit-learn pipelines or in feature views in Hopsworks, ensuring consistent transformations in both training and inference pipelines. Similarly, on-demand transformations are version-controlled Python or Pandas user-defined functions (UDFs) in Hopsworks that are applied in both feature and online inference pipelines.

Query Engine for Point-in-Time Consistent Feature Data for Training
Feature stores can use existing columnar data stores and data processing engines, such as Spark, to create point-in-time correct training data. However, as of December 2023, Spark, BigQuery, Snowflake, and Redshift do not support the ASOF LEFT JOIN query that is used to create training data from feature groups. Instead, they have to implement stateful windowed approaches.



The other main performance bottleneck with many current data warehouses is that they provide query interfaces to Python with either a JDBC or ODBC API. These are row-oriented protocols, and data from the offline store needs to be pivoted from columnar format to row-oriented, and then back to column-oriented in Pandas. Arrow is now the backing data format for Pandas 2.+.

In open-source, reproducible benchmarks by KTH, Karolinska, and Hopsworks, they showed the throughput improvements over a specialist DuckDB/ArrowFlight feature query engine that returns Pandas DataFrames to Python clients in training and batch inference pipelines. We can see from the table below that throughput improvements of 10-45X JDBC/ODBC-based query engines can be achieved.

Pandas Read (rows/secs)	5m rows	10m rows	20m rows	50m rows
Hopsworks	2,314,815	2,155,172	2,724,796	3,313,453
Sagemaker	155,328	170,358	167,364	202,053
Vertex	38,011	54,672	77,289	172,247
Databricks*	85,807	27,666	*	*

*Databricks failed at 20m, 50m rows

Query Engine for Low Latency Feature Data for Online Inference

The online feature store is typically built on existing low latency row-oriented data stores. These could be key-value stores such as Redis or Dynamo or a key-value store with a SQL API, such as RonDB for Hopsworks.

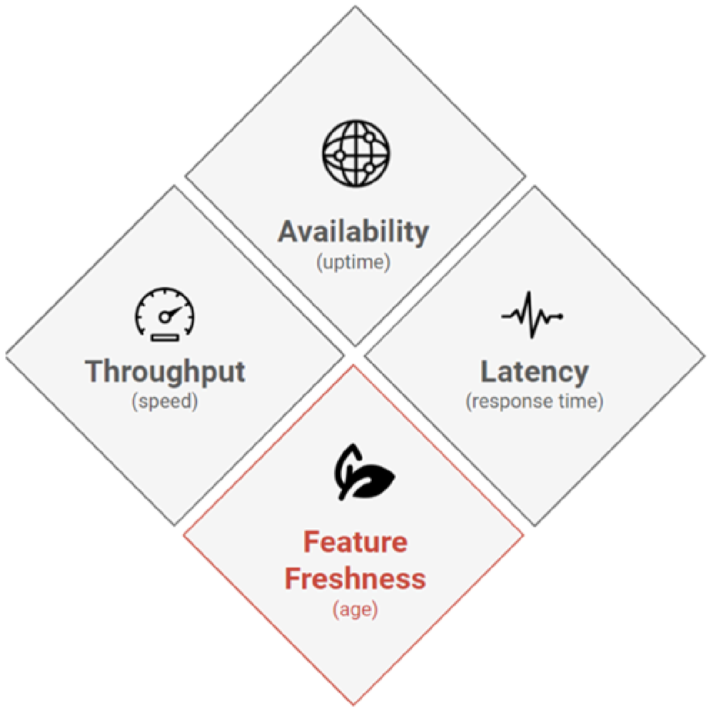
The process of building the feature vector for an online model also involves more than just retrieving precomputed features from the online feature store using an entity ID. Some features may be passed as request parameters directly and some features may be computed on-demand - using either request parameters or data from some 3rd party API, only available at runtime. These on-demand transformations may even need historical feature values, inference helper columns, to be computed.

In the code snippet below, we can see how an online inference pipeline takes request parameters in the *predict* method, computes an on-demand feature, retrieves precomputed features using the request supplied *id*, and builds the final feature vector used to make the prediction with the model.

Python

```python
def loc_diff(event_ts, cur_loc) :
    return grid_loc(event_ts, cur_loc)

def predict(id, event_ts, cur_loc, amount) :
    f1 = loc_diff(event_ts, cur_loc)
    df = feature_view.get_feature_vector(
        entry = {"id":id},
        passed_features ={"f1" : f1, "amount" : amount}
    )
    return model.predict(df)
```

In the figure below, we can see important system properties for online feature stores. If you are building your online AI application on top of an online feature store, it should have **LATS properties** (low **L**atency, high **A**vailability, high **T**hroughput, and scalable **S**torage), and it should also support **fresh features** (through streaming feature pipelines).



Some other important technical and performance considerations here for the online store are:

- **Projection pushdown** can massively reduce network traffic and latency. When you have popular features in feature groups with lots of columns, your model may only require a few features. Projection pushdown only returns the features you need. Without projection pushdown (e.g., most key-value stores), the entire row is returned and the filtering is performed in the client. For rows of 10s of KB, this could mean 100s of times more data is transferred than needed, negatively impacting latency and throughput (and potentially also cost).
- Your feature store should **support a normalized data model, not just a star schema.** For example, if your user provides a booking reference number that is used as the entity ID, can your online store also return features for the user and products referenced in the booking, or does either the user or application have to provide the user ID and product ID? For high performance, your online store should support pushdown LEFT JOINs to reduce the number of database round trips for building features from multiple feature groups.

Query Engine to find similar Feature Data using Embeddings
Real-time ML systems often use similarity search as a core functionality. For example, personalized recommendation engines typically use similarity search to generate candidates for recommendation, and then use a feature store to retrieve features for the candidates, before a ranking model personalizes the candidates for the user.

The example code snippet below is from Hopsworks, and shows how you can search for similar rows in a feature group with the text "Happy news for today" in the *embedding_body* column.

```python
    news_desc = "Happy news for today"
    df = news_fg.find_neighbors(model.encode(news_desc), k=3)
    # df now contains rows with 'news_desc' values that are most
    similar to 'news_desc'
```

Do I need a feature store?
Feature stores have historically been part of big data ML platforms, such as Uber's Michelangelo, that manage the entire ML workflow, from specifying feature logic, to creating and operating feature pipelines, training pipelines, and inference pipelines.

More recent open-source feature stores provide open APIs enabling easy integration with existing ML pipelines written in Python, Spark, Flink, or SQL. Serverless feature stores further reduce the barriers of adoption for smaller teams. The key features needed by most teams include APIs for consistent reading/writing of point-in-time correct feature data, monitoring of features, feature discovery and reuse, and the versioning and tracking of feature data over time. Basically, feature stores are needed for MLOps and governance. Do you need Github to manage your source code? No, but it helps. Similarly, do you need a feature store to manage your features for ML? No, but it helps.

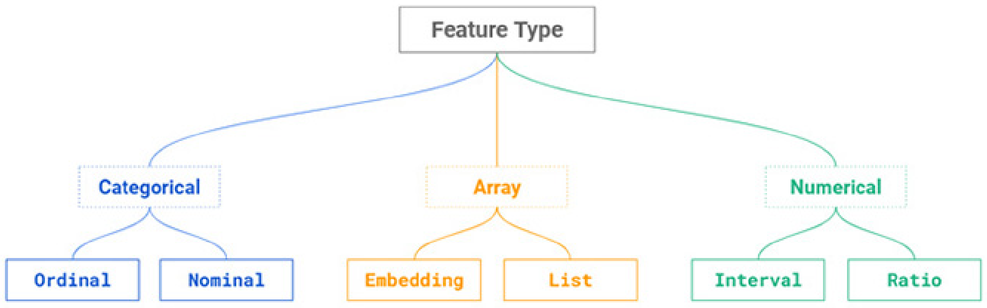
What is the difference between a feature store and a vector database?
Both feature stores and vector databases are data platforms used by machine learning systems. The feature store stores feature data and provides query APIs for efficient reading of large volumes feature data (for model training and batch inference) and low latency retrieval of feature vectors (for online inference). In contrast, a vector database provides a query API to find similar vectors using approximate nearest neighbour (ANN) search.

The indexing and data models used by feature stores and vector databases are very different. The feature store has two data stores - an offline store, typically a data warehouse/lakehouse, that is a columnar database with indexes to help improve query performance such as (file) partitioning based on a partition column, skip indexes (skip files when reading data using file statistics), and bloom filters (which files to skip when looking for a row). The online store is row-oriented database with indexes to help improve query performance such as a hash index to lookup a row, a tree index (such as a b-tree) for efficient range queries and row lookups, and a log-structured merge-tree (for improved write performance). In contrast, the vector database stores its data in a vector index that supports ANN search, such as FAISS (Facebook AI Similarity Search) or ScaNN by Google.

Is there an integrated feature store and vector database?
Hopsworks is a feature store with an integrated vector database. You store tables of feature data in feature groups, and you can index a column that contains embeddings in a built-in vector database. This means you can search for rows of similar features using embeddings and ANN search. Hopsworks also supports filtering, so you can search for similar rows, but provide conditions on what type of data to return (e.g., only users whose age>18).

Feature Type

What are feature types in machine learning?
A feature type defines the set of valid encodings (model-dependent transformations) that can be performed on a feature value. The standard feature types are categorical (ordinal or nominal), numerical (interval or ratio), and array types (lists or embeddings).



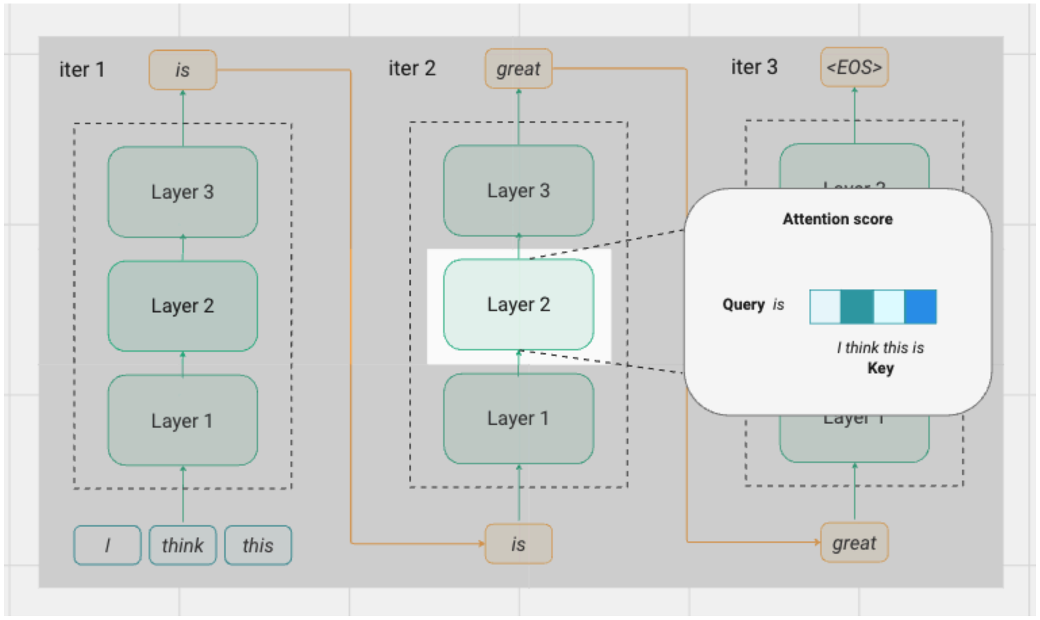
The above figure is a taxonomy for data types for features; you can read more about it in the article feature types for machine learning.

