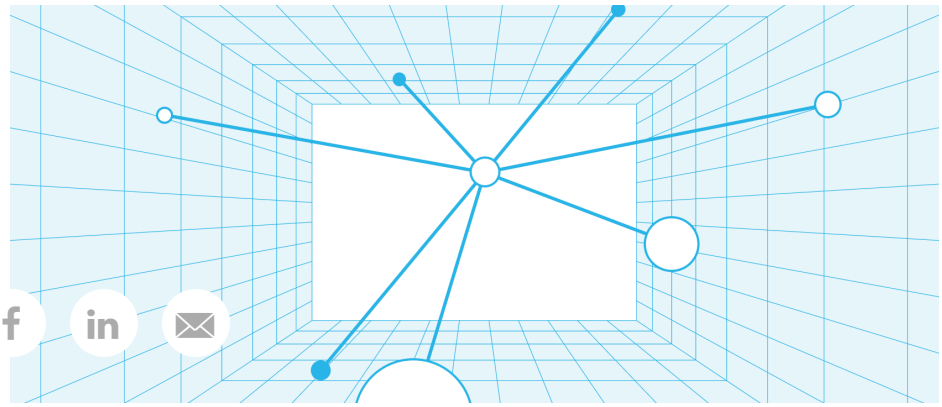
AUG 06, 2024

AUTHOR

Xinyi Jiang

SHARE

Accelerating Hyperparameter Tuning in Snowflake ML



Hyperparameter optimization (HPO) is a frequently used machine learning technique to select the best parameter combination for a given model. Popular open source packages, such as scikit-learn, provide easy-to-use APIs for users to train their model using different hyperparameter combinations with different numbers of cross-validation folds.

Snowflake ML is a set of integrated capabilities for users to carry out a complete machine learning workflow on top of your governed data. You can load data tables from Snowflake, then proceed with the Snowpark ML Modeling APIs to do preprocessing, modeling training and hyperparameter tuning using familiar scikit-learn-style APIs. Additionally, you can register your model for inference with the Snowflake Model Registry. The entire process is seamlessly managed, with Snowflake's engine running behind the scenes.

Sounds like a good deal, right? Our product documentation on Distributed Preprocessing indicated an impressive speed-up for the preprocessing class. Naturally, you'd expect similar enhancements in the model tuning phase. However, challenges

arise when dealing with larger data sets (exceeding 10GB) and an extensive hyperparameter search space. So, why is your hyperparameter tuning taking so long? Why does expanding your hyperparameter search space slow down the process? And why might you encounter an out-of-memory error with a 10GB data set?

Initially, we integrated directly with scikit-learn's GridSearchCV and RandomizedSearchCV tuning implementations, and off-loaded the computation to the Snowflake engine. Underneath the hood, we use a single node (**stored procedure**) to execute the tuning job. We used a stored procedure because the training process requires access to the full data set and can be repurposed for every function. However, while stored procedures are easier to implement, they are inherently single-node, regardless of the chosen warehouse size. This limitation means that even if a warehouse has eight nodes, not all will be used for this API

AUTHOR

Xinyi Jiang

Figure 1. *This diagram illustrates the process of fitting a machine learning model using Snowpark ML within a Snowflake environment. The process begins in a notebook environment, where a model, such as LinearRegression, is instantiated and trained using the fit method on a Snowpark DataFrame. The steps are as follows: (1) A temporary stage is created. (2) The unfitted underlying estimator is uploaded to this stage. (3) A temporary stored procedure for fitting is created and invoked. (4) Data from Snowflake tables is loaded into the fit stored procedure for training. (5) Once the training is complete, the fitted estimator is written back to the temporary stage. (6) Finally, the fitted estimator is downloaded back to the client environment, where it can be converted to a scikit-learn-compatible format using to_sklearn(). This figure highlights the seamless integration between Snowflake's processing power and a machine learning workflow, enabling efficient model training directly within the Snowflake environment.*

Ever striving to improve the efficiency of compute for our customers, we shortly thereafter started exploring how to

leverage all the nodes within the warehouse. If tuning on one node takes eight hours for a large data set, can we reduce the tuning time to one hour with eight nodes? To execute the tuning job in a multi-node fashion we process each hyperparameter combination as a table in a user-defined table function (UDTF). We made each of those GridSearchCV only accept one hyperparameter combination, disabling the parallelization backend. A Snowpark UDTF scheduler can handle the distribution of each worker and make the best use of the resources of your warehouse.

