Chapter 12. Model Serving Patterns

Once they've been trained, ML models are used to generate predictions, or results, a process referred to as *running inference* or *serving the model*. The ultimate value of the model is in the results it generates, which should reflect the information in the training data as closely as possible without actually duplicating it. In other words, the ML model should generalize well and be as accurate, reliable, and stable as possible. In this chapter, we will look at some of the many patterns for serving models, and the infrastructure required.

The primary ways to serve a model are as either a batch process or a real-time process. We'll discuss both, along with pre- and postprocessing of the data, and more specialized applications such as serving at the edge or in a browser.

Batch Inference

After you train, evaluate, and tune an ML model, the model is deployed to production to generate predictions. In applications where a delay is acceptable, a model can be used to provide predictions in batches, which will then be applied to a use case sometime in the future.

Prediction based on batch inference is when your model is used offline, in a batch job, usually for a large number of data points, and where predictions do not have to (or cannot) be generated in real time. In batch recommendations, you might only use historical information about customer–item interactions to make the prediction, without any need for real-time information. In the retail industry, for example, batch recommendations are usually performed in retention campaigns for (inactive) customers with high propensity to churn, or in promotion campaigns.

Batch jobs for prediction are usually generated on a recurring schedule, such as daily or weekly. Predictions are usually stored in a database and can be made available to developers or end users.

Batch inference has some important advantages over real-time serving:

- You can generally use more complex ML models to improve the accuracy of your predictions, since there is less constraint on inference latency.
- Caching predictions is generally not required:
  - Employing a caching strategy for features needed for prediction increases the cost of the ML system, and batch inference avoids that cost.
  - Data retrieval can take a few minutes if no caching strategy is employed, and batch inference can often wait for data retrieval to

make predictions, since the predictions are not made available in real time. This is not always the case, however, and will depend on your throughput requirements.
- There are cases where caching is beneficial for meeting your throughput requirements, even for batch inference.

However, batch inference also has a few disadvantages:

- Predictions cannot be made available for real-time purposes. Update latency of predictions can be hours, or sometimes even days.
- Predictions are often made using “old data.” This is problematic in certain scenarios. Suppose a service such as Netflix generates recommendations at night. If a new user signs up, they might not be able to see personalized recommendations right away. To help with this problem, the system might be designed to show recommendations from other users in the same age bracket or geolocation as the new user so that the new user has better recommendations while they are showing what their preferences are through their own choices.

Batch Throughput

While performing batch predictions, the most important metrics to optimize are generally cost and throughput. We should always aim to increase the throughput in batch predictions, rather than the latency for individual predictions. When data is available in batches, the model should be able to

process large volumes of data at a time. As throughput increases, the latency with which each prediction is available increases with the size of the batch, since the individual predictions are generally not available until the entire batch is finished. But batch prediction scenarios assume that predictions need not be available immediately. Predictions are usually stored for later use, and hence latency can be compromised.

The throughput of a model or a production system that is processing data in batches can be increased by using hardware accelerators such as GPUs and TPUs. We can also increase the number of servers or workers in which the model is deployed, and we can load several instances of the model on multiple workers to increase throughput by splitting the batch between workers that run concurrently.

Batch Inference Use Cases

Batch inference is common and lends itself well to several important use cases.

Product recommendations

New-product recommendations on an ecommerce site can be generated on a recurring schedule using batch inference, which results in storing these predictions for easy retrieval rather than generating them every time a user

logs in. This can save inference costs since you don’t need to guarantee the same latency as real-time inference requires.

You can also use more predictors to train more complex models, since you don’t have the constraint of prediction latency. This may help improve model accuracy, but it depends on using delayed data, which may not include new information about the user.

Sentiment analysis

User reviews are usually in text format, and you might want to predict whether a review was positive, neutral, or negative. Systems that use customer review data to analyze user sentiment for your products or services can make use of batch prediction on a recurring schedule. Some systems might generate product sentiment data on a weekly basis, for example.

Real-time prediction is not needed in this case, since the customers or stakeholders are not waiting to complete an action in real time based on the predictions. Sentiment analysis is used to improve a product or service over time, which is not a real-time business process.

An approach based on a convolutional neural network (CNN), a recurrent neural network (RNN), or long short-term memory (LSTM) can be used for sentiment analysis. These models are more complex, but they often provide

higher accuracy. This makes it more cost-effective to use them with batch prediction.

Demand forecasting

You can use batch inference for models that estimate the demand for your products, perhaps on a daily basis, for use in inventory and ordering optimization. Demand forecasting can be modeled as a time series problem since you are predicting future demand based on historical data. Because batch predictions have minimal latency constraints, time series models such as ARIMA and SARIMA, or an RNN, can be used over approaches such as linear regression for more accurate prediction.

ETL for Distributed Batch and Stream Processing Systems

Now let’s explore what batch inference looks like with time series data, or other data types that are updated frequently and that you need to read in as a stream.