Why is it important to understand feature types?
Feature types determine how feature values can be encoded and represented in a machine learning model. Different types of features require different types of encoding and processing, and using the wrong encoding for a feature type can lead to inaccurate or suboptimal model performance.

● *Flash Attention*

What is Flash Attention?
Flash Attention is a method to improve the efficiency of transformer models, in particular large language models (LLMs), helping reduce both model training time and inference latency. Inference latency is, in particular, a challenge for LLMs, and flash attention has become a key technique that enables your LLM applications to respond faster.

Transformer models are built on the attention mechanism, which helps the model focus on relevant parts of the text input when making predictions. However, as transformer-based models become larger and larger to handle more complex tasks or larger datasets, a major limitation arises from the self-attention mechanism. This mechanism becomes increasingly slow and memory intensive as the model size grows. This is because it keeps loading and unloading data from memory. Flash Attention is introduced as a solution to mitigate this memory bottleneck problem associated with attention mechanisms in transformer models.



Why is Flash Attention important?
By improving the efficiency of attention operations, Flash Attention allows for faster training and inference of transformer-based models. Rather than loading queries, keys, and values, or intermediate computation results multiple times for each computation iteration, Flash Attention loads all the data (queries, keys, and values) just once. It then computes the attention score (conducts a series of operations) on this loaded data before writing back the final results. Additionally, it divides the loaded data into smaller blocks, aiding parallel processing.

By strategically minimizing the back-and-forth data transfers between memory types, Flash Attention optimizes resource utilization. Key strategies include "kernel fusion," which combines multiple computation steps into a single operation, reducing the need for repetitive data transfers thus reducing overhead. This streamlined approach not only enhances computational efficiency but also simplifies the implementation process, making it accessible to a broader audience of practitioners. Another key strategy is "tiling", which involves partitioning the input data into smaller blocks to facilitate parallel processing. This strategy optimizes memory usage, enabling scalable solutions for models with larger input sizes.

Optimizing Data Movement with Flash Attention

High Bandwidth Memory (HBM) offers large memory capacity but suffers from slower processing speeds. On the other hand, SRAM (Static Random-Access Memory) is a type of memory that provides fast access to data but is typically limited in capacity compared to HBM. On-chip SRAM, as the name suggests, is located directly on the chip, enabling even faster access times compared to off-chip memory.

In standard attention mechanisms, such as those used in standard transformer models, HBM is used to store, read, and write the keys, queries, and values used in the attention computation. However, the operations involved in attention calculations often lead to frequent data transfers between HBM and on-chip SRAM. For example, during computation, keys, queries, and values are loaded from HBM into on-chip SRAM for processing, and intermediate results and final outputs are written back to HBM after each step of the attention mechanism. The frequent movement of data between HBM and SRAM results in high overhead due to the time spent on data transfer and processing.

Instead, Flash Attention optimizes the movement of data between HBM and on-chip SRAM by reducing redundant reads and writes. Instead of performing these operations for each individual attention step, Flash Attention loads the keys, queries, and values only once, combines or "fuses" the operations of the attention mechanism, and then writes the results back to memory. This reduces the overall computational overhead and improves efficiency.

In summary, while standard attention mechanisms rely heavily on data movement between HBM and SRAM, Flash Attention introduces optimizations such as optimized data movement, kernel fusion, and efficient memory usage to minimize overhead and improve efficiency in memory access and computation. The impact of Flash Attention offers tangible benefits in terms of both training speed and inference latency.

Flash Attention in Fine-Tuning Frameworks

Axolotl supports flash-attention for open-source models like Llama-2 and Mistral. You can enable flash-attention by installing its profile along with axolotl:

```
pip install axolotl[flash-attn]
```

Axolotl can be used for [fine-tuning models]() on Hopsworks by simply installing it as a Python dependency in your project. Your fine-tuning training data can be loaded from Hopsworks by Axolotl using the built-in FUSE support that makes your training data, stored on HopsFS-S3, available as local files to Axolotl.

Model Serving Servers that support Flash Attention
Several [model serving]() servers now support flash attention, including vLLM, and HF's one. It is anticipated that many more model serving servers will support flash attention to supercharge LLMs.

For an Enterprise model serving solution with flash attention, Hopsworks comes with [KServe]() support, which includes support for both [vLLM]() and HF model serving servers. This gives you the benefits of scale, low latency, logging, monitoring, and access control for serving LLMs at high performance.

- ## *Function Calling with LLMs*

What is Function Calling with LLMs?
In the realm of large language models (LLMs), Function Calling refers to the ability of a LLM to impute, from the user prompt, the correct function to execute from a set of available functions and the correct parameters to pass to that function. Instead of generating standard text responses, a LLM for function calling is typically fine-tuned to return structured data responses, typically JSON objects. The returned structured data can be used to execute predefined functions, such as retrieving data from a data warehouse or feature store, retrieving real-time data, or calling 3rd party APIs. Function Calling has emerged as a significant technique, particularly in models like OpenAI's GPT models, in how these models interact with users and the external world, offering a structured approach to handling complex queries and tasks. In particular, function calling is a promising technique for opening up enterprise data directly to LLMs without the need for a vector database.

Function Calling significantly expands the capabilities of LLMs by bridging the gap between natural language understanding and practical tasks. With Function Calling, these models can seamlessly integrate with external systems, perform complex operations, and provide more accurate and contextually relevant responses.

Using Function Calling in AI Systems
You should look into function calling if you are designing an AI system that allows users to not only get text-based responses, but also executes tasks on your behalf. For example, let's consider the task of sending an email. Instead of simply generating text-based responses, the LLMs can use Function Calling to generate a structured response (such as a JSON object) that is used to execute the email-sending task

- The user initiates a conversation with the virtual assistant, expressing their intent to send an email.
- The virtual assistant will first use the LLM (e.g., a LLM fine-tuned for Function Calling), to process the user's request. Based on the context of the conversation, the model detects the need to call a specific function for sending an email.
- The model then generates a function call response, e.g., a JSON object, specifying the details of the email as the arguments of the function call, such as the recipient's email address ('to') and the content of the email ('body'). This request is then passed to the pre-defined function in the system that is responsible for sending emails.

send_email(to: "colleague@example.com", body: "Hi [Colleague's Name], I hope this email finds you well. Can we reschedule our meeting tomorrow to next Monday? Best regards, [Your Name]")

- In the backend, the system receives the function call request from the LLM. Using the provided arguments, the system executes the necessary steps to send the email(s).
- Once the email has been sent, the system prepares a response for the LLM. This response may include confirmation of the email being sent or any relevant details related to the task. The LLM can then generate natural language text to the users about this request.

Why is Function Calling Useful?
By incorporating Function Calling into LLM-based AI systems, the interactivity and utility can be largely enhanced, enabling the AI system to perform real-world tasks on behalf of users. Furthermore, Function Calling streamlines the execution of tasks, eliminating the need for users to switch between multiple applications or interfaces. On the other hand, Function Calling provides flexibility to users to define custom functions tailored to specific use cases, allowing for offering personalized experience.

Function Calling in ML Frameworks
OpenAI supports function calling and has been fine-tuned for function calling. The open-source Instructor library supports function calling with the help of Pydantic objects. OpenHermes is a fine-tuned LLM that supports function calling. Together with Instructor, they can be used by the DSPy library to build compound AI systems that optimize the chains of commands that make up the system. Both Instructor and DSPy can be installed as Python libraries in Hopsworks. OpenHermes can be downloaded for free from Hugging Face. Hopsworks additionally supports RAG through its feature store with approximate nearest neighbor indexing. You can write DataFrames to Hopsworks that are both available for querying with either function calling or similarity search.

In summary, Function Calling represents a significant new capability for LLMs to interact with Enterprise Data and Enterprise Systems, empowering developers to create more interactive and functional applications. By seamlessly integrating with external applications, systems, and APIs, LLMs become even more versatile, capable of performing a diverse range of tasks with ease.

G

● *Gradient Accumulation*

What is Gradient Accumulation

Imagine you have to **fine-tune a LLM**, but you only have a small number of GPUs, making your training memory-constrained. Or imagine you want to train an image classifier but you don't have enough GPU memory. In these cases, Gradient Accumulation can help. Gradient Accumulation is a technique used when training neural networks to support larger batch sizes given limited available GPU memory.

In traditional (mini-batch) stochastic gradient descent (SGD), training is performed in batches, primarily to improve throughput (reduce training time). During the forward pass, a batch is fed into the model, and gradients with respect to the loss function are computed. The model's parameters are then updated after computing the gradients on a single batch of training data.

However, in Gradient Accumulation, instead of updating the model parameters after processing each individual batch of **training data**, the gradients are accumulated over multiple batches before updating. This means that rather than immediately incorporating the information from a single batch into the model's parameters, the gradients are summed up over multiple batches. Once a certain number of batches have been processed (typically denoted as N batches), the accumulated gradients are used to update the model parameters. This update can be performed using any optimization algorithm like SGD or Adam. This approach reduces the amount of memory needed for training and can help stabilize the training process, particularly when working with the batch size is too large to fit into the memory.

What are the Advantages of Gradient Accumulation?

The main advantages of gradient accumulation are:

- Memory Efficiency: It allows training with larger effective batch sizes without requiring additional memory. This can be crucial when working with limited computational resources or when dealing with large models.
- Stable Training: Accumulating gradients over multiple batches can provide a more stable update direction, especially when dealing with noisy gradients. Accumulating gradients over multiple batches can reduce the impact of noisy

gradients, which may arise due to the inherent randomness in sampling mini batches. By accumulating gradients, the batch size seen by the optimizer can be effectively increased, which can lead to more stable updates and better utilization of hardware resources.

- Improved Generalization: Some studies suggest that Gradient Accumulation can lead to better generalization performance by effectively increasing the effective batch size during training.

Implementing Gradient Accumulation

When introducing Gradient Accumulation for training **machine learning** models, it's essential to understand the various considerations that come into play to ensure its effective use.

- Learning Rate Adjustment: Adjusting the learning rate is often necessary when using Gradient Accumulation. Since the effective batch size increases with accumulation, the learning rate might need to be scaled accordingly to ensure stable training. A common approach is to divide the learning rate by the accumulation factor, which is the number of batches accumulated before updating the parameters. This adjustment helps maintain a consistent learning rate relative to the effective batch size.
- Convergence Behavior: Gradient Accumulation can impact the convergence behavior of the training process. Depending on factors such as the accumulation factor and the learning rate adjustment strategy, the training dynamics may change. It's essential to monitor the training process and experiment with different accumulation strategies to ensure convergence to an optimal solution. In some cases, excessive accumulation might lead to slower convergence or even hinder convergence altogether, so finding the right balance is crucial.
- Computational Overhead: While Gradient Accumulation can be memory-efficient, it may introduce additional computational overhead. Accumulating gradients over multiple batches requires storing the gradients for each parameter until they are updated, which can increase memory usage and computation time. It's essential to consider the trade-offs between memory efficiency and computational overhead when deciding on the accumulation strategy, especially when working with limited resources.

