Model updates using SideInput

Visibility: Public

Authors: anandinguva@google.com
Contributors: Please add your name if you contribute to the doc.
Tracking GitHub Issue: https://github.com/apache/beam/issues/24042

Context

Objective

RunInference PTransform currently loads a single static model for the lifetime of the streaming pipeline. This doc presents a feature, where users can be able to update the model without requiring the use of pipeline lifecycle events.

Background

RunInference uses ModelHandler objects to load the model and perform the inference on those models. Users can pass a **model_path** to the ModelHandler and it takes care of loading the model during the lifecycle of the Pipeline. With the current implementation, once the model is loaded using ModelHandler, it would remain static for the entire pipeline. In a streaming pipeline, this could create friction to the users if the users want to update/change the model, they would have to replace the existing job with a new job. The streaming model updater would provide flexibility for the streaming pipeline to update the models used in RunInference PTransform without stopping or replacing the pipeline.

Design

Overview

To update the model running in the RunInference PTransform scope, the user can provide a side input PCollection, which can emit latest model metadata, to the RunInference transform.

The current implementation of the RunInference PTransform depends on **_RunInferenceDoFn.** In the DoFn, the model gets loaded in the setup method. For the dynamic updates of the model

in the streaming pipeline, the **_RunInferenceDoFn** should accept model updates via **side input**. As the DoFn **process** method gets the update to the model via side input, the _load_model method should be extended from the _RunInferenceDoFn **setup** method to the **process** method. Model caching is achieved between the instances of the DoFn with the help of the shared.

Accepting PCollection of model updates as side inputs

Similar to side input patterns, Users can pass a PCollection of model paths as side input to the RunInference transform for dynamic model updates. This allows users to update the model path via unbounded sources, such as PubSub. The model URI can be provided to the ModelHandler at the start of the pipeline, and updates can be passed in via an unbounded source. This feature can be used for different models in the pipeline, with each model having its own unbounded source.

Note: Update of the model using side inputs is non-deterministic.

Output type for the output emitted by the side input

It is recommended to use a NamedTuple `ModelMetadata` to wrap the model path in the following way:

```
# in base.py
class ModelMetadata(NamedTuple):
    model_path: str

# in user's example
model_path_side_pcoll >> beam.Map(lambda x: ModelMetadata(model_path=x)
```

By following this convention, it will ensure consistency and if there is a need to add additional parameters to the ModelMetadata in the future, it can be easily done with the help of the NamedTuple structure.

Use case: Watching a path for a new model update

For RunInference PTransform, the user will have the PTransform to watch over a path accessible by the pipeline for any model updates. A file glob pattern can be specified to the RunInference transform and the transform will take care of watching the pattern. In the RunInference transform, side input data is read periodically using PeriodicImpulse into the main PCollection windows by following the beam side input pattern.

- For example, A periodic impulse generating an infinite sequence, followed by a watch transform to the specified directory in the Global Window with an **AfterCount(1)** trigger. For every side input update, the main window PCollection gets the latest model path via side input update.

- MatchContinuously PTransform can be used to watch for new file updates. A new transform ***WatchFilePattern*** will be used on top of MatchContinuously transform.

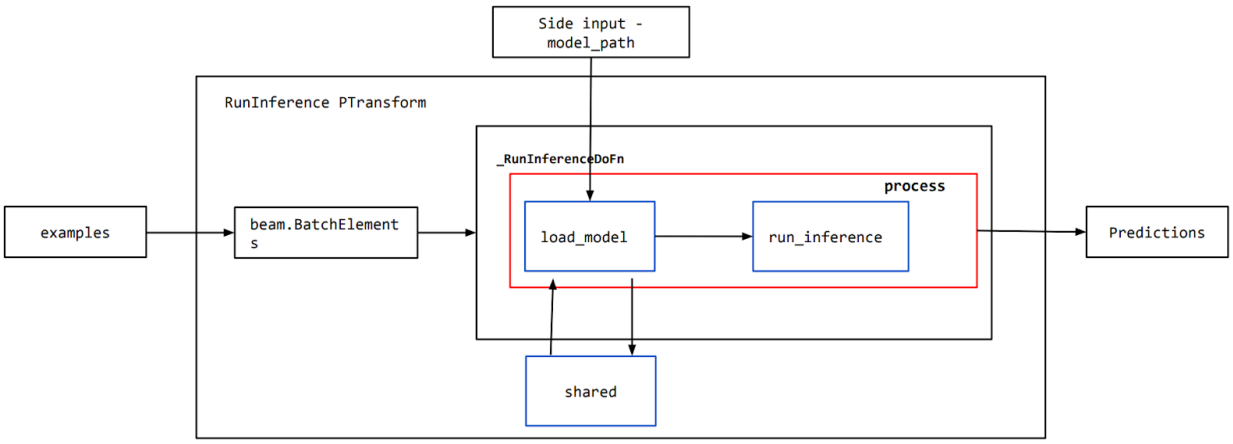
```
# user code
file_pattern = glob.glob('.../.../*.pt')
model_handler = ModelHandler(...,
side_input_pcoll=WatchFilePattern(file_pattern=file_pattern)
(pipeline | examples | base.RunInference(model_handler, side_input))

# in base.RunInference PTransform
 (examples |  beam.BatchElements() | beam.ParDo(_RunInferenceDoFn(...,),
side_input_pcoll if side_input_pcoll else None)
```

Note: MatchContinuously requires the use of Stateful DoFns to store the latest model path information.