Data can be of different types based on the source. Large volumes of batch data are available in data lakes, from CSV files, logfiles, and other formats. Streaming data, on the other hand, arrives in real time. One example of streaming data would be the data from sensors.

Before data is used for making batch predictions, it has to be extracted from multiple sources such as logfiles, CSV files, APIs, apps, and streaming sources. The extracted data is often loaded into a database and then queried in batches for prediction.

As we discussed in Chapter 7, the entire pipeline that prepares data is known as an ETL pipeline. An ETL pipeline is a set of processes for extracting data from data sources, transforming it (if necessary), and loading it into some form of storage such as a database or data warehouse, from where it might be used for multiple purposes including running batch predictions, performing analytics, or mining data. Extraction from data sources and transformation of data can be performed in a distributed manner, where data is split into chunks and processed in parallel by multiple workers.

ETL is often performed using frameworks such as Apache Spark, Apache Flink, or Google Cloud Dataflow. Apache Beam is especially useful for ETL processes such as these because of the portability it enables through its support of a wide range of underlying frameworks, including Spark, Flink, and Dataflow.

Streaming data such as sensor data can be ingested into streaming frameworks such as Apache Kafka and Google Cloud Pub/Sub. Cloud Dataflow using Apache Beam can perform ETL on streaming data as well. Spark has a product specifically for processing streaming data, called Spark

Streaming. Apache Kafka can also act as an ETL engine for streaming data. The streaming data may in turn be collected into a data warehouse such as BigQuery, or into a data mart or data lake. It can also serve as a source for streaming data in another pipeline.

Introduction to Real-Time Inference

Generating inferences from trained models in real time, often while a customer is waiting, can be very challenging. That’s especially true with high volumes of requests and limited compute resources, especially in mobile deployments.

In contrast to batch prediction, in real-time prediction you often need the current context of the customer or whatever system or application is making the request, along with historical information, to make the prediction. This often requires joining their input data with historical data to form the request.

The number of requests or queries per second can vary widely based on the time of the day or day of the week, and your resources need to be able to scale up to serve peak demand and scale down, if possible, to save on cost.

Real-time inference is often a business necessity, since it allows you to respond to user actions in real time based on predictions with new data. This is extremely helpful for doing personalization on products and services

based on user requests. Recommendation systems also take advantage of real-time inference. Using new data to make predictions allows you to adapt quickly to changes in users or systems. For example, knowing that a customer has just purchased blue socks tells your recommendation system to stop recommending blue socks to that customer. Historical data in a batch system would be delayed until the next batch is run, and the customer would be annoyed by recommendations for blue socks.

Making real-time inferences often requires your system to respond within milliseconds. In many cases, data required for the prediction will be stored in multiple places, so the process for retrieving features necessary for predictions also needs to meet the latency requirements. For instance, a prediction may require user data that is stored in a data warehouse. If the query to retrieve this data takes too long to return, the data may need to be cached for quicker retrieval. This requires additional resources, but it can be less costly than scaling up compute resources.

Depending on the algorithm used, you may need to allocate more computational resources so that your system is able to produce inferences in a reasonable time frame. If budget is a concern, and it almost always is, you might want to consider using simpler models if you can get an acceptable level of accuracy from them.

Models may also sometimes generate invalid predictions. For instance, if a regression model predicting housing prices generates a negative value, the

inference service should have a policy layer that acts as a safeguard, and this policy layer must also meet the latency requirements. This requires the data scientist or ML engineer to understand the potential flaws of the model outputs and the response times of the different systems that might be involved in generating a prediction, as well as their scalability.

As you consider your options, keep in mind that as a general rule, shorter latency equals higher cost. Delivery of real-time predictions can be done either synchronously or asynchronously, which we'll discuss next. We'll then consider ways to optimize real-time inference.

Synchronous Delivery of Real-Time Predictions

There are two ways to deliver real-time predictions: synchronously or asynchronously. Let's first consider synchronous delivery.

In this context, the client interacts with an ML gateway. The gateway serves as a hub to interact with the deployed model to send requests and receive predictions. The request for prediction and the response (the prediction itself) are performed in sequence between the caller and the ML model service. That is, the caller blocks, waiting until it receives the prediction from the ML service before continuing.

Asynchronous Delivery of Real-Time Predictions

Asynchronous predictions are delivered to the consumer independent of the request for prediction. There are two main approaches:

Push

The model generates predictions and pushes them to the caller or consumer as a notification. An example is fraud detection, where you want to notify other systems to take action when a potentially fraudulent transaction is identified.

Poll

The model generates predictions and stores them in a database. The caller or consumer periodically polls the database for newly available predictions.

Notice the difference in complexity between this asynchronous system and the synchronous system we just looked at. A synchronous or blocking system tends to be much less complex to implement and maintain, but it can have a significantly higher level of wasted resources.

Optimizing Real-Time Inference

We can adopt several strategies to try to optimize online inference. For example, we can try scaling our compute resources. We can try using

hardware accelerators such as GPUs instead of CPUs for inference if we can tolerate the increased costs. We can also add more than one GPU or CPU to enable parallel processing of requests in order to balance increasing load on the server.