Gradient Accumulation in ML Frameworks

Axolotl supports gradient accumulation for open-source models like Llama-2 and Mistral, by adding to the Axolotl yaml config file:

gradient_accumulation_steps: N

Axolotl can be used for fine-tuning models on Hopsworks by simply installing it as a Python dependency in your project. Your fine-tuning training data can be loaded from Hopsworks by Axolotl using the built-in FUSE support that makes your training data, stored on HopsFS-S3, available as local files to Axolotl.

In summary, Gradient Accumulation is a technique used to improve memory efficiency and stabilize training in neural networks by accumulating gradients over multiple batches before updating the model parameters. Gradient Accumulation offers advantages in terms of memory usage, stability, and potentially improved generalization performance, but it requires careful consideration of implementation details and tuning for optimal results.

● *In Context Learning (ICL)*

What is In Context Learning (ICL)?

In-context learning (ICL) is a specific method of prompt engineering where demonstrations of the task are provided to the model as part of the prompt (in natural language). With ICL, you can use off-the-shelf large language models (LLMs) to solve novel tasks without the need for fine-tuning. ICL can also be combined with fine-tuning for more powerful LLMs.

The main types of machine learning (supervised ML, unsupervised ML, semi-supervised ML, and reinforcement learning) can only learn with data they are trained on. That is, they can only solve tasks that they are trained to solve. LLMs that are large enough have shown a new type of machine learning - in-context learning - the ability to learn to solve new tasks by providing “training” examples in the prompt. In contrast to the aforementioned types of ML, the newly learnt skill is forgotten directly after the LLM sends its response - model weights are not updated.

In-context learning (ICL) learns a new task from a small set of examples presented within the context (the prompt) at inference time. LLMs trained on sufficient data exhibit ICL, even though they are trained only with the objective of next token prediction. Much of the interest in LLMs is due to the prompting with examples as it enables applications on novel tasks without the need for fine-tuning the LLM.

How to Engineer Prompts for In-Context Learning

Imagine a recipe generation service where you enter what ingredients you have available, and ask the service to generate a recipe for you. One way to implement this service would be with prompts prefixed with example recipes before your text with your available ingredients is finally added to the prompt. For this, you may have thousands of recipes indexed in a VectorDB. When the query arrives, you use the ingredients to look up the most relevant recipes in the VectorDB, then paste them in at the start of the prompt, and then write the list of available ingredients, and finally, ask your LLM to generate a prompt. This is an example of retrieval-augmented generation for LLMs.

The above service follows one piece of advice on prompt engineering - add the most relevant context at the beginning or the end of a prompt to improve the performance of LLMs. Researchers have shown that adding relevant context (i.e., the question of what recipe to generate) in the middle of the prompt leads to worse performance.

How many example recipes can you add to your prompt? That depends on the context window size for your model. GPT-4 can include ~50 pages of input text (32KB of data), and inference time slows down with larger input prompts.

Zero-Shot vs One Shot vs Few Shot Learning
Reynolds and McDonell describe fews-shot prompts as where a small number of examples of solved tasks are provided as part of the input to the trained LLM. The goal is to make predictions for new classes based on just a few examples.

Zero-Shot Inference

The following example uses zero prompts as context.

Unset
```
Prompt: Who won the 2014 World Cup?
```

Unset
```
GPT-4 Response: Germany won the 2014 FIFA World Cup.
The tournament was held in Brazil, and Germany defeated
Argentina 1-0 in the final, with the match-winning goal
scored by Mario Götze during extra time
```

The zero-shot completion includes additional information about the match that I didn't really want.
Can we use in-context learning to teach the model the format of the response I am looking for?

One-Shot Learning/Inference
The following example adds an instruction and one-shot prompt to the context:

```
Unset
    Prompt:  Answer the question using the format
    shown in the context.
    Who won the World Cup in 2014?
    Germany won the 2014 FIFA World Cup. They defeated
    Argentina in the final. It was a low scoring game.
    Who won the world cup in 2018?
```

```
Unset
    GPT-4 Response: France won the 2018 FIFA World Cup.
    They defeated Croatia in the final. It was a higher
    scoring game compared to the 2014 final.
```

Few-Shot Learning/Inference

The following example adds a few-shot prompt to the context. You can give more examples to ensure that the response by your LLM is more consistent.

```
Unset
    Prompt: Answer the question using the format
    shown in the context.
    Who won the World Cup in 2014?
    Germany won the 2014 FIFA World Cup. They defeated
    Argentina in the final. It was a low scoring game.
    Who won the World Cup in 2018?
    France won the 2018 FIFA World Cup. They defeated
    Croatia in the final. It was a high scoring game.
    Who won the world cup in 2022?
```

```
Unset
    GPT-4 Response: Argentina won the 2022 FIFA World Cup.
    They defeated France in the final. It was a high scoring
    game.
```

Model Size and Context Window Size for ICL
In-context learning benefits from larger context window sizes, as it makes it easier for ICL with prompts. Similarly, ICL can benefit from LLMs with more parameters. For example, in GPT-4, with a large 32K context window size, Kosinski showed that GPT-4 could solve 95% of a set of 40 classic false-belief tasks widely used to test Theory-of-Mind (ToM) in humans. In contrast, GPT-3 has a smaller model (up to 1000 times smaller than GPT-4) with a context window size of 2K, and it could only solve 40% of the false-belief tasks. Similarly, Levenstein et al in LLaMA 30b, a LLM with only 10s of billions of parameters and a smaller context window size, could not show ability to solve ToM problems.

Is In-Context Learning Real?
Yes, in this paper by Raventós et al, where they study ICL for linear regression and each task corresponds to a different latent regression vector, as pre-training task diversity increases beyond a threshold, transformer models outperform Bayesian estimators on unseen tasks. The implication is that ICL is an emergent phenomenon, as their transformer model moves beyond memorization of the pretraining tasks when there is sufficient diversity and scale in pre-training data. With ICL, transformers can solve new tasks not seen during pre-training.

Why does ICL work?
Informally, Charles Fyre declares, prompting is mostly subtractive, we delete potential words with each input. ICL is more about defining the task than about [learning] it. However, other researchers disagree and believe the LLMs can learn in a single shot.

Dileep et al speculate that schema learning and rebinding are key mechanisms - they believe that ICL can be explained as schema learning, and simultaneously inferring the slot-filling and latent template from the prompt.

As Dileep explains “fast rebinding is a special case of expectation maximization, where the updates are local to surprising tokens found during inference. Most of the content of the latent circuit remains ‘anchored’ to prior training, while some ‘slots’ are filled on the fly with rebinding”. In contrast, Xie et al speculate that implicit Bayesian inference is the main mechanism at work, although Dileep et al show that this is not enough to solve the “dax test” like novel word usage.

● *Inference Pipeline*

What is an inference pipeline?
An inference pipeline is a program that takes input data, optionally transforms that data, then makes predictions on that input data using a model. Inference pipelines can be either batch programs or online services. In general, if you need to apply a trained machine learning model to new data, you will need some type of inference pipeline.

Do I need an inference pipeline?
If you are building a machine learning model that will be used in production to make predictions on new data, then you will likely need an inference pipeline to take that new input data and apply the trained model to make predictions.

How do I implement an inference pipeline?
An inference pipeline typically involves first an initialization step that includes downloading the trained model from a model registry, loading any necessary dependencies (such as importing libraries), and establishing a connection to a feature store. The inference step involves setting up the input data pipeline, making predictions on the input data using the model, and optionally post-processing the predictions. If a feature store is used, the inference pipeline can read precomputed features directly from the feature store, perform any necessary on-demand feature transformations and any feature encodings, and then pass the transformed features to the model for prediction.

What is the difference between an online inference pipeline and an offline (batch) inference pipeline?
Model deployments run 24x7 and serve inference requests over the network, typically using some model serving infrastructure. The online inference pipeline is the code that is executed before and after the trained model makes predictions as part of an online inference request. In contrast, a batch inference pipeline is a program that is run on a schedule to make predictions on (typically) new inference data that has arrived since the last time the batch inference pipeline ran.

- ## *Instruction Dataset for fine-tuning LLMs*

What are Instruction Datasets for Fine-Tuning LLMs?
Instruction datasets are used to fine-tune LLMs. Fine-tuning LLMs typically uses supervised machine learning and includes both an input string and an expected output string. The input and output string follow a template known as an instruction dataset format (e.g., [INST] <<SYS>>). ChatML by OpenAI and Alpaca from Stanford are examples of Instruction Dataset Formats. The following is the instruction data format used by Alpaca for fine-tuning the includes context information (the input field below):

Python
```
Below is an instruction that describes a task, paired with an
input that provides further context.
Write a response that appropriately completes the request.

### Instruction:
```

```
{instruction}

### Input:
{input}

### Response:
```

L

- ***Lagged features***

  What are lagged features?
  Lagged features are a feature engineering technique used to capture the temporal dependencies and patterns in time series data. A lagged feature is created by taking the value of a variable at a previous time point and including it as a feature in the model at the current time point. This is done by shifting the time series data by a certain number of time steps, which is referred to as the lag or time lag.

  For example, if we have a time series of daily temperatures for the past 7 days, we can create lagged features by including the temperature values at the previous day, two days ago, three days ago, and so on. This allows the model to capture patterns in the data that are related to the previous values of the feature.

- ***LLMs - Large Language Models***

  What are LLMs (large language models)?
  LLMs stands for Large Language Models. These are machine learning models that have been trained on massive amounts of text data, such as books, articles, and web pages, to understand and generate human language. (This definition was generated by a LLM - OpenAI's GPT-3 (Generative Pre-trained Transformer 3), which has 175 billion parameters and can generate highly coherent and contextually relevant language text.)

  LLMs require graphic processing units (GPUs) to be trained and also for inference (otherwise they are very slow).

  Retrieval Augmented Generation (RAG)and In-Context Learning
  A LLM takes a query in natural language (such as English) as input and produces a

response. The input query is called a prompt. Often, you can improve the response from a LLM by carefully designing the prompt, in a process called **prompt engineering** or prompt tuning. LLMs can work well when you give them explicit instructions about how the output format of the response should be, or give them examples that you would like them to learn from (**RAG and in-context learning**). For example, if the LLM training cut off time was in 2021, and you provide the LLM as a prompt the wikipedia article for the 2022 football world cup, and add at the end of your prompt the query - "who won the 2022 world cup in football?", it will answer correctly with Argentina.

Training LLMs

LLMs are **typically trained in 3 stages**: pre-training on massive text corpus with a next-word prediction task, where individual words are masked out, and the model learns to predict the next word. The second stage is supervised fine-tuning (SFT) the LLM using instruction-output pairs, where a much smaller curated dataset of instructions and appropriate output text is used to fine-tune the LLM. The third, and final stage, is the use of RLHF to fine-tune the model with proximal policy optimization. A human takes the outputs (often 4 to 9 responses) and ranks the responses based on their preference. The ranking is used by the reward model to finetune the LLM. Llama 2 has two reward models - one for helpfulness and one for safety. **Current models like Llama** 2 use ~10K+ prompts and responses for supervised fine-tuning and ~100K+ human preference pairs.

Fine-Tuning LLMs

Recently, many pre-trained LLMs have been open-sourced and can be downloaded, and then fined-tuned on your private data to perform a specific task for you. For example, maybe you have large amounts of documentation about your company or products. In this case, you could download a pre-trained LLM (with frozen weights, from somewhere such as **HuggingFace**), and then add some extra layers and fine-tune those layers using your private data. You now have a model that should perform better on queries on your private data.

