DOORDASH

Blog

Building Flexible Ensemble ML Models with a Computational Graph

January 26, 2021 | Hebo Yang Arbaz Khan Param Reddy

Modern machine learning (ML) teams have found great success in combining multiple ML models and rule-based models as ensembles for better predictive performance, but it can be a challenge to productionize them. For real-time production systems, models are usually trained with Python for fast development, serialized to a model store, and evaluated with C++ during model serving for efficiency.

However, rule-based models and models trained using multiple ML frameworks can't be serialized into a single combined model as each framework has its own serialization format and runtime prediction library.

DoorDash's ML Platform team wanted to enable ML engineers and data scientists to easily develop such models with seamless production integration. We used a computational graph approach via a domain-specific language (DSL) to allow teams to use multiple frameworks at once in a single combined model, while meeting development speed and prediction performance requirements.

Our data scientists and engineers have been using this new capability of our ML platform to develop and improve models that better serve our customers, for example, providing personalized food recommendations, improving delivery efficiency, and giving more accurate estimations for delivery time.

The need for flexible ensemble models

In a general machine learning pipeline, models are developed inside a Python notebook (i.e. Jupyter), checked into Git source control, and then trained inside some computational cluster. Trained models are uploaded to a model artifact store and loaded inside a prediction cluster in production, which may run performant C++ code to serve real-time prediction requests from other business services, as shown in Figure 1 below:

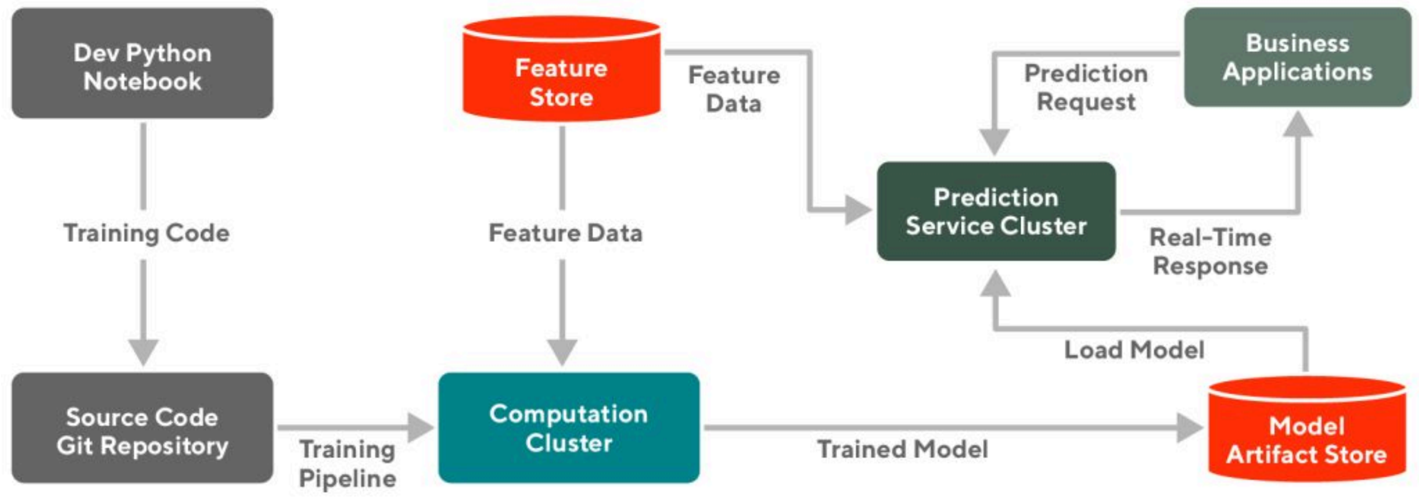

Figure 1: In a simple ML pipeline, data scientists develop a model in a Python notebook and check it into a Git repository, after which it is trained on a computation cluster. The model is served by a prediction service in a production system.

At DoorDash, the machine learning platform provides this whole infrastructure so that machine learning engineers and data scientists can focus on model development. Currently, most of the online machine learning problems at DoorDash could be solved with a decision tree-based model using LightGBM or neural networks with PyTorch (See DoorDash's ML Platform – The Beginning). However, there are many use cases that require more complex models to achieve better business outcomes:

Rule-based models

Rule-based models are an easy way to initially tackle many problems and we would like to support them with the ML framework.