We should always try to optimize the models that are being served. Sadly, in the quest for higher model metrics, the benefits of a less accurate but highly optimized model are sometimes not appreciated as much as they should be.

In an online serving environment, it is always better to use simpler models such as linear models for inference (rather than complex models such as deep neural nets), if and only if an acceptable level of prediction accuracy can be achieved. This is because latency, rather than accuracy, is the key requirement for many or most online serving systems. Less accuracy has an incremental impact on the value of the prediction, but latency that is too long can result in a model that is simply not usable.

Using simpler models will of course not work for some applications where prediction accuracy is of utmost importance, if acceptable accuracy cannot be achieved with a simpler model. In those cases, accepting higher costs is often unavoidable.

Another strategy we can adopt is caching features that should be fetched from a datastore for prediction. Using fast caches that can support faster

retrieval of input features will help achieve lower latency.

Real-Time Inference Use Cases

To make this discussion more concrete, let’s consider some real-world use cases:

Target marketing

A system might check to see whether to send a retention or promotion offer to a particular customer while they are browsing a website, based on the propensity score predicted in real time for this customer. For example, how likely are they to buy if they receive a discount?

Bidding for ads

This involves synchronously recommending an ad and optimizing a bid when receiving a bid request. This information is then used to return an ad reference in real time. Many ad brokers, including Google, have developed highly optimized systems for this use case. Often, the difference between success and failure is measured in milliseconds and/or hundredths of a cent.

Food delivery times

Food delivery companies such as DoorDash, Uber Eats, Grubhub, and Gojek need to estimate how long food delivery will take based on current traffic in the area, average recent food preparation time,

average recent delivery time in the area, and other available information. This is core to their business, since food should be delivered before it gets cold.

Autonomous driving systems

Latency is critical for autonomous driving systems. Autonomous vehicles use several different kinds of models in real time. For example, object detection models for scene understanding use data from devices such as cameras, radars, and lidars. These models must be small enough to be deployed to systems on the vehicle and fast enough to have prediction times on the order of 10–20 ms, while still being accurate and resilient enough to handle a wide range of road and weather conditions without failure. Failures of these models can be catastrophic, including delays in returning inference results that are caused by unacceptable latency.

Serving Model Ensembles

Increasingly, we are seeing use cases in which using a collection of models composed as an ensemble is far more effective than using a single, larger model. There are several potential motivations for doing this:

- Models that are already trained for specific tasks and data can be composed to serve new use cases.

- Models can be loaded on distributed systems for more flexible scaling, and sometimes to be more regionally distributed.
- Intelligent routing of requests to smaller or larger models can reduce costs.

Ensemble Topologies

Model ensembles are traditionally grouped into topologies based on the graph structure of the ensemble, the most basic being a simple linear pipeline or cascade ensemble. Other topologies include voting and stacking ensembles. Note that although bagging and boosting models are technically ensembles of models, they are nearly always trained and served as a single model, so we will not include them in this discussion.

More generally speaking, model ensembles are typically implemented as directed acyclic graphs (DAGs), although through the use of conditionals, they can potentially include cycles. This makes serving them similar in some ways to running the types of training pipelines we have discussed throughout this book so far.

Example Ensemble

A very simple example of an ensemble is a cascade ensemble that implements a voice chatbot. The user's voice request is sent to a speech-to-text model, whose output is sent to a large language model (LLM) to

compose a response, whose output in turn is sent to a text-to-speech model to respond to the user (see Figure 12-1).

Figure 12-1. A simple cascade ensemble

Ensemble Serving Considerations

When serving an ensemble, it helps to have a server that supports serving models as a group. Both Ray Serve and NVIDIA Triton offer support for *model composition* (i.e., serving models in an ensemble).

One key consideration is the memory residency of the models in the ensemble. If only some of the models can be loaded into memory concurrently, the latency caused by having to load models to complete a request can be prohibitive for many real-time use cases. For batch inference, this is less of a problem but can still considerably increase the time required to run a batch, so batching intermediate results between models becomes important. It's also often more efficient to configure the model to accept asynchronous calls, rather than incurring the overhead of synchronous calls.

Model Routers: Ensembles in GenAI

Generative AI (GenAI) has increased the usage of more complex model topologies, including chaining (see Chapter 22). At the same time, it has increased the need for more sophisticated management of inference costs, due to the large costs incurred by running the largest, most capable models. This has motivated the development of smaller models with capabilities that begin to approach those of larger models as a way of decreasing costs. But those smaller models are not always capable of responding to all the requests at a level that is acceptable for some applications, which has led to the need to route requests to different models in an attempt to use the smallest, most cost-effective model while still offering an acceptable level of quality.