Size of LLMs (number of Parameters)

The size of a LLM is typically measured as the number of parameters it contains. The largest known model, GPT-4, has **been speculated to have 1,700 billion parameters**. In contrast, the l**argest Llama-2 model has 70 billion parameters.**

The size of a LLM is important, because certain capabilities only emerge when models grow beyond certain sizes. In a paper by Wei et al from Google Research, they showed that mathematical and word skills and instruction following appear in LLMs when they grow past certain sizes and have been trained for long enough (measured in training FLOPs).

The number of parameters in a model also determines the size of the model in memory. For example, in Llama-2, the model parameters in 16-bit precision consume:

- **Llama-2-70b with 16-bit precision = 2 bytes * 70 billion = 140 GB of memory**

In practice, this means Llama-2-70b will need at least 2 A100 GPUs (80GB) for inference or fine-tuning.

Proprietary LLMs
The most well-known LLMs released by OpenAI (ChatGPT, GPT-4) are proprietary models - you are not able to download the models, they are only accessible via a UI on their website or via API calls. For example, you pay OpenAI to use the higher end LLMs (GPT-4) and build commercial applications that call their models via APIs. Sometimes, organizations are legally prevented from using proprietary LLMs as they are restricted on what type of data they can send to an external proprietary LLM (e.g., due to data privacy). You can customize responses from proprietary LLMs with prompt engineering, but you cannot fine-tune them.

Open-Source LLMs
There are now hundreds of open-source LLMs available. Currently, the most powerful is Llama-2, released by Meta in July 2023, with 70 billion parameters. Open-source LLMs have the advantage over proprietary models that they can be fine-tuned for task-specific goals. Organizations may have valuable proprietary data (such as their customer help data or internal documentation) that they can leverage to build custom LLMs with fine-tuning. Open-source LLMs also enable organizations to deploy models within their own data centers or cloud accounts, so sensitive data will not leave their network. However, the largest open-source LLMs are still an order of magnitude smaller than the largest proprietary LLMs, so their performance is still not as good for general purpose language tasks.

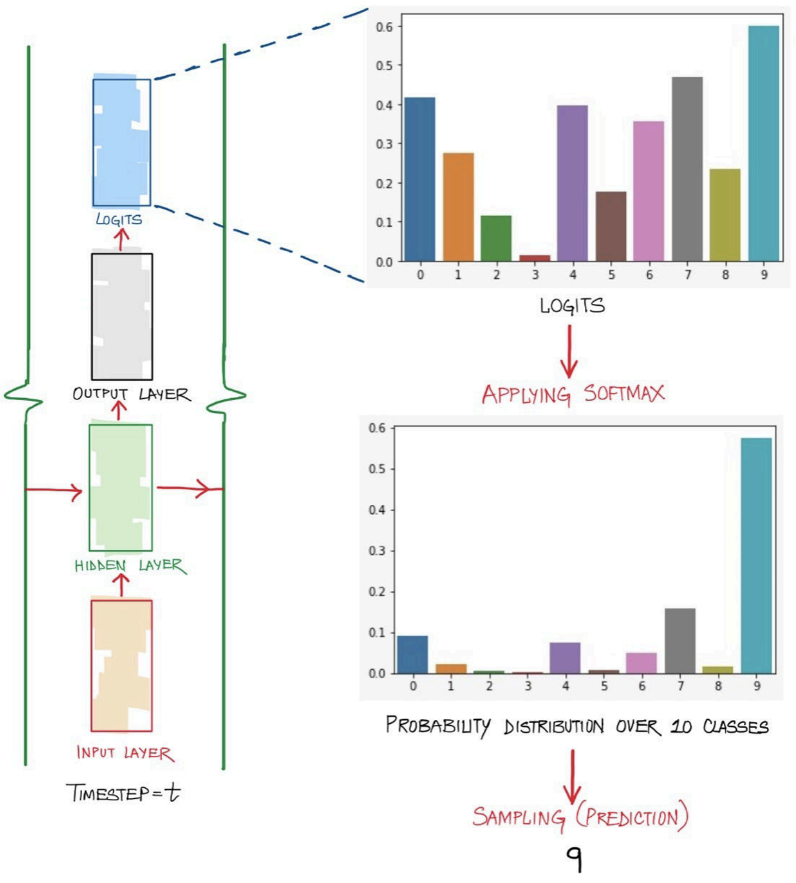
● *LLM temperature*

What are LLMs (large language models)?
LLMsIn recent years, Large Language Models (LLMs) have stood out as revolutionary tools that are capable of crafting human-quality texts, including producing coherent and contextually relevant text, translating languages elegantly, and dreaming up creative content on demand. Beneath the surface, there lies a fascinating factor that affects the nature and quality of the generated output, which is known as LLM temperature.

At its core, LLM temperature controls the balance between playing more safely and exploring new possibilities - exploration versus exploitation in the model's output. Lower temperatures favor exploiting the patterns LLMs have already learned and mastered, making the outputs more predictable and reliable. This is ideal when accurate and factual information is needed. Conversely, higher temperatures encourage exploration, meaning that the LLMs get adventurous. It ventures beyond the familiar patterns and increases the chance of being surprising and creative, potentially yielding more diverse, albeit riskier, outputs. This can be useful for brainstorming ideas.

How does LLM Temperature work?

LLMs are usually trained on large amounts of text data. They learn the patterns of how likely words appear together or apart, building a complex network of possibilities. When LLMs generate output, there are usually a few candidates in the vocabulary for each word and each candidate word has a certain likelihood of being chosen. Those likelihoods are represented by a set of logits. Then the softmax function takes the set of logits and transforms them into probabilities that sum to 1. A temperature value in the softmax function scales these logits, influencing the final possibilities calculated for each candidate word and affecting the selection of the next word in the output.



How a word is output by an LLM

The Softmax Function with LLM Temperature

Mathematically, the softmax function for a given candidate word *i* with logits *yi* is defined as:

$$P_i = \frac{e^{\frac{y_i}{T}}}{\sum_{k=1}^{n} e^{\frac{y_k}{T}}}$$

Where:
e is Euler's number (approximately 2.71828).
T is the LLM temperature parameter.
n is the size of the vocabulary.

From the above softmax function, we can see that the LLM temperature acts as a control mechanism. It affects the probabilities assigned to each candidate word by scaling the logits.

The Impact of LLM Temperature
Lower Temperature (T<1): When T is small, the softmax function magnifies differences between logits, leading to sharper probability distributions. This means that the model becomes more confident in selecting words with higher logits, making the LLM prioritize the most probable next word and effectively reducing randomness in the generated text. As a result, lower temperatures promote the exploitation of high-confidence predictions, often yielding more deterministic and conservative outputs.

Higher Temperature (T>1): On the other hand, increasing T softens the differences between logits, resulting in flatter probability distributions. Less probable words become more likely contenders. This encourages the model to explore a wider range of word choices, even those with lower logits. Consequently, higher temperatures foster diversity in the generated text, allowing the model to produce more varied and creative outputs.

Sampling Strategies
In practice, LLMs may employ different sampling strategies to incorporate LLM temperature during text generation. For example, at T=0, greedy sampling is usually employed. The model selects the word with the highest probability, effectively choosing the most confident prediction at each step.

How to choose the perfect LLM Temperature?
There's no magic number for LLM temperature. The ideal setting is based on the specific goal. Choosing the most suitable LLM temperature involves balancing various factors such as coherence, diversity, and specific task requirements. While there's no one-number-fits-all solution, here are some strategies:

Task Requirements
Coherence: If your task requires generating text that closely follows the input context or maintains a formal tone, such as summarizing research papers or writing technical reports, lower temperatures (T<1 or maybe around 0.5) may be preferable to ensure high coherence and accuracy.

Creativity and Diversity: For tasks where creativity and diversity are valued, such as creative writing or brainstorming, higher temperatures (T>1) can encourage the generation of more varied and innovative outputs.

Experiments
Experiment with different temperature values and evaluate the quality of the generated outputs. The evaluation can be done via human or user feedback. Monitor and observe how varying temperature selections impact the qualitative feedback on the performance of the LLMs. It is also worth noting that the optimal LLM temperature may not stay the same as the context or tasks evolve. Periodic reassessment and iteration are often beneficial.

Task-Specific Tuning
In some cases, fine-tuning the LLM temperature parameter for specific tasks or datasets may be necessary to achieve optimal performance. Train the LLM on domain-specific data and adjust the temperature based on the specific requirements of the task.

Finding the ideal temperature for an LLM is a delicate balancing act. Push it too high, and you risk nonsensical outputs; too low, and it becomes repetitive. It takes practice and experimentation to find the sweet spot. Furthermore, the LLM temperature value isn't the only factor affecting output. The prompt or question you provide to the LLM also plays a crucial role. A strong prompt with clear instructions might work well with a higher temperature, while a more open-ended one could benefit from a lower temperature for better exploration.

M

- ### *ML System*

What is a ML System?
A machine learning system is a computer system that is responsible for managing the data and the programs that train and operate the machine learning models that power an AI-enabled application or service.

Four types of Machine Learning Systems
Machine learning systems (ML systems) can be categorized into four different types:

- *real-time* interactive applications that take user input and use a model to make a prediction;
- *batch* applications that use models to make predictions on a schedule;
- *stream processing* applications that use models to make predictions on streaming data;
- *embedded/edge* applications that use models and sensors in resource constrained environments.

Real-time, interactive applications differ from the other machine learning systems as they often use models as external network callable services that are hosted on standalone model serving infrastructure. Batch, stream processing, and embedded/edge machine learning systems typically embed the model as part of the system and invoke the model via a function or inter-process call.

The following are examples of the four different types of machine learning systems:

Batch ML Systems

- *Dashboards* are built from predictions made by a batch ML system.
  Predict Air Quality - take observations of air quality from sensors and use weather as features for predicting air quality. A dashboard can predict air quality by using the weather forecast (input features) to predict air quality (target).
- *Interactive Systems* that use predictions made by a batch ML system.
  Google Photos Search - when your photos are uploaded to Google, it runs a classification model to identify things and places in the photo. Those things/places are indexed against the photo, so that you can search in free-text to find matching photos. For example, if you type in “bike”, it will show you your photos that have one or more bicycles in them.

Stream Processing ML Systems

- *Real-time pattern matching systems* that do not require user input are often stream processing ML systems.
  Network Intrusion Detection - if you use stream processing to extract features about all traffic in a network, you can then use a model to predict anomalies such as network intrusion.

Real-Time ML Systems

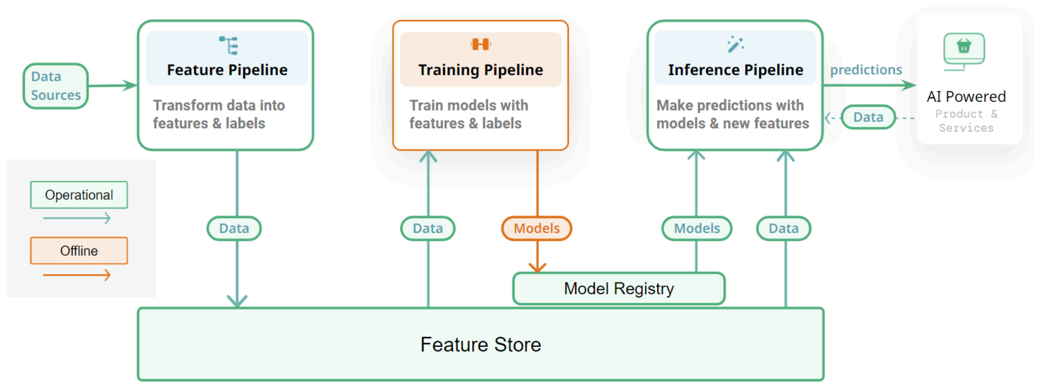
- Interactive systems that make predictions based on user input.
  ChatGPT is an example of a system that takes user input (a prompt) and returns an answer in text.