AUTHOR

Xinyi Jiang

SHARE

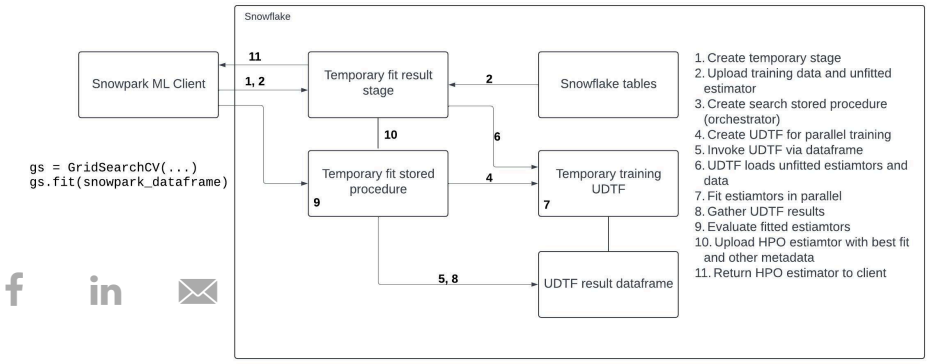


Figure 2. *This diagram illustrates the process of performing hyperparameter optimization using GridSearchCV within Snowflake ML. The steps are as follows: (1) The process begins in a notebook environment, where a GridSearchCV object is instantiated. (2) A temporary stage is created to hold both the training data and the unfitted estimator. (3) A search stored procedure is created, acting as the orchestrator for the search process. (4) A UDTF is created to facilitate parallel training across multiple configurations. (5) The UDTF is invoked via a DataFrame, which triggers the training process. (6) The UDTF loads the unfitted estimators and the data for parallel processing. (7) The models are fit in parallel, with each UDTF instance handling different hyperparameter combinations. (8) The results from the UDTF are gathered into a result DataFrame. (9) The fitted estimators are evaluated to determine the best model. (10) The best-performing hyperparameter combination, along with the estimator and other metadata, is uploaded back to the temporary stage. (11) Finally, the HPO estimator is returned to the client environment. This figure demonstrates the scalable approach to hyperparameter tuning within Snowflake, leveraging its parallel processing capabilities to efficiently search across a wide range of hyperparameter combinations.*

While this approach might seem reasonable, we quickly found that multi-node jobs became memory constrained. Astute readers will realize that this workload soon becomes memory-bound. Since we are executing multiple independent ‘fits’ per machine in parallel, we are loading the training data multiple times into each node. We observed that the distributed tuning API can only handle the data

sets up to 10GB. What happens above 10GB? Jobs see inefficiencies as a result of memory usage.

```
snowflake.snowpark.exceptions.SnowparkSQLExcep
tion: (1304): <<Your Query ID>>: 100387
(53200): Function available memory exhausted.
Please visit
https://docs.snowflake.com/en/developer-
guide/udf/python/udf-python-designing.ht
ml#memory for help.
```

AUTHOR

Xinyi Jiang

To enhance the memory efficiency of Snowflake's hyperparameter tuning jobs, we adjusted how Snowflake schedules distributed table function workloads, specifically for our hyperparameter tuning APIs in Snowflake ML. Previously, the UDTF scheduler would stack as many workers onto each node as possible. For our tuning use case, though, every worker in our UDTF requires loading the user's entire training data set. Thus, we modified the hyperparameter tuning scheduler such that each node generates only one UDTF worker. Furthermore, with the help of cachetools, we cache the data set loading results, ensuring each node loads the data set only once, saving on data loading time between sequential worker processes. Within each worker, we utilize joblib for parallel scheduling. Thus, we achieve a hybrid distribution model, where Snowflake ML's UDTF scheduler is responsible for distributing work across the nodes in the warehouse, and within each node we provide additional parallelization via joblib. Using this hybrid approach, all training data only needs to be loaded into each machine in the warehouse one time, without duplicating data replications or data load operations.

With the improvements to the scheduling and tuning algorithms, we have shown that instead of maxing out at ~10GB data sets, Snowflake users can run distributed tuning jobs using a Snowpark-Optimized Warehouse with data sets of 100 GB and beyond.

This results in massive improvements in tuning efficiency as we scale up the number of rows (Figure 3). On small data sets you

might not notice the difference, but as you hit 10+ million rows, you will see up to 250% improved compute usage.