Just prior to the publication of this book, researchers at UC Berkeley, Anyscale, and Canva collaborated on RouteLLM, an open source framework for cost-effective LLM routing. The code is available on GitHub. RouteLLM trains a model that attempts to send user requests to the best model for that specific request based on model capabilities and cost, selecting between a larger, more expensive, and more capable model and a smaller, cheaper, but less capable model. Currently, RouteLLM only selects between two models. While it's easy to know which model is cheaper to use, it's more challenging to know whether the less expensive model will meet the quality requirements for the use case. The researchers' evaluation of RouteLLM on widely recognized benchmarks shows that it significantly

reduces costs—by over two times in certain cases—without compromising the quality of responses.

Data Preprocessing and Postprocessing in Real Time

Processing data for real-time serving can be particularly challenging due to latency requirements. This includes all data processing in the entire flow, from accepting the user's request to delivering a response, including preprocessing before the model and postprocessing after the model. For time series applications, techniques such as windowing become important. In all cases, it's important that the processing that is done when the model is served exactly matches the processing that was done when the model was trained, in order to avoid training–serving skew.

Let's begin by defining some terms:

Raw data

: The data that is not prepared for any ML task. It might be in a raw form in a data lake, or in a transformed form in a data warehouse or other data source.

Prepared data

A dataset in a form that is ready for training a model or running inference, or just for studying the data. Data sources are parsed, and they typically are joined and put into tabular form.

Engineered features

Features that have been tuned so that they are in a format that is expected by ML models and that helps the model learn. Examples are normalization of numerical values so that they fall between 0 and 1, and one-hot encoding of categorical values.

Data engineering

Converts raw data to prepared data. Data in incoming requests, which may include real-time data streams, might need to be converted to prepared data before making a prediction. If we are using statically stored features for prediction, they will be converted beforehand and stored for lookup.

Feature engineering

Creates engineered features by performing transformations and joins; for example, projecting text features into an embedding space, performing z-scores for numerical features, and creating feature crosses.

Some preprocessing operations include:

Data cleansing

: Correcting any invalid or empty values in incoming data

Feature tuning

: Conducting operations such as normalizing the data, clipping outliers, and imputing missing values

Representation transformation

: Performing one-hot encoding for converting categorical features to numerical features

Bucketization

: Converting numerical features to categorical features

Feature construction

: Constructing new features through feature crossing or polynomial expansion

Training Transformations Versus Serving Transformations

During both training and serving, there are many transformations that can be done element-wise, meaning we can transform individual examples without knowledge of the rest of the dataset. However, many other

transformations require knowledge of the characteristics of the dataset, such as the median or standard deviation for a numerical feature. An example of this is a z-score, which requires the standard deviation of the feature values.

This creates the need to make a full pass over the dataset to calculate the required values, such as the mean, median, and standard deviations for numerical features or the terms that are included in a vocabulary. For large datasets, making a full pass can require a large amount of compute resources. *Therefore, transformations during training include both element-wise and full-pass operations.*

Once we have gathered the required values during training, we need to store them for use during serving. We must perform the same transformations on each prediction request as we did during training so that the model receives data that is processed the same way. *Serving requests are always transformed element-wise*, which often requires the values that we calculated through a full pass during training.

Windowing

Windowing involves creating features by summarizing data values over time. That is, the instances to aggregate are defined through temporal window clauses. For example, imagine you want to train a model that estimates taxi trip time based on the traffic metrics for a route in the past 5 minutes, in the past 10 minutes, in the past 30 minutes, or at other intervals.

Another example where windowing would be used is predicting the failure of an engine part based on the moving average of temperature and vibration values computed over the past 3 minutes. Although these aggregations can be prepared offline for training, they have to be computed in real time from a data stream during serving.

More precisely, when you are preparing training data, if the aggregated value is not in the raw data, it is created during the data engineering phase. The raw data is usually stored in a database with the format (entity, timestamp, value).

However, when the model for real-time (online) prediction is being served, the model expects features derived from the aggregated values as an input. Thus, you can use a stream processing technology such as Apache Beam to compute the aggregations on the fly from the real-time data points streamed into your system. You can also perform additional feature engineering (tuning) on these aggregations before training and prediction.

Options for Preprocessing

Preprocessing of data can be performed in a number of different ways, using different tooling, including:

- Google Cloud Bigtable or BigQuery (only for training data, filtering to remove irrelevant instances, sampling to select data instances, and

performing training/validation splits)

- Dataflow (Apache Beam pipeline)
- TensorFlow
- Dataflow (Apache Beam and TensorFlow Transform)
- Some feature stores

Dataflow can perform instance-level transformations, stateful full-pass transformations, and window aggregation feature transformations. In particular, if your ML models expect an input feature such as `total_number_of_clicks_last_90sec`, Apache Beam windowing functions can compute it based on aggregating the values of time windows of real-time (streaming) event data (e.g., clicks).

Figure 12-2 illustrates the role of Dataflow in processing stream data for near real–time predictions. In essence, events (data points) are ingested into Pub/Sub. Dataflow consumes these data points, computes features based on aggregates over time, and calls the deployed ML model API for predictions. The predictions are then sent to an outbound Pub/Sub queue. From there, they can be consumed by downstream (monitoring or control) systems or pushed back (e.g., as notifications) to the original requesting client.