Tiktok builds its personalized recommendations engine using ML and a real-time feature store that provides historical user information and context to better personalize recommendations.

Embedded or Edge ML Systems

- *Real-time pattern matching systems* that run on resource-constrained or network detached devices.
  Tesla Autopilot is a driver assist system powered by ML that uses sensors from cameras and other systems to help the ML models make predictions about what driving actions to take (steering, acceleration, braking, etc).

Offline/Online Architecture for ML Systems



Machine learning systems are both trained and operated using cleaned and processed data (called features), created by a program called a feature pipeline. The feature pipeline writes its output feature data to a feature store that feeds data to both the training pipeline (that trains the model) and the inference pipeline. The inference pipeline makes predictions on new data that comes from the feature pipeline. Real-time, interactive ML systems also take new data as input from the user. Feature pipelines and inference pipelines are operational services - part of the operational ML system. In contrast, a ML system also has an offline component - model training. The training of models is typically not an operational part of a ML system. Training pipelines can be run on separate systems using separate resources (e.g., GPUs). Models are sometimes retrained on a schedule (e.g., once day/week/etc), but are often retrained when a new improved model becomes available, e.g., because new training data is available or the existing model's performance has degraded and the model needed to be retrained on more recent data.

● *Model Architecture*

What is model architecture in machine learning?
A model architecture is the choice of a machine learning algorithm along with the underlying structure or design of the machine learning model. Model architecture consists of layers of interconnected nodes or neurons, where each layer of the model performs a specific function, such as data preprocessing, feature extraction, or prediction.

What type of model architecture should I use for my prediction problem?
The choice of the model architecture depends on the type of problem being solved, the size and complexity of the dataset, and the available computing resources. Popular model architectures include decision trees for smaller datasets, and deep neural networks for larger datasets, including feedforward neural networks, convolutional neural networks, and transformers. The architecture of a machine learning model is a critical factor that determines its accuracy and performance, and it is often optimized through a process called hyperparameter tuning,

● *Model Bias*

What is model bias in machine learning?
Model bias refers to the presence of systematic errors in a model that can cause it to consistently make incorrect predictions. These errors can arise from many sources, including the selection of the training data, the choice of features used to build the model, or the algorithm used to train the model.

What types of model bias are there?
Common forms of model bias include selection bias, measurement bias, and algorithmic bias. Selection bias occurs when the training data is not representative of the population being modeled, leading to biased predictions. Measurement bias occurs when the measurements used to train the model are inaccurate or imprecise, leading to biased estimates. Algorithmic bias occurs when the algorithm used to train the model produces biased predictions due to inherent biases in the algorithm or the data used to train it.

How do you prevent bias in ML models?
You can prevent selection bias by ensuring your training data is representative of the different groups that your model will make predictions for. You can use evaluation sets (slices of your test set with data from groups identified of being at risk of bias) to evaluate your model performance across different groups (e.g., based on gender, ethnicity, location, etc) and identify any performance differences across those groups.

P

- ### *Parameter-Efficient Fine-Tuning (PEFT) of LLMs*

What is Parameter-Efficient Fine-Tuning (PEFT) of LLMs?
Parameter-Efficient Fine-Tuning (PEFT) enables you to fine-tune a small subset of parameters in a pretrained LLM. The main idea is that you freeze the parameters of a pre-trained LLM, add some new parameters, and fine-tune the new parameters on a new (small) training dataset. Typically, the new training data is specialized for the new task you want to fine-tune your LLM for (e.g., for the clinical domain).

What are examples of PEFT techniques?
Adapters add tunable layers to the various transformer blocks of an LLM. Prefix tuning adds trainable tensors to each transformer block.

LoRA (Low-rank adaptation of large language models) has become a widely used technique to fine-tune LLMs. An extension, known as QLoRA, enables fine-tuning on quantized weights, such that even large models such as Llama-2 can be trained on a single GPU. The QLoRA paper states that " [QLoRA] reduces memory usage enough to finetune a 65B parameter model on a single 48GB GPU while preserving full 16-bit fine tuning task performance. QLoRA backpropagates gradients through a frozen, 4-bit quantized pretrained language model into Low Rank Adapters (LoRA)."

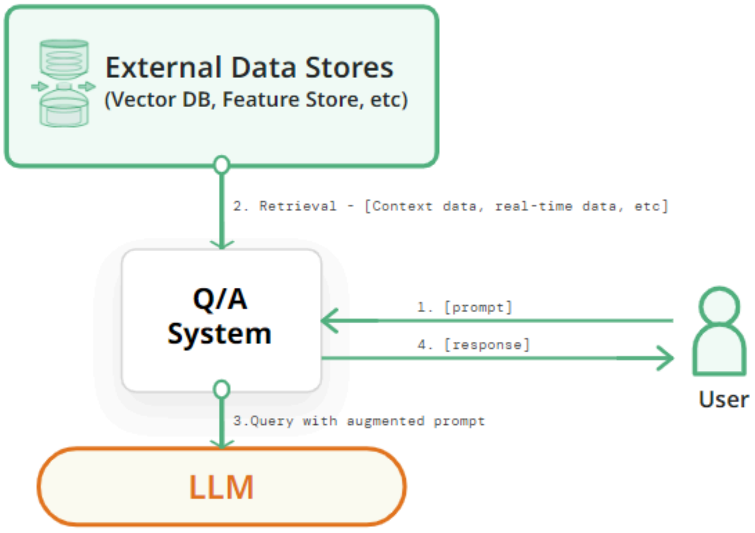
Why is Parameter-Efficient Fine-Tuning important?
Fine-tuning a Large-Language Model (LLMs) has traditionally required retraining its entire set of parameters. However, with even open-source models, such as Llama-2-70b requiring 140GB of GPU memory, this approach is computationally expensive. PEFT enables you to fine-tune a LLM with less resources.

R

- ### *Retrieval Augmented Generation (RAG) for LLMs?*

What is Retrieval Augmented Generation (RAG) for LLMs?
Retrieval-augmented generation (RAG) for large language models (LLMs) aims to improve prediction quality by using an external datastore at inference time to build a richer prompt that includes some combination of context, history, and recent/relevant knowledge (RAG LLMs). RAG LLMs can outperform LLMs without retrieval by a large margin with much fewer parameters, and they can update their knowledge by replacing their retrieval corpora, and provide citations for users to easily verify and evaluate the predictions.

The most common systems used to provide external data for RAG LLMs are vector databases and feature stores. RAG for LLMs works because of the ability of LLMs to perform in-context learning.


External Data Stores
(Vector DB, Feature Store, etc)
2. Retrieval - [Context data, real-time data, etc]
Q/A System
1. [prompt]
4. [response]
User
3.Query with augmented prompt
LLM

RAG for LLMs

RAG integrates information retrieval (or searching) into LLM text generation. It uses the user input prompt to retrieve external "context" information from a data store that is then included with the user-entered prompt to build a richer prompt containing context information that otherwise would not have been available to the LLM. Some examples of context information used by RAG include:

- real-time context (the weather, your location, etc);
- user-specific information (orders the user has made at this website, actions the user has taken on the website, the user's status, etc);
- relevant factual information (documents not included in the LLM's training data - either because they are private or they were updated after the LLM was trained).

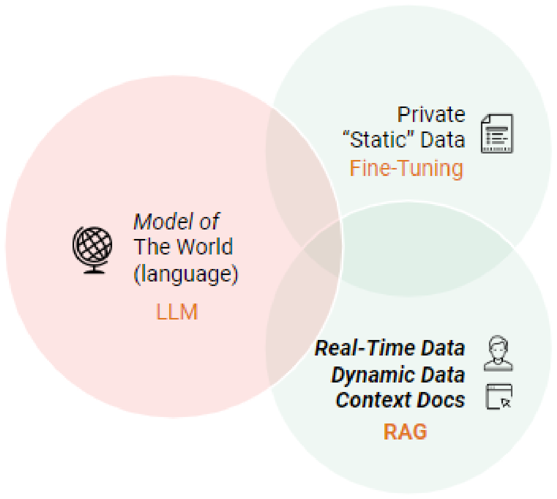
Why is there a need for RAG LLMs?

Pre-trained LLMs (foundation models) do not learn over time, often hallucinate, and may leak private data from the training corpus. To overcome these limitations, there has been growing interest in retrieval-augmented generation which incorporate a vector database and/or feature store with their LLM to provide context to prompts, also known as RAG LLMs.

What data systems are used for RAG LLMs?

Vector databases are used to retrieve relevant documents using similarity search. Vector databases can be standalone or embedded with the LLM application (e.g., Chroma embedded vector database). When structured (tabular) data is needed, an operational data store, such as a feature store, is typically used. Popular vector databases and feature stores are Weaviate and Hopsworks that both provide time-unlimited free tiers.

How does RAG compare with Fine-Tuning?
Fine-tuning takes a pre-trained LLM and further trains the model on a smaller dataset, often with data not previously used to train the LLM, to improve the LLM's performance for a particular task.

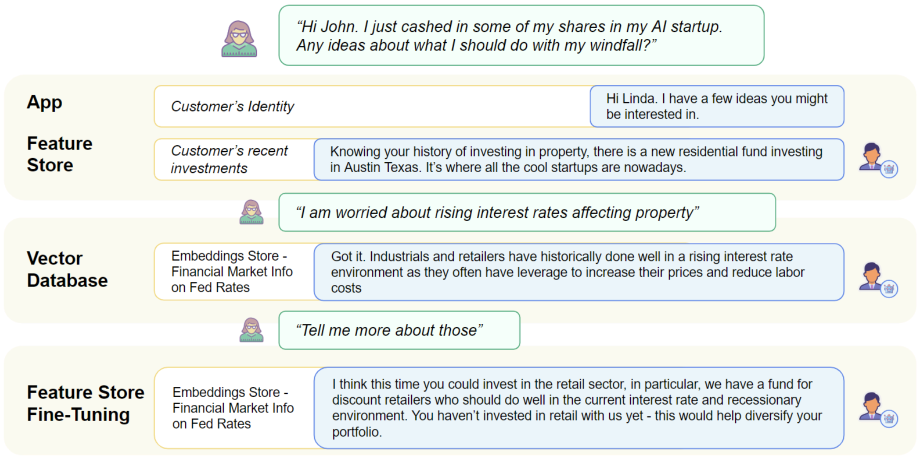


LLMs can be extended with both RAG and Fine-Tuning

In the above figure, we can see that fine-tuning is appropriate when you want to customize a LLM to perform well in a particular domain using private data. For example, you can fine-tune a LLM to become better at producing Python programs by further training the LLM on high-quality Python source code.

In contrast, you should use RAG when you are able to augment your LLM prompt with data that was not known to your LLM at the time of training, such as real-time data, personal (user) data, or context information useful for the prompt.

Challenges with RAG
One of the challenges in using a vector database is the stochastic nature of similarity search - how do you know the document you retrieve is actually relevant for the prompt?