Note: This transform will be only supported on ***streaming*** pipelines.

Model loading on the _RunInferenceDoFn instance

The loading of the model is called in the setup method of the DoFn and also in the process method of the DoFn when there is an update in the side input model path.



Side inputs to the _RunInferenceDoFn

Side input determines which model file is the latest file and this is passed to the RunInference DoFn process call. For every element of the process method, the model needs to be loaded and to avoid loading the model multiple times across the multiple DoFn instances, shared cache is used to maintain a weak-reference to the model's object.

Whenever there is an update to the model path, we update the shared along with a unique tag for the model. The **assumption** is that only one model will be cached in the shared as weak reference. If a new model is observed, the cached model and the cached model path will be

updated with their new values. The current caching mechanism with shared works efficiently when there aren't multiple models running at the same time.

Note: The streaming model update API expects the user to provide a ModelHandler with a valid path to the model/model weights.

Open questions

1. What happens to the in-flight process() calls of the DoFn's on the other threads?

- For a small amount of time, some threads might be running the DoFn process method with the old model while the new model is updated via thread locking mechanism in the shared cache. In this case, two models would live at the same time in memory. We should make the **assumption** that memory should be sufficient enough to be able to load 2 models per process for a single RunInference PTransform.
- Another alternative is to have a time.sleep(n) for n seconds. This could give the in-flight process calls to complete the inference on the old model.
    - This sleep will be called only once during loading the updated model.
        - It is possible to derive N from the previous model inference time.
- Using Read-Write locks.
- **Run the update_model and run_inference method under threading.Lock() context when there is an update to the side input.**
    - With a threading.Lock, only one thread can acquire the lock at a time and execute the critical section, whereas with a read-write lock, multiple threads can acquire the lock for reading, allowing the critical section to be executed simultaneously by multiple threads, while ensuring that the state remains consistent.
    - Using a threading.Lock, we have to lock the whole critical section and make sure that nobody writes to it while we read it, even though it would be safe to have multiple readers simultaneously.
    - As part of the first iteration, we would be using threading.Lock. Once it is out, read-write locks will be implemented.

Metric updates

With the model update, the current RunInference metrics should be changed for **metrics per model basis**.

If there is an update to the model, create a new **MetricsCollector** object for the updated model. Each model needs to have an unique identifier attached to the metrics so that differentiating among metrics for multiple models will be easy.

```
cached_object = self._shared_model_handle.acquire(load)
# Verify cached model path and current model_path acquired from side input
```

```
# are the same. If not, update the cache, and also instantiate a new
# MetricsCollector object for the new model.
if cached_object['model_path'] != model_path
      cached_object = self._shared_model_handle.acquire(load, tag=model_path)
      # TODO - add a unique tag to the metric namespace for each model.
      self._metrics_collector = _MetricsCollector(
          self._model_handler.get_metrics_namespace())
return cached_object
```

Error handling

The errors found during the runtime such as the updated model file is invalid or the example size doesn't match with model input's size. Streaming pipeline can have a side-output for errors and here we could have a code for error handling. With this approach, users can look up what the error is and can update the model otherwise, the error would be permanent and will result in stopping the pipeline.

- Errors can be marked with tag **errors** and will be emitted as a separate stream produced by the _RunInferenceDoFn.

```
examples | beam.ParDo(_RunInferenceDoFn(...).with_outputs('errors')
```

Adding a parameter to PredictionResult

An additional parameter named **model_id** will be added to the PredictionResult. The idea here is that the user can identify the model that was used to run a particular example.

[WIP]Testing :

For testing, A PubSub topic will be used as a source to read inputs.

- Streaming pipeline using RunInference Transform
  - On CPUs and, GPUs
    - Models of different sizes(500 MB, 1GB, 10 GB etc) will be used for the updates to make sure OOMs are not occurring.

Questions from comments:

1. [XQ] Provide a method `update_method` in the ModelHandler and let the user handle all the update related logic inside that method. This wouldn't require usage of side inputs and the user would be responsible for the update logic.
    - Pros on why using beam side inputs is useful
        i. It's easy to use our built in IOs in conjunction with model updates
        ii. It's easy to keep our usage of shared.py opaque to a user (users don't need to worry about efficiently loading models without killing their memory). You could get around this with a `should_update_model` function and then return the materialized model in `update_model`, but that seems non-optimal.
        iii. We can make future improvements to model loading that benefit everyone.

Design review notes - Jan 10, 2022

Design review notes on Side input model updates