Another approach for this kind of data preprocessing is to store the predictions in a low-latency datastore such as Cloud Bigtable for real-time fetching. Cloud Bigtable can also be used to accumulate and store these

real-time aggregations so that they can be looked up when needed for prediction.

You can also implement data preprocessing and transformation operations in the TensorFlow model itself; for example, by using `tf.data`. The preprocessing you implement for training the TensorFlow model becomes an integral part of the model when the model is exported and deployed for predictions. Since it's included in the model, it avoids the potential for training–serving skew. However, making full passes over the dataset cannot be included in the model, so that must be done before reaching the stage of element-wise transformations.

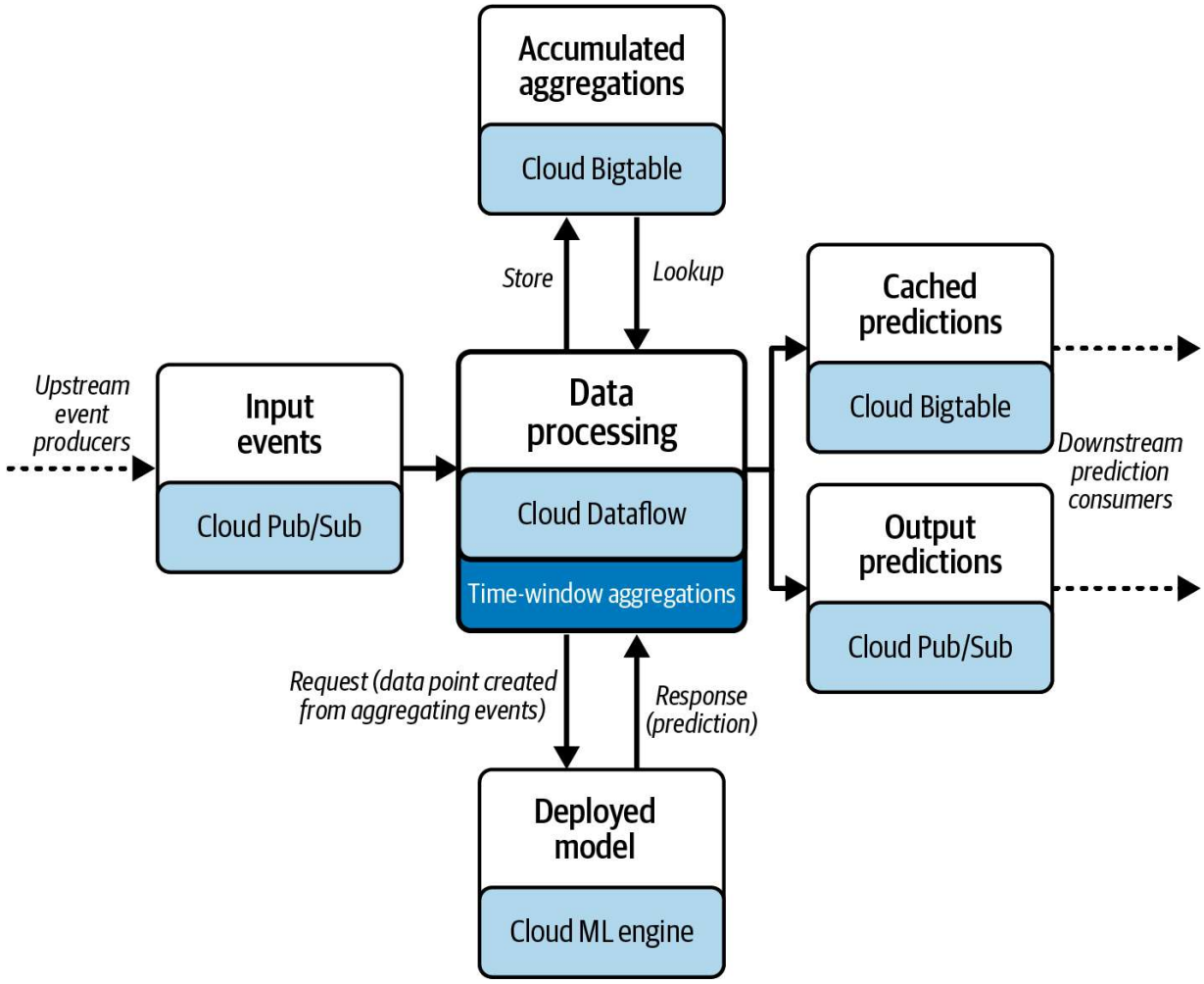

Figure 12-2. Dataflow preprocessing

TRAINING–SERVING SKEW

Training–serving skew is the difference between the data preprocessing that is done during training and the preprocessing that is done during serving. This skew can be caused by:

- A discrepancy between how you handle data in the training and serving pipelines (often caused by different code used for training and serving)
- A change in the data between when you train and when you serve
- A feedback loop between your model and your algorithm

We are concerned with training–serving skew because of the preprocessing mismatch in training and serving pipelines. If the data is preprocessed differently, the model results may be significantly different.