"Hi John. I just cashed in some of my shares in my AI startup. Any ideas about what I should do with my windfall?"
App
Customer's Identity
Hi Linda. I have a few ideas you might be interested in.
Feature Store
Customer's recent investments
Knowing your history of investing in property, there is a new residential fund investing in Austin Texas. It's where all the cool startups are nowadays.
"I am worried about rising interest rates affecting property"
Vector Database
Embeddings Store - Financial Market Info on Fed Rates
Got it. Industrials and retailers have historically done well in a rising interest rate environment as they often have leverage to increase their prices and reduce labor costs
"Tell me more about those"
Feature Store Fine-Tuning
Embeddings Store - Financial Market Info on Fed Rates
I think this time you could invest in the retail sector, in particular, we have a fund for discount retailers who should do well in the current interest rate and recessionary environment. You haven't invested in retail with us yet - this would help diversify your portfolio.

LLM Question-Answering System using RAG (feature store and vector database) as well as Fine-Tuning

In contrast, using structured data in RAG, such as in the example above is quite straightforward. In this figure, the user has logged onto the website, so we have the user's ID. With the user's ID, we can retrieve the customer's recent investments as a list of strings that we prepend to the prompt. In the above figure, looking up the relevant document in a vector database to the user's concern about "rising interest rates affects property" is more challenging. If you query the vector database with "rising interest rates affects", it has a good chance of returning a relevant document, but if you include "property" in your vector database query, it may return a document related to property, which is not the context we would like to provide to steer the conversation towards other investment opportunities.

● *RoPE Scaling*

What is RoPE Scaling?

Imagine training an LLM on sentences with an average length of 10 words. It performs brilliantly within this range, understanding the relationships between words and generating coherent text. But what happens when you throw a 50-word sentence at it? The LLM might struggle on word order and context. This is the issue of extrapolation.

LLMs rely on Rotary Position Embeddings (RoPE) to understand the relative position of words within a sequence. Each word in the sequence is assigned a unique embedding based on its position. This embedding is calculated using a combination of sine and cosine functions, incorporating its distance from the beginning and end of the sequence. However, standard RoPE struggles with longer sequences than those encountered during training. The embedding values for distant words become very similar, making it difficult for the LLM to distinguish their relative positions. Therefore, the embeddings become less effective, leading to poor performance. Extrapolation essentially refers to the maximum sequence length an LLM can handle effectively with its original RoPE settings. Beyond this limit, performance degrades significantly.

RoPE Scaling modifies the RoPE calculations to improve the model's ability to handle longer sequences. The core idea is to tweak the base value used in the RoPE calculations. This value controls the rate at which the sine and cosine functions oscillate, affecting the embedding distribution. Increasing the base value can spread out the embeddings, making them more distinct for longer sequences. While decreasing it can introduce periodicity, allowing the model to handle longer sequences that wrap around this cycle.

Adjusting the base value can involve either increasing or decreasing it, depending on the specific LLM architecture. In the paper **‘Scaling Laws of RoPE-based Extrapolation**’, it emphasizes the importance of finding the optimal base value for a specific LLM and task, often achieved through experimentation and fine-tuning. Once the base value is adjusted, the LLM undergoes further training with longer sequences. This fine-tuning helps the model adapt to the modified RoPE embeddings and learn how to interpret the position information more effectively for unseen lengths.

RoPE Scaling for LLMs

By incorporating RoPE Scaling, LLMs become more adept at handling sequences exceeding their training data, and process diverse data formats and structures, leading to more accurate and reliable outputs for various tasks. RoPE Scaling also opens doors for exploring new applications of LLMs, such as text summarization of longer documents or code generation for complex functionalities.

While RoPE Scaling offers exciting possibilities, it's essential to consider the following aspects during implementation:

- While RoPE Scaling improves extrapolation, it may not perfectly generalize to all sequence lengths.
- Finding the right balance between base value adjustment and fine-tuning is crucial for optimal performance. Research papers like ‘**Scaling Laws of RoPE-based Extrapolation’** delve deeper into the mathematical foundations of RoPE Scaling and explore different scaling strategies.
- The success of RoPE Scaling hinges on proper fine-tuning with relevant data sets for the desired task.
- Modifying RoPE embeddings and retraining the LLM can be computationally expensive, requiring significant resources.

In conclusion, RoPE Scaling equips developers with a valuable tool to push the boundaries of LLMs. By overcoming the limitations of extrapolation, we can unlock a new era of possibilities for LLMs.

S

- ***Similarity Search***

What is similarity search for vector embeddings?
Vector embeddings (or embeddings or vectors) are compressed representations of data such as text, images, and audio. Vector similarity search (or similarity search for embeddings) finds the "top K" most similar vectors to a query vector in a vector database. In order to be able to search for items in a vector database, you need to first insert vector embeddings for items, then the items will be indexed by the vector database.

What is similarity search used for?
Similarity search is used to build:

- **single modality similarity search** where a user supplies an Image/video/audio that is run through an embedding model to create a vector with which a similarity search is performed on the vector database to find similar images/video/audio. Here the similarity search is for the same modality (e.g., use an image to find images);
- **Two-modality similarity search** where a user enters a search string (a query) or the last video they watched, and we augment that with the user's history and preferences and context information like what's trending and with all that data, we use an embedding model to create a vector. We then look up items (a different modality from the user search) in a vector database. Two-modality similarity search is enabled by training models using the two-tower model, where you have samples where a user query (and its embedding) is linked to an item the user clicked on (or negatively - the user didn't click on it).
- **Prompt engineering for generative AI tools (LLMs)** where you take a user's query and retrieve similar text passages stored in a vector database, and add those text passages as "prompts" to your query. This will better instruct the LLM how to generate a more relevant, informed response. For example, if you retrieve recent information from your vector database (not available when the LLM aws trained), the LLM can give you a correct answer, even though it didn't know that data at the time of training;
- **Anomaly detection** where you supply a vector and search for outliers in the vector database (outliers are very different from all other vectors in the vector DB);
- **Deduplication and record matching** where similarity search can find very similar (but maybe not 100% identical) items. This can be useful for detecting plagiarism, for example.

What indexing algorithms are used for similarity search?
Approximate k-nearest neighbors (ANN) is commonly used to return the k nearest vectors - exact matching is too computationally expensive. Approximate approaches find good enough matches without exhaustively checking all the possible matches. Your ANN algorithms should be configurable to enable you to tradeoff recall (percentage of results with true top-k nearest neighbors), latency, throughput, and vector insertion time.

Popular approximate approaches include:

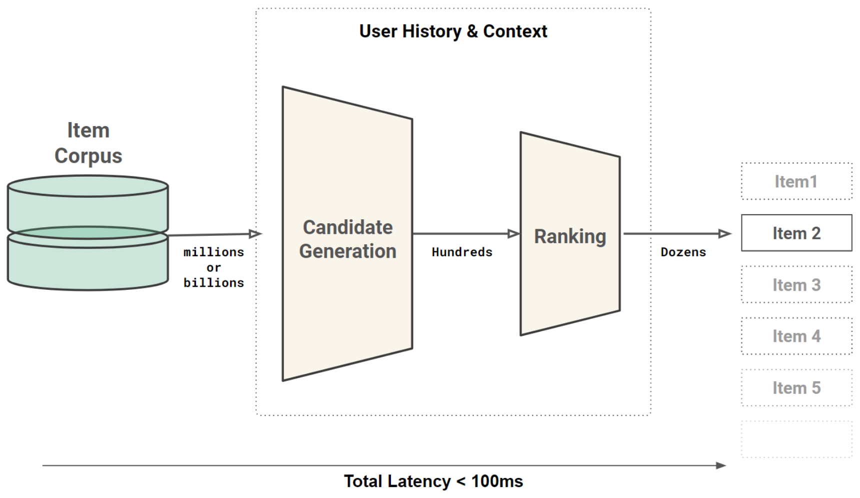
- **HNSW** (Hierarchical Navigable Small Worlds) is a proximity graph indexing and retrieval algorithm. Upper layers contain only "long connections," while lower layers have only "short connections" between vectors in the database. HNSW searches through the graph starting from the uppermost layer moving to the lowest layer, greedily traversing the graph with the longest inter-vector connections for the vector closest to our query vector - until at algorithm termination, you are left with the closest neighbors;
- **FAISS** (Facebook AI Similarity Search) enables vectors to be compared with L2 (Euclidean) distances or dot products and uses quantization and binary indexes to reduce search latency at the cost of recall. Compared to HNSW, it does not build a complex indexing structure, enabling it to be optimized for memory usage and speed;
- **SCANN** (Scalable Approximate Nearest Neighbors) uses search space pruning and quantization for Maximum Inner Product Search and also supports other distance functions such as Euclidean distance.

T

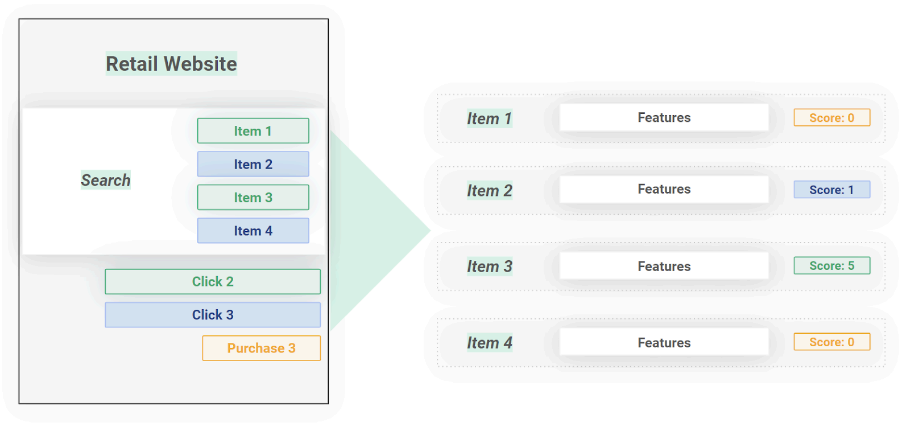
• *Two-Tower Embedding Model*

What is a Two-Tower Embedding Model?
The two-tower (or twin-tower) embedding model is a model training method for connecting embeddings in two different modalities by placing both modalities in the same vector space. For example, a two-tower model could generate embeddings of both images and text in the same vector space. Personalized recommendation systems often use items and user-histories as the two different modalities. The modalities need to be “grounded”. For example, image and text can be “grounded” by creating training data where a caption matches an image. Two-tower models are able to map embeddings from different modalities into the same space by ensuring both modalities have the same dimension “d”. For example, if the item embedding is of length 100, then the query embedding dimension should be 100.

Personalized recommendations for Products - Linking Two Modalities



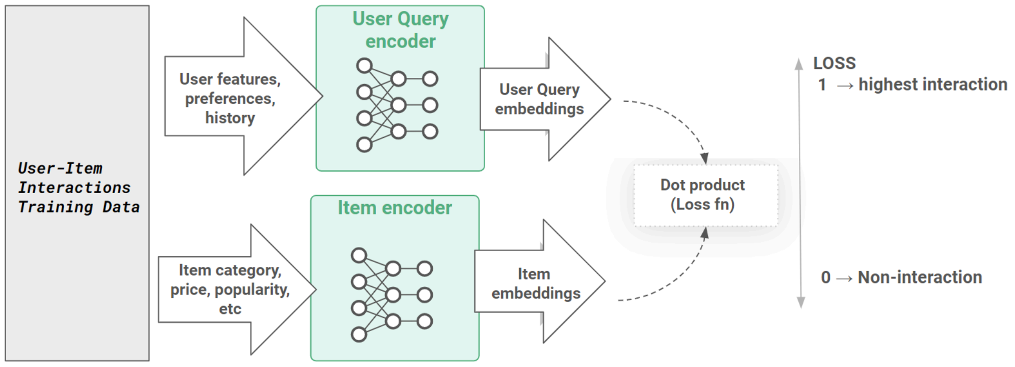
The two-tower model for personalized recommendations combines two items and "user history and context". That is, given user history and context, can we generate hundreds of candidate items from a corpus of millions or billions of items? Given those hundreds of candidates, can personalize the ranking of the candidates to the user's history and context (e.g., items that are trending)?