For example, when a consumer places a food order through our platform, we need to offer a nearby Dasher, our term for a delivery driver, the chance to make the delivery. To estimate the best time to send the offer to the Dasher, we initially used a model that tries three features sequentially and uses the first available one. The features are historical average food preparation time estimates at individual stores, historical average food preparation time estimates across all stores, and a default numerical value. In addition, some models may require combining the predictions from multiple ML models through a simple heuristics and return the aggregated results.

Pre-processing and guardrails

Feature preprocessing and value guardrails are often needed for model inputs like linear transformations and embedding averaging. It's true that preprocessing on features could be handled as part of an ETL pipeline and stored into the feature store. However, for large scale features, it is very resource-intensive to precompute and store transformed data, especially for the features requiring exhaustive cartesian products between two features (i.e. cosine similarities between store and consumer embeddings). We'd like to support these "feature engineering on-the-fly" scenarios to increase flexibility and efficient resource usage.

For example, we only store features for aggregated historical non-overlapping data over a 30 minute time window inside the feature store. For a model that runs every minute to predict the ratio of outstanding orders to available Dashers for the next 30 minute period, we need to use weighted average features from these historical aggregated non-overlapping data sets. If we compute and store these transformed data sets in the feature store, we will have to store these weighted averages over one minute windows instead of the 30 minute window. A one minute window would require 30 times the storage space of the original data for this single model, resulting in high resource usage and scalability problems for the feature store.

Post-processing and guardrails

Another use case for ensemble models is having guardrails and post-processing for custom adjustments on model outputs. In some cases, we have seen better results when fitting models on the relative change instead of the absolute value, especially when there is significant growth. It is better to support these as part of the model framework instead of having the business applications to handle it.

When predicting merchant onsite preparation time, the LightGBM submodel prediction is compared against the merchant-provided estimated time. If the difference is significant, we will adjust the prediction result based on predefined rules. Our data scientists are also exploring stacking models using linear models (i.e linear regression) to capture major linearity between features and nonlinear models like LightGBM to model the residuals for better model performance and interpretability.

The challenge of supporting ensemble models

The machine learning platform needs to support developing and training models with these scenarios inside Python, and transforming the processing logic to C++ for each model to ensure that it is fast and scalable in production. However, not only do models from different frameworks each have their own C++ library, but the custom rules and processings defined in Python also need to be implemented for each model. We should not burden data scientists and machine learning engineers with writing their own C++ production optimizations for every model. We need a generic solution to enable easy model development and fast production performance for ensemble models.

Computational graph solution

DoorDash ML Platform team has developed an in-house generic solution using a computational graph to handle transformations and models from different frameworks. From a user's perspective, developing such an ensemble model only requires they:

1. Define a static computation graph with Python DSL
2. Call helper function to upload to a model store, which serializes the graph in JSON and models in native format (text for LightGBM and TorchScript bytes for Pytorch)

This computational graph solution with Python DSL significantly simplifies the effort needed from data scientists and ML engineers to create a model and meanwhile ensures performance as serialized model artifacts are loaded inside the Sibyl Prediction Service to serve real-time prediction requests with C++. We will explain the computational graph structure, the Python DSL, and real-time model serving in detail below.

Detailing the graph structure

The structure of ensemble models can be thought of as a directed acyclic computational graph with two types of nodes:

- **Input nodes**
  Features that will go into the ensemble model. Each node has a Name and Dimension.
- **Computation nodes**
  Derived values. Each node has a Name and Operation. The Operation defines the dependencies and transformations on dependencies to derive the node’s value. An Operation could be another node, supported arithmetic and function computation between nodes, or a ML model. An Operation can also optionally define a condition to support if/elif/else evaluations.

Computation nodes are evaluated in the exact order that appears in the configuration file. Writes to the computation node are done only if the current value in the computation node's target is NaN to support if/elif/else use cases. The computed value will be NaN if the condition on the computation node is false. Note that this is different from usual last write wins semantics in imperative languages.