Enter TensorFlow Transform

The TensorFlow Transform (TF Transform) library is useful for transformations that require a full pass. The preprocessing performed in TF Transform is exported as a TensorFlow graph, which represents the instance-level transformation logic as well as the statistics computed from full-pass transformations. The Transform graph is used for preprocessing for training and serving. Using the same graph for both training and serving prevents skew because the same transformations are applied in both stages. In addition, TF Transform can run at scale in a batch processing pipeline

running on a compute cluster, to prepare the training data up front and improve training efficiency. Figure 12-3 introduces the structure of TF Transform and the most typical way in which it is used with a model.

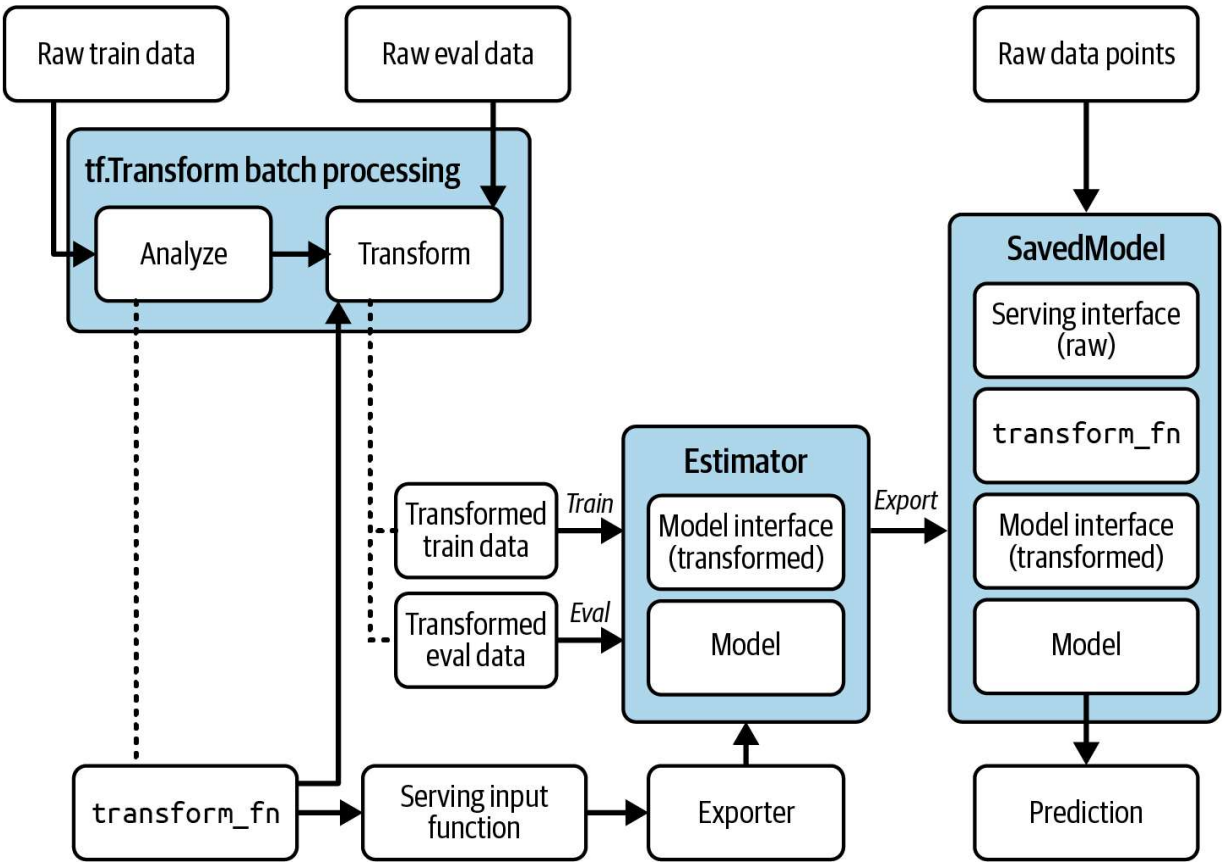

Figure 12-3. TF Transform structure

TF Transform preprocesses raw training data using transformations in the `tf.Transform` Apache Beam APIs, and it runs at scale on Apache Beam distributed processing clusters. The preprocessing occurs in two phases:

- During the *analyze phase*, the required statistics (such as means, variances, and quantiles) for stateful transformations are computed on the training data with full-pass operations. This phase produces a set of transformation artifacts, including the `transform_fn`. The `transform_fn` is a TensorFlow graph that has the transformation logic as instance-level operations and includes the statistics computed in this phase as constants.
- During the *transform phase*, the `transform_fn` is applied to the raw training data, where the computed statistics are used to process the data records (e.g., to scale numerical columns) in an element-wise fashion.

To preprocess the evaluation data, only element-wise operations are applied, using the logic in the `transform_fn` as well as the statistics computed from the analyze phase in the training data. The transformed training and evaluation data is prepared at scale, using Apache Beam, before it is used to train the model.