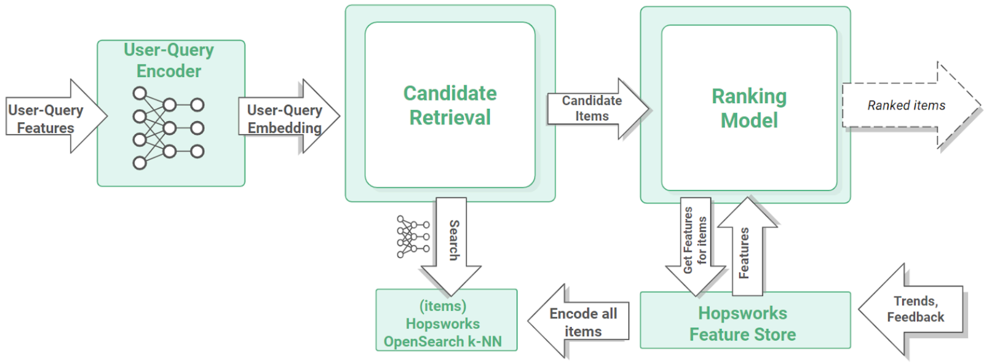
We can create training data for our personalized recommender system that combines the two modalities by presenting items in response to the user query. The user may click on some item, purchase another item, place another item in a shopping cart, and not click on another item. We collect training samples as the combination of the item, a score for the user's action (0=not clicked, 5=purchased, 1=clicked), and the user's query, user history, and context.

What is a two-tower embedding model architecture
Most two-tower architectures are used for personalized search/recommendations, where you have queries and items as the two modalities. In this case, the two-tower embedding model architecture is a deep learning model architecture that consists of a query tower and an item tower. The query tower encodes search query and user profile to query embeddings, and the item tower encodes the item, store, and location features to item embeddings.

The probability of a user query resulting in an item being clicked or placed in a shopping cart is computed using a distance measure (such as the dot product, cosine similarity, Euclidean distance, or Hadamard product) between the embeddings from two towers. The query and item tower models are trained jointly on the history of user queries and item interactions.



Personalized Recommendations/Search with Hopsworks



You can use the [Hopsworks platform] to manage the collection and usage of feature data when building two-tower models. Hopsworks includes a [feature store], [model registry], and vector database, providing both the online services needed to collect and manage training data, and the online infrastructure for candidate retrieval (VectorDB) and personalized ranking (feature store).

What other types of applications use the twin-tower model architecture?
Bytedance used test/images with ALBERT and Vision transformer twin-tower model architecture. Another example is the TextGNN Architecture that uses the two tower structure for decoupled generation of query/keyword embeddings

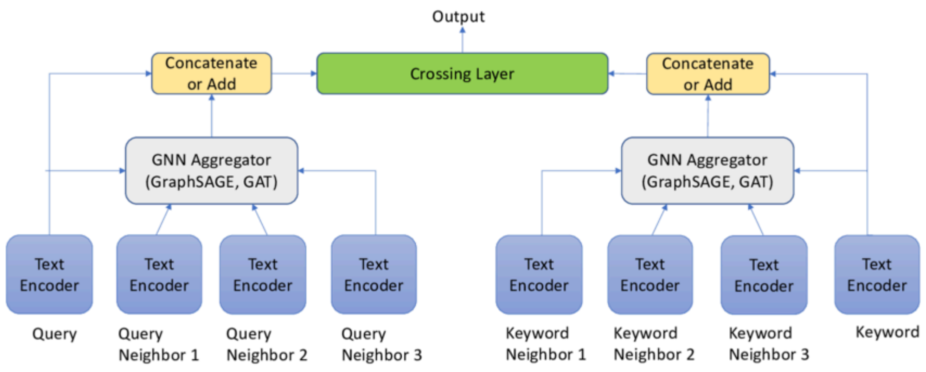


Image from TextGNN: Improving Text Encoder via Graph Neural Network in Sponsored Search by Zhu et al

Spotify have used Graph embeddings in combination with a two-tower embedding model to build a personalized recommendation system for audio books - "unlike music and podcasts, audiobooks, initially available for a fee, cannot be easily skimmed before purchase, posing higher stakes for the relevance of recommendations" [De Nadai et al]

Personalized Audiobook Recommendations at Spotify
Through Graph Neural Networks

WWW '24, May 13–17, 2024, Singapore

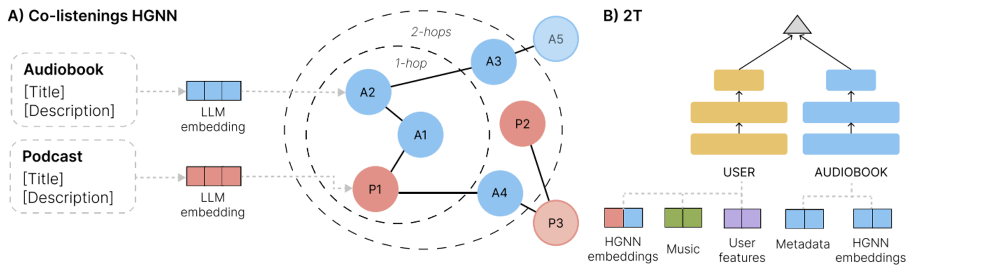


Figure 3: Overview of our model. A) We represent audiobook-podcast relationships using a heterogeneous graph comprising two node types: audiobook and podcast, connected to each other whenever at least one user has listened to both. Each node has LLM embedding features extracted from the titles and descriptions of audiobooks and podcasts. We use a 2-layers HGNN on top of this graph. B) Our 2T model recommends audiobooks to users by exploiting HGNN embeddings, user demographic features (e.g. country and age), and historical user interactions (music, podcasts and audiobooks) represented as embeddings.

Image from Personalized Audiobook Recommendations at Spotify Through Graph Neural Networks by De Nadai et al

Can two-tower embedding models be extended to more modalities?
A two-tower embedding model connects vectors in only two different modalities. Research is ongoing in generalizing to 3 modalities or more. Multi-modal models trained as N-tower embedding models could potentially connect vectors from N different modalities.

V

- ### *Vector Database*

What is a vector database in ML?
A vector database for machine learning is a database that stores, manages, and provides semantic query support for embeddings (high-dimensional vectors). The semantic queries supported are typically similarity search, nearest neighbor search, and clustering. Vector databases are available as the following: standalone servers, as a library to embed in an application, or as a capability in an existing database.

When do I need a vector database?
The most popular applications of vector databases are in AI, where you may want to quickly find similar items (e.g., a matching fashion item in an ecommerce store, or a matching piece of text for use as a prompt input to a large language model). Similarity search can also be used to bridge two modalities. For example, you can take a user's history and search query (first modality) and match it to the most similar item (product/video/song/etc) in a catalog (second modality). This two-tower similarity search method is used for personalized search and recommendation systems.

Why can I not use my existing database?
Some databases have added support for both the storage of vectors and vector similarity search. In relational databases, Postgres has added vector database support with pgvector. In document databases, OpenSearch has had vector database support since 2021 with the kNN plugin, and ElasticSearch added vector database support recently.

There is a tradeoff to make between using an existing database and a standalone vector database. There are higher operational costs in managing a new system, but standalone vector databases tend to have wider support for more ANN algorithms than databases that support vector storage and similarity search.

What are the most popular ANN algorithms used by Vector Databases?
When you query a vector database, you enter a vector (array of integers) and you ask to

return the N, e.g., N=100, most similar vectors. K-nearest neighbor algorithms are accurate (they find the nearest neighbors) but are too slow to be used in practical systems. Hence, approximate nearest neighbor (ANN) algorithms are used to find the closest neighbors (most similar vectors) to an input vector. Typically they use distance measures to compare vectors, such as L2 (Euclidean) distances or dot products.

Most of the ANN algorithms used in existing vector databases have been written in the last 5 years (it is a fast moving space), and they generally are designed to favor either accuracy (recall) or performance (throughput). The three most popular algorithms are:

- hierarchical navigable small worlds network (hnsw),
- faiss-ivf,
- Scann.

For an open-source benchmark with more details, we refer you to **ANN Benchmarks**.

What are the most popular open-source libraries that existing vector databases build on? **FAISS** (Facebook AI Similarity Search) FAISS is an open-source library for efficient similarity search and clustering of dense vectors. **SCANN** (Scalable Compressed Approximate Nearest Neighbors), developed by Google, is an open-source library for efficient similarity search and approximate nearest neighbor search in high-dimensional vector spaces. **HNSW library** is a fast approximate nearest neighbor search library that is incorporated in many existing vector databases. Some companies have also built their own proprietary libraries, such as **Weaviate.**

How do I evaluate the performance of a vector database?
You can perform benchmarks on different vector databases and ANN algorithms by using one or more appropriate datasets and distance measures. A good place to start is **ANN Benchmarks** that includes a github repository with benchmarks for many vector databases and libraries. The most common benchmarks are:

- write throughput (measured in inserts per second)
- read throughput (measured in queries per second)
- accuracy (measured as a *Recall percentage*)

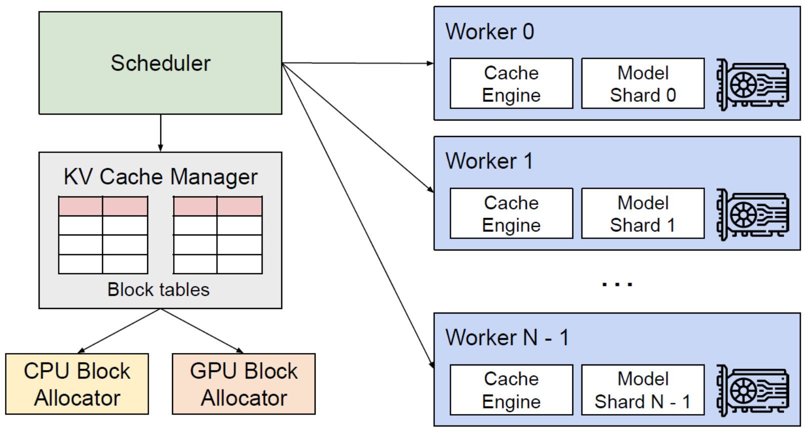
Recall vs Throughput Tradeoffs
A useful property to evaluate for your requirements in benchmarks is accuracy vs performance for Vector Database, dataset, and ANN algorithm That is,

- *Recall* (the fraction of true nearest neighbors found, on average over all queries) vs
- *Throughput* (the number of queries per second)

Depending on your requirements, you typically need to make a choice to favor an algorithm that has higher Recall or higher throughput. Different ANN algorithms make different tradeoffs in favoring throughput or Recall.