We use JSON as the serialization format so that the model is still readable to human eyes. Here is an example of a simple model that subtracts 30 from the input feature:

```
{
  "numerical_features": [
    {
      "name": "feature_1"
    },
  ],
  "compute_nodes": [
    {
      "target_name": "result",
      "operation": {
        "type": "expr",
        "expr": {
          "operator": "subtract",
          "operand1": "feature_1",
          "operand2": 30
        }
      }
    }
  ]
}
```

Python DSL for easy development

We implemented a Python library so that users could easily write Python code to define an ensemble model, without worrying about details of the computational graph structure.

We will walk through the steps needed to create an ensemble model with Python DSL, starting with defining the pytorch sub-model.

```
class ToyModel(nn.Module):
  def __init__(self):
    super(Model, self).__init__()
    ...

  def forward(self, numerical_features, categorical_feature, embedding_features):
    ...
torch_model = ToyModel()
```

Next, the input nodes are declared. The node's name is optional and will be extracted from the Python variable name via inspect if possible, otherwise a unique name will be generated by the library.

```
feature_1 = NumericalInputVariable()
feature_2 = NumericalInputVariable(target_name="num_feature_2")
feature_3 = NumericalInputVariable()
cat_feature = CategoricalInputVariable()
emb_feature = EmbeddingInputVariable(dimension=2)
```

Pre-processing and guardrails on features can be performed with Python operators and custom classes. Computations between nodes are captured by operator overloading on the node classes. The ConditionalChain defines if/elif/else conditional statements.

```
feature_sum = feature_1 + feature_2
input_val = ConditionalChain(
  nodes=[
    ExpressionNode(condition=feature_1 < 0, expression=0),
    ExpressionNode(condition=feature_1 >= 0,expression=feature_1),
    ExpressionNode(expression=feature_sum)
  ]
)
```

Lastly, post-processing and guardrails on the model prediction result can be done similarly. The result node already captured all the dependencies and transformations. The model serializer uses the result node to generate and serialize the computation graph.

```
torch_node = PytorchNode(
  model=torch_model,
  numerical_features=[input_val],
  categorical_features=[cat_feature],
  embedding_features=[emb_feature],
)

result = ConditionalChain(
  nodes=[
    ExpressionNode(condition=feature_2 <= 0, expression=feature_3),
    ExpressionNode(condition=torch_node > 0, expression=torch_node * feature_3),
    ExpressionNode(expression=feature_1)
  ]
)
```

With support from Python operators, the Python DSL makes it significantly easier to define the computational graph and also makes the logic more readable to users for future reference or modifications.

Model serving in production with efficient C++

Our Sibyl prediction service, a gRPC Kubernetes microservice, serves real-time models in production. The main logic to serve requests and fetch features is written in Kotlin. It uses JNI bindings to invoke C++ implementations for all model computations. A JNI model class is initialized once with its model path and implements a function taking in features and returning prediction results. Inside this function, the native C++ library (c_api for LightGBM and libtorch for PyTorch) is invoked to compute model results for single models while our computational graph implementation handles ensemble models. The language and components are illustrated in Figure 2, below. The final prediction result is passed back to Kotlin and returned to the client in real-time.

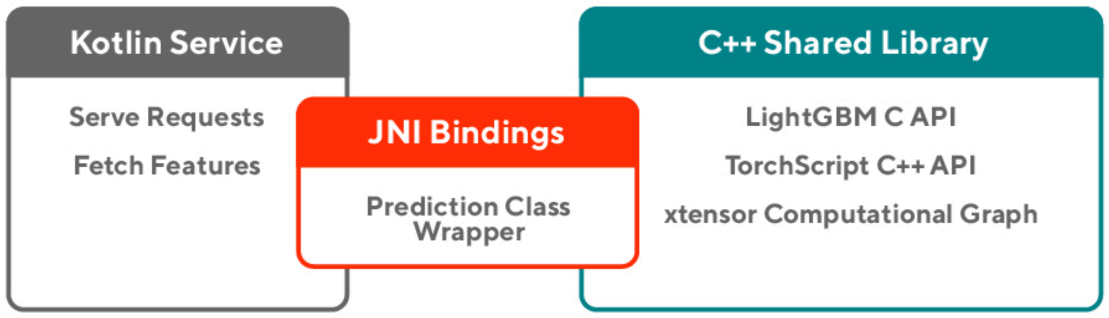

Figure 2: JNI Bindings serve as the interface between Kotlin and C++ for predictions.