AUTHOR

Xinyi Jiang

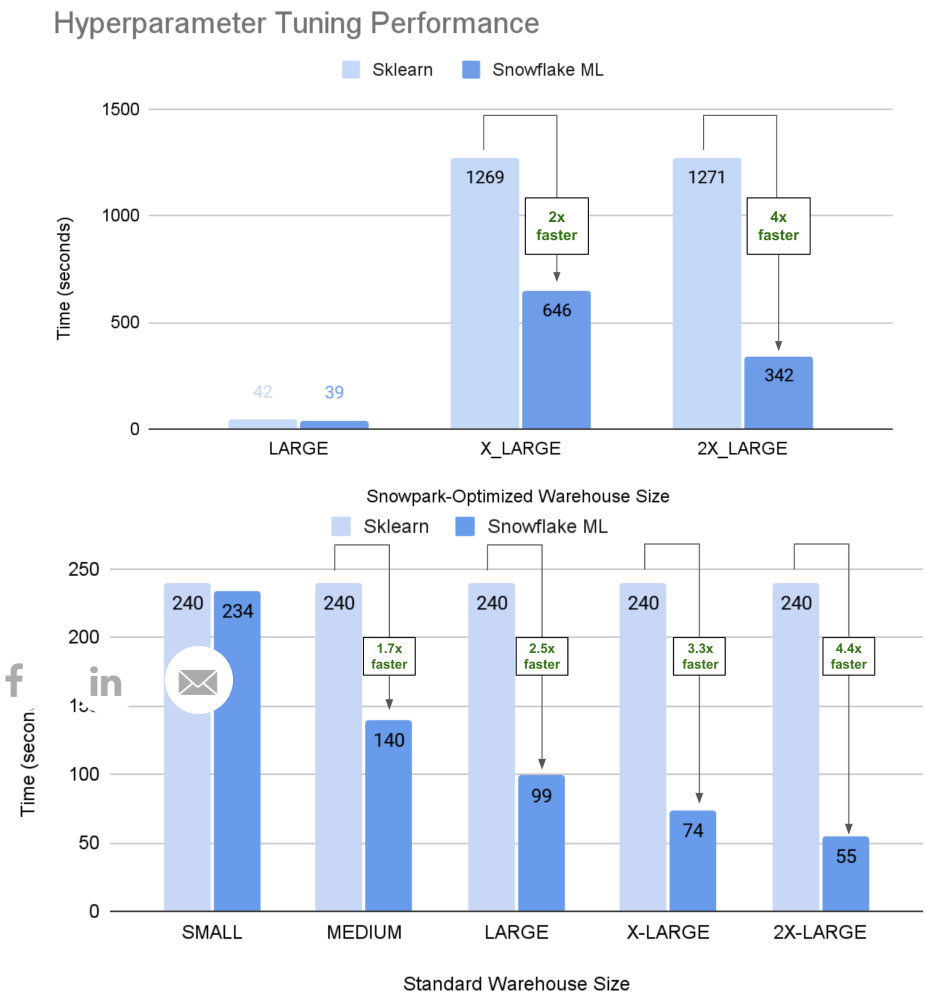


Figure 3. *Improvement of HPO efficiency by warehouse and table size with and without HPO memory optimization in both Snowpark-Optimized (top chart) and standard (bottom chart) warehouses.*

Conclusion

Implementing distributed hyperparameter optimization has led to significant improvements in model training efficiency, particularly when scaling from smaller warehouses to larger ones, such as the 2X-Large Snowflake warehouse. This approach has allowed us to leverage Snowflake's full potential, dramatically reducing processing times and enhancing overall performance. Distributed HPO in Snowflake ML is generally available for use today. The success of this implementation wouldn't have been possible without the dedicated support and involvement of everyone on

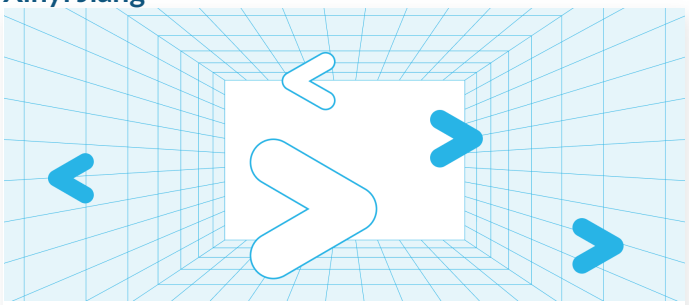
the team. Thank you all for your contributions and commitment to this project.

SHARE

AUTHOR

RELATED CONTENT

Xinyi Jiang



JUL 30, 2024

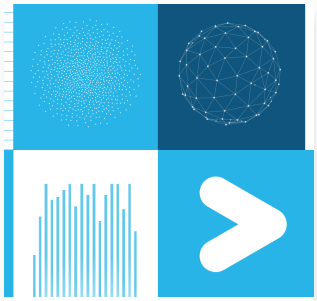
Adaptive Network Optimizations for Faster Query Performance

At Snowflake, we strive to deliver “automatic performance” to all our customers. This performance is driven through multiple areas of investment: hardware-level optimization, intelligent resource allocation, proactive storage optimization, adaptive...

JUN 17, 2024

Snowflake Arctic Cookbook Series: Instruction-Tuning Arctic

On April 24, we released Snowflake Arctic with a key goal in mind: to be...



Discover

More to follow

JUL 23, 2024

Achieve Low-Latency and High-Throughput Inference with Meta's Llama 3.1 405B using Snowflake's Optimized AI Stack

Meta's Llama 3.1 405B represents a groundbreaking milestone for open-weight large language models (LLMs), pushing...

Explore

AUTHOR

Xinyi Jiang

AUTHOR

Xinyi Jiang

SHARE