Custom Inference Functions in RunInference Model Handlers

Jack McCluskey (jrmccluskey@apache.org)
Last updated: October 14th, 2022

Overview

This document aims to explore potential routes through which we can allow users to specify a custom model inference method for a given `RunInference` Model Handler, resolving the feature request found in issue #22572.

Goals

- Allow users to provide a custom inference function instead of the default option for each model handler
- Properly handle arguments for this custom function when called
- Be implemented in a way that compliments the effort to allow custom batching in `RunInference`

Background

The current implementations of `RunInference` model handlers assume that the user will always want to call a fixed prediction method for their mode. For PyTorch this is a forward pass using the `__call__` method, and for Sklearn this is the model's `predict` method; however, this is not always going to be the case for every model and user. For example, a number of pretrained models loaded with the HuggingFace transformers library recommend using `generate()` for inference. Providing a route through which users can specify a custom function for generating predictions enables easier use of alternative methods without writing a custom model handler, lowering the barrier to entry.

Design

Providing the Function

The three options considered for providing the alternative model inference function are as follows:

1. Take the function as an argument when initializing the model handler
2. Take the function as an explicit argument when calling `run_inference()`
3. Take the function as a `kwarg` with a constant key

Option 1 requires that the provided function be written in such a way that the model is not required to be serialized or initialized during pipeline construction. This adds a new argument to the model handler constructor like the following:

```
class SklearnModelHandlerNumpy(ModelHandler[numpy.ndarray,
                                            PredictionResult,
                                            BaseEstimator]):
  def __init__(
      self,
      model_uri: str,
      model_file_type: ModelFileType = ModelFileType.PICKLE,
      inference_fn: Optional[Callable] = default_numpy_inference_fn):
    """ Implementation of the ModelHandler interface for scikit-learn
    using numpy arrays as input.

    Example Usage::

      pcoll | RunInference(SklearnModelHandlerNumpy(model_uri="my_uri"))

    Args:
      model_uri: The URI to where the model is saved.
      model_file_type: The method of serialization of the argument.
        default=pickle
      inference_fn: The inference function to use.
        default=_default_numpy_inference_fn
    """
    self._model_uri = model_uri
    self._model_file_type = model_file_type
    self._model_inference_fn = inference_fn
```

This allows for some type checking on the function and a clear setting of a reasonable default inference function. Accepting the custom inference function at model handler construction-time also allows cross-language usage of this feature.

Option 2 alters the function signature for `run_inference()` to be:

```
def run_inference(
        self,
        batch: Sequence[ExampleT],
        model: ModelT,
        model_inference_fn: Optional[Callable] = None,
        inference_args: Optional[Dict[str, Any]] = None) -> Iterable[PredictionT]
```

This explicit form has the