The `transform_fn` produced by the tf.Transform pipeline is stored as an exported TensorFlow graph, which consists of the transformation logic as element-wise operations as well as all the statistics computed in the full-pass transformations as graph constants. When the trained model is exported for serving, the Transform graph is attached to the SavedModel as part of its `serving_input_fn`.

While it is serving the model for prediction, the model-serving interface expects data points in raw format (i.e., before any transformations). However, the model's internal interface expects the data in the transformed format. The Transform graph, which is now part of the model, applies all the preprocessing logic on the incoming data points.

This resolves the preprocessing challenge of training–serving skew, because the same logic (implementation) that is used to transform the training and evaluation data is applied to transform the new data points during prediction serving.

Postprocessing

Postprocessing transformations are transformations done on the inference results before they are sent as a response to the client. They can be simple transformations, such as converting categorical data to dictionary entries or looking up additional data such as fields associated with the prediction in a database. Postprocessing is typically performed outside the model.

Vertex AI Prediction enables customizing the prediction routines, which are called when sending prediction requests to deployed models. Prediction routines implement custom preprocessing and postprocessing logic. TensorFlow Serving also allows developers to customize the prediction routine that is called when a prediction request is sent to a deployed model.

Inference at the Edge and at the Browser

If you run inference on a server, it requires a network connection to make a request and return a result. That’s not always convenient, or even possible in some use cases. This has led to the development of ways to serve models without requiring a connection to a server, meaning at the network edge or even self-contained in a web browser.

Edge computing is a distributed computing technology in which information processing and storage is done on the edge of the network infrastructure, close to the location of the device. Edge computing does not rely on processing and storage that is centrally located many miles away, but instead uses resources that are located close to the user or application. Because of this, real-time data does not suffer any latency issues, but it may introduce other issues because of constrained local resources.

There are several motivational factors for shifting AI inferencing to the edge:

Real-time responsiveness

Some applications, such as autonomous vehicles, cannot afford to contact the server every time a decision must be made. Responses must be delivered in real time so that the vehicle can respond instantaneously.

Privacy

Keeping data locally, especially personal identifiable information (PII), reduces the chance that the data will leak out of the secure environment. Any data that is uploaded to central storage should be anonymized before upload. See “Pseudonymization and Anonymization”.

Reliability

Especially for applications with strong latency requirements, depending on having a good connection to a central server can create failures and timeouts. This is also true for applications with more elastic latency tolerance, but which may operate in disconnected scenarios for significant lengths of time.

There are several applications of ML inference at the edge, including the following:

Smart homes

A set of connected Internet of Things (IoT) devices such as smart security cameras, door locks, and temperature control devices can have trained models deployed on them to make predictions so as to make your home smart.

Self-driving cars

These cars have models that use data from their sensors and cameras for inference. They cannot afford the latency involved in contacting the server before making real-time decisions, such as applying brakes when obstacles are detected in the path.

Predictive keyboards and face recognition on smartphones

These are examples of models that not only perform inference but also are trained on the device, leveraging user data for personalization to provide a better experience.

Challenges

There are several challenges involved in moving ML model inferencing to the edge. The most crucial ones are balancing energy consumption with processing power, performing model retraining and updates, and securing the user data.

Balancing energy consumption with processing power

Most edge devices have limited processing power, as compared to a central server. Inferencing using large models such as deep neural networks requires devices of higher processing power. But more advanced processors drain more battery. And to incorporate a better battery, you might need to redesign your device to be larger, which might not be feasible. Therefore, you should always design your ML models so that they use as little

processing power as possible for inferencing, while still meeting the needs of the application. This can be done by applying various optimizations during and after model building.

Performing model retraining and updates

Since data changes can cause model decay, any ML application should support retraining and updates to the model. In edge devices, performing frequent updates to the model is complicated for several reasons. For example, each edge device may have a different hardware configuration. Some might not support a particular framework or some operations. Most edge devices have wireless connectivity, and hence may not always be online, so it can be difficult to make frequent deployments or updates. For devices with no network connectivity support, you will have to manually deploy your ML model. For devices that can be connected to the internet, you can consider using containers to perform model deployments.

Securing the user data

Securing the user data collected on the edge device for inferencing or training is another concern when running models on edge devices. Storing it locally helps ensure privacy, since user data does not leave the device. But enhanced security on the device is needed because edge devices hold on to user data. Currently, there are no standard security guidelines for edge devices.

Model Deployments via Containers

In this approach to inferencing, the ML model is built, trained, and tested on some central infrastructure, typically in the cloud. The model is saved and then deployed using a container image into edge devices with different configurations, or to some server hosted in the cloud to make deployments to different hardware and software configurations more standardized. Each of these devices will have the container runtime installed, so they can run the services in the container image. The deployment workflow can be designed to meet the level of MLOps that the entire system needs to achieve.

Azure IoT Edge is a service that can help you deploy ML models and other services into IoT devices using containers. Azure IoT Edge supports a wide range of devices. It helps you package your application into standard containers, deploy those containers into any of the devices it supports, and monitor it all from the cloud.