The computational graph, with all the input and computation nodes, is initialized from a serialized model JSON file during class initialization. We use the xtensor matrix to store and perform computations for all the features as well as intermediate nodes. The library handles transformations against data easily and efficiently without copying data. The computation nodes are iterated sequentially, performing computations and storing each nodes' value inside the matrix. If a node contains an ML sub-model, its native C++ library is used again to compute the node's value. After passing through all computation nodes, the matrix is filled with values including the prediction results.

Performance metrics

We tested the time taken to compute 10,000 predictions against the model for estimating the ratio of outstanding orders to available Dashers described above, and compared the performance between Python code and our computational graph. This model uses a total of 37 features and a LightGBM submodel. The model involves pre-processing to compute the weighted average from historical aggregated data for 13 features, post-processing to adjust prediction against the current ratio, and a guardrail to enforce the model output range.

We wrote Python code for the logic described above, using Python-API for the LightGBM submodel. The code loads all the feature data into Pandas DataFrame, loads the LightGBM model, and then starts to measure the time to iteratively make 10,000 predictions. Feature transformations and guardrails are handled inside each prediction with Python code to simulate production scenarios. The workflow for C++ is similar, except that features data are loaded into xtensor and then passed to our generic computation graph class. Both are run inside a Docker container with one CPU since we run production code in containers inside a Kubernetes cluster. This table shows the performance data:

	Prediction Time	Total Memory Footprint	Data + Model Memory
Python	270s	120MB	21MB
C++	20s	75MB	17MB

In terms of code complexity, the custom code we wrote above to handle everything in Python takes about 20 Python statements. Our serialized computational graph contains 75 nodes and is a JSON file with 800 lines. However, using the

Python DSL we created to define the model computational graph, it only takes about 20 Python statements to generate the entire computational graph.

Conclusion

We can see that the computational graph with C++ reduces CPU time by more than a factor of 12. Given a peak QPS of three million predictions per second (though not all of them are complex ensemble models), this significantly reduces the number of nodes needed for model serving and thus infrastructure usage. On the other hand, the Python DSL could notably help boost model development speed by reducing a complex and error-prone model definition from 800 lines of JSON down to 20 lines of Python code, similar to writing native Python. Our solution empowers the data science and machine learning teams at DoorDash to enjoy easy model development with Python and fast production performance with C++ at the same time.

For a team or company that implements dedicated model-serving pipelines for similar high QPS and low latency models, the gains from implementing C++ computational graphs to support flexible ensemble models could be significant. However, for models that are of smaller scale or where latency is not a concern, the engineering effort to develop and maintain such a solution should be seriously considered against the alternative to sacrifice performance and/or to handle the ensemble logic on the client side.

Future Work

We are essentially implementing a custom computational graph with the Python SDK and C++ computation. Although this method provides fast predictions and easy model composition, it also has some drawbacks, such as consistency between Python and C++ as well as resource use to support custom and new functions.

When we first started on this work, there was limited support in PyTorch for serializing computation graphs with Python dependency, requiring that we implement an efficient in-house solution to support our business needs. We have seen recent development and better support in TorchScript with Python operations. We did a proof of concept using TorchScript custom C++ class wrappers against LightGBM c_api to achieve similar functionalities as our computational graph approach. While TorchScript actually achieves similar performance, it also requires significant effort to set up due to some bugs and incomplete documentation. Seldon core also supports model ensembling via Inference Graph from a service level and a custom inference server could potentially support the transformations.

We will continue exploring with open source frameworks for opportunities to adopt and contribute to them.

Acknowledgements

Many thanks to Ezra Berger and Wayne Cunningham for the continuous support, review, and editing on this article. Thanks to Chi Zhang, Santhosh Hari, and Xiaochang Miao for sharing their insights from the data science and machine learning perspective. And thanks to Kornel Csernai, Kunal Shah, and Swaroop Chitlur from our ML Platform team for support and review throughout the course of this work. Last but not least, thanks to the mentorship of Hien Luu on making this and many other projects for our ML Platform a success!

About the Authors

gender non-conforming, LGBTQIA+, American Indian or Native Alaskan, Black or African American, Hispanic or Latinx, Native Hawaiian or Other Pacific Islander, differently-abled, caretakers and parents, and veterans are strongly encouraged to apply. Thank you to the Level Playing Field Institute for this statement of non-discrimination.