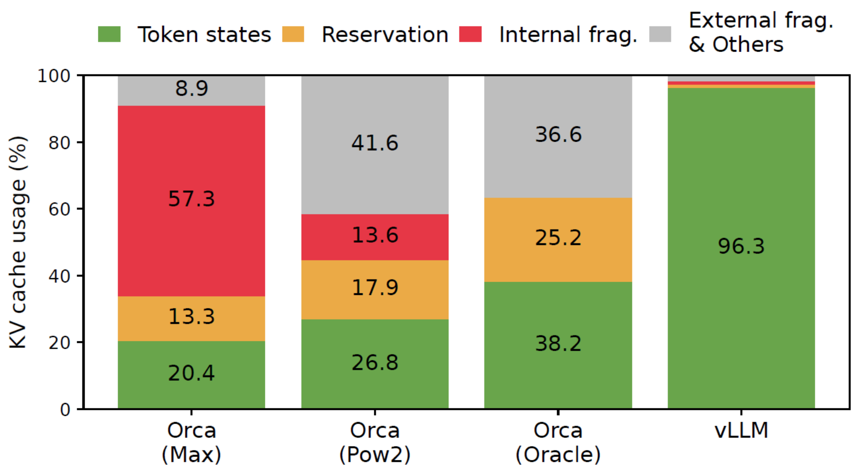
● *vLLM*

What is vLLM?
Nowadays large language models (LLMs) have revolutionized various domains. However, deploying these models in real-world applications can be challenging due to their high computational demands. This is where vLLM steps in. vLLM stands for Virtual Large Language Model and is an active open-source library that supports LLMs in inferencing and **model serving** efficiently.



vLLM architecture

vLLM was first introduced in a paper - Efficient Memory Management for Large Language Model Serving with PagedAttention, authored by Kwon et al. The paper identifies that the challenges faced when serving LLMs are memory allocation and measures their impact on performance. Specifically, it emphasizes the inefficiency of managing Key-Value (KV) cache memory in current LLM serving systems. These limitations can often result in slow inference speed and high memory footprint.



Memory usage and waste in different LLM serving systems.

To address this, the paper presents PagedAttention, an attention algorithm inspired by virtual memory and paging techniques commonly used in operating systems. PagedAttention enables efficient memory management by allowing for non-contiguous storage of attention keys and values. Following this idea, the paper develops vLLM, a high-throughput distributed LLM serving engine that is built on PagedAttention. vLLM achieves near-zero waste in KV cache memory, significantly improving serving performance. Moreover, leveraging techniques like virtual memory and copy-on-write, vLLM efficiently manages the KV cache and handles various decoding algorithms. This results in 2-4 times throughput improvements compared to state-of-the-art systems such as FasterTransformer and Orca. This improvement is especially noticeable with longer sequences, larger models, and complex decoding algorithms.

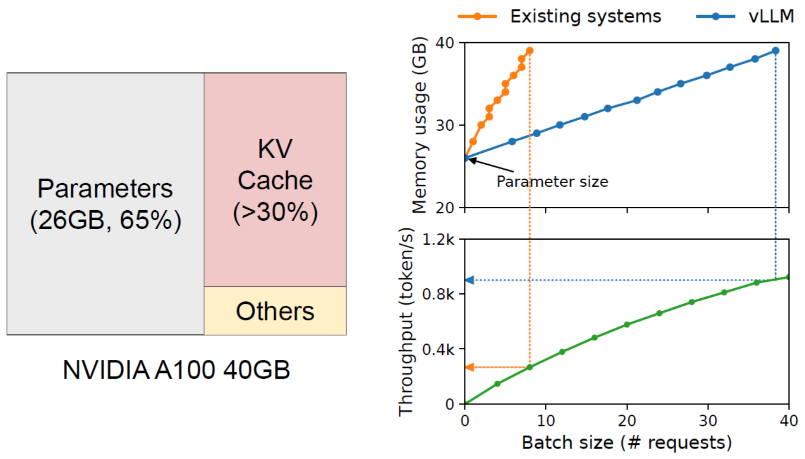
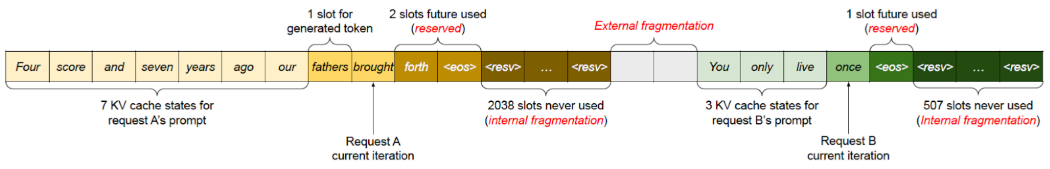


Illustration of vLLM's performance in optimizing memory usage and boosting serving throughput.

What is the core idea in vLLM?

PagedAttention

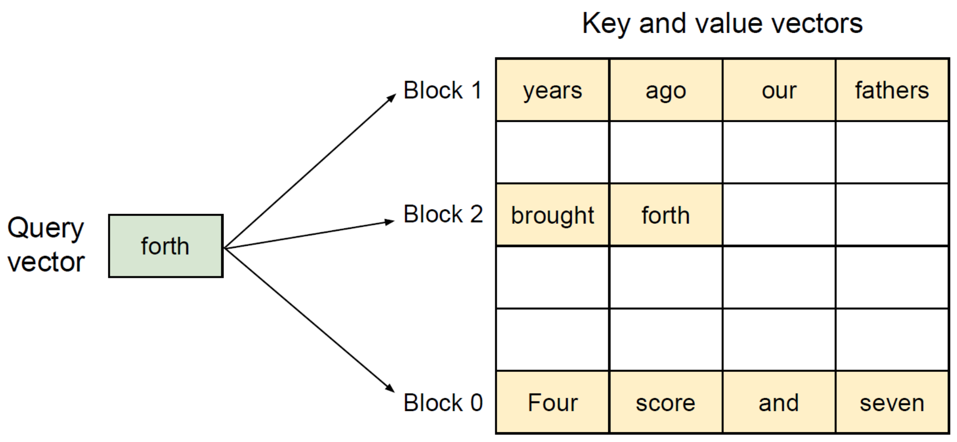
The attention mechanism allows LLMs to focus on relevant parts of the input sequence while generating output/response. Inside the attention mechanism, the attention scores for all input tokens need to be calculated. Existing systems store KV pairs in contiguous memory spaces, limiting memory sharing and leading to inefficient memory management.


1 slot for generated token
2 slots future used (reserved)
External fragmentation
1 slot future used (reserved)
Four
score
and
seven
years
ago
our
fathers
brought
forth
<eos>
<resv>
...
<resv>
You
only
live
once
<eos>
<resv>
...
<resv>
7 KV cache states for request A's prompt
Request A current iteration
2038 slots never used (internal fragmentation)
3 KV cache states for request B's prompt
Request B current iteration
507 slots never used (Internal fragmentation)

KV

Cache memory management in existing systems.

PagedAttention is an attention algorithm inspired by the concept of paging in operating systems. It allows storing continuous KV pairs in non-contiguous memory space by partitioning the KV cache of each sequence into KV block tables. This way, it enables the flexible management of KV vectors across layers and attention heads within a layer in separate block tables, thus optimizing memory usage, reducing fragmentation, and minimizing redundant duplication.



PagedAttention algorithm.

What are the other techniques used in vLLM for efficient serving?
vLLM doesn't stop at PagedAttention. It incorporates a suite of techniques to further optimize LLM serving.

- Continuous Batching: Incoming requests are continuously batched together to maximize hardware utilization and reduce computing waste, minimizing idle time.

- Quantization: vLLM utilizes quantization techniques like FP16 to optimize memory usage by representing the KV cache in reduced precision, leading to smaller memory footprints and faster computations.
- Optimized CUDA Kernels: vLLM hand-tunes the code executed on the GPU for maximum performance. For example, for fused reshape and block write, optimized kernels are developed to split the new KV cache into blocks, reshape them for efficient memory access, and save them based on a block table, all fused into a single kernel to reduce overheads.

How to use vLLM?
vLLM is easy-to-use. Here is a glimpse into how it can be used in Python:

One can install vLLM via pip:

```
Python
    # (Recommended) Create a new conda environment.
    conda create -n myenv python=3.9 -y
    conda activate myenv

    # Install vLLM with CUDA 12.1.
    pip install vllm
```

Offline inference
Then import the vLLM module into your code and do an offline inference with vLLM's engine. The LLM class is to initialize the vLLM engine with a specific built-in LLM model. The LLM models are by default downloaded from HuggingFace. The SamplingParams class is to set the parameters for inferencing.

```
Python
    from vllm import LLM, SamplingParams
```

Then we define an input sequence and set the sampling parameters. Initialize vLLM's engine for offline inference with the LLM class and an LLM model:

```
Python
    prompts = ["The future of humanity is"]
    sampling_params = SamplingParams(temperature=0.8,
    top_p=0.95)
```

```
llm = LLM(model="meta-llama/Meta-Llama-3-8B-Instruct")
```

Finally, the output/response can be generated by:

```python
Responses = llm.generate(prompts, sampling_params)
print(f"Prompt: { Responses[0].prompt!r}, Generated text: {
Responses[0].outputs[0].text!r}")
```

The code example can be found here.

Online serving
To use vLLM for online serving, OpenAI's completions and APIs can be used in vLLM. The server can be started with Python:

```python
python -m vllm.entrypoints.openai.api_server --model
NousResearch/Meta-Llama-3-8B-Instruct --dtype auto --api-key
token-abc123
```

To call the server, the official OpenAI Python client library can be used. Alternatively, any other HTTP client works as well.

```python
from openai import OpenAI
client = OpenAI(
    base_url="http://localhost:8000/v1",
    api_key="token-abc123",
)

completion = client.chat.completions.create(
  model="NousResearch/Meta-Llama-3-8B-Instruct",
  messages=[
    {"role": "user", "content": "Hello!"}
```

```
        ]
    )

    print(completion.choices[0].message)
```

More examples can be found on the official vLLM documentation.

What are the use cases of vLLM?

vLLM's efficient operation of LLMs opens numerous practical applications. Here are some compelling scenarios that highlight vLLM's potential:

- Revolutionizing Chatbots and Virtual Assistants: With its efficient serving support, vLLM can contribute chatbots and virtual assistants to hold nuanced conversations, understand complex requests, and respond with human-like empathy. By enabling faster response times and lower latency, vLLM ensures smoother interactions. Additionally, vLLM empowers chatbots to access and process vast amounts of information, allowing them to provide users with comprehensive and informative answers. vLLM's ability to handle diverse creative text formats can be harnessed to craft personalized responses that address the user's specific needs and preferences. This combination of speed, knowledge, and adaptability can transform chatbots from simple FAQ machines into invaluable tools for customer service, technical support, and even emotional counseling.

- Democratizing Code Generation and Programming Assistance: The field of software development is constantly evolving, and keeping pace with the latest technologies can be challenging. vLLM can act as a valuable companion for programmers of all experience levels. By leveraging its code-understanding capabilities, vLLM can suggest code completions, identify potential errors, and even recommend alternative solutions to coding problems. This can significantly reduce development time and improve code quality. vLLM's ability to generate documentation can also alleviate a major pain point for developers. Automatically generating clear and concise documentation based on the written code would save developers valuable time and effort, and the quality and consistency of the documentation can also be controlled. vLLM can be used to create educational tools that introduce coding concepts in a fun and interactive way, making programming more accessible to students and aspiring developers.