For example, an image classifier container can be developed on a local machine and staged to the Azure Container Registry. Azure IoT Edge deploys the image classifier into the edge device, which runs the Azure IoT Edge Runtime. The Edge Runtime manages all the containers deployed in the device. There are a wide range of devices that support running the Azure IoT Edge Runtime, and deployments and updates to these devices can be standardized using Azure IoT Edge.

Training on the Device

Currently, model building, training, and testing for edge applications are usually done in the cloud, in data centers, or on the developer’s own machine. These trained models are deployed to devices at the edge for inference.

Wouldn’t it be better if training could be done locally on the device rather than in a separate location? Is training on the device possible?

The answer is yes, although with limited capabilities. Devices such as smartphones with good processing power can perform training. The best example for a model trained on smartphones is personalization for predictive typing. This model quickly learns the user’s typing patterns and learns to complete their sentences. Perhaps you’ve experienced this yourself?

There are several benefits to training a model on edge devices. Apps can learn from user data directly rather than relying on a model trained on a generic dataset. User privacy can be protected, since the data never leaves the device, not even for training a personalized model. Performing ML training or inference on edge devices can be less expensive than training on huge servers. By performing training near the location of the data, continuous learning and more frequent updates to the model are possible.

Federated Learning

Federated learning enables devices to share anonymized data and model updates. By training on data from more than one device, model accuracy and generalization is typically improved because of the larger, more varied training dataset. TensorFlow Federated is an open source framework for federated learning.

In federated learning, a device downloads the current model, improves it by learning from local data, and then summarizes the changes as a small, focused update. Only this update to the model is sent to the cloud, using encrypted communication, where it is immediately averaged with other user updates to improve the shared model. All of the training data stays on the local device, and no individual updates are stored in the cloud.

Runtime Interoperability

When working with ML models, there are many popular frameworks to choose from, including PyTorch, TensorFlow, Keras, scikit-learn, and MXNet. Once you decide which framework to choose for training models, you have to figure out how to deploy these models to a runtime environment, such as a workstation, smartphone, IoT devices like smart cameras, or even in the cloud. Different platforms and devices might be running various operating systems such as Linux, Windows, macOS, Android, iOS, or even some real-time operating system (RTOS) such as

TinyOS. And different hardware accelerators such as GPUs, TPUs, field programmable gate arrays (FPGAs), or neural processing units (NPUs) might power the device, server, or workstation.

This can make it challenging to manage deployment strategies for inferencing. This is especially true for embedded systems such as IoT devices, which run minimal versions of the Linux OS or RTOSes. There are a wide range of hardware configurations and hardware accelerators that are used in embedded systems, adding to the complexity.

One obvious strategy for ensuring interoperability is to build the model using an ML framework that is supported by the edge device you want to deploy to. Table 12-1 lists the libraries supported by a few of the popular IoT devices. For example, if you want to run your model on Raspberry Pi 4, which supports inferencing using the TensorFlow, TF Lite, and ELL libraries, you should train your model in the TensorFlow or ELL framework.

Another strategy is to use a standard model format that can be deployed to a wide variety of IoT devices with different configurations. One popular model format is Open Neural Network Exchange (ONNX), a community-driven open source standard for deep learning models. However, be aware that formats such as ONNX often have limitations that can reduce the performance of your models, or even make publishing your models in that format impossible. This situation is expected to improve in the future.

Table 12-1. Edge device software support

Edge device	Software support
Google Coral SoM	TensorFlow Lite, AutoML Vision Edge
Intel Neural Compute Stick 2	TensorFlow, Caffe, OpenVINO toolkit
Raspberry Pi 4	TensorFlow, TF Lite, ELL
NVIDIA Jetson TX2	TensorFlow, Caffe

Inference in Web Browsers

Inference can be done in web browsers with no additional software installed and without the need for an ongoing network connection. This is done by serializing the trained model as JavaScript. JavaScript is widely supported by all modern web browsers, and in most cases it will leverage hardware acceleration when available.

Deploying in the browser moves the processing burden to each client, greatly reducing the centralized resources required. It also keeps the user’s data on their client, which improves privacy. One of the most popular frameworks for making deployments in the web browser is TensorFlow.js (TFJS).

TFJS is a library for developing and training ML models in JavaScript and for deploying models in either a web browser or a Node.js server. It comes with pretrained models from Google for several common tasks such as object detection, image classification, image segmentation, and speech recognition. You can also perform transfer learning by retraining existing models such as MobileNet. TFJS can deploy models written using either JavaScript or Python.

Conclusion

As you've seen in this chapter, there are many different ways to "serve" a trained model. By "serve," what we really mean is perform inference—using the model to create a response to a request. There are also data processing considerations for serving, considerations when doing real-time serving versus batch serving, and considerations when serving model ensembles. The way you serve your model will often depend on the needs of your application and/or users, but sometimes you may have a choice between different options, and this chapter has tried to give you some understanding of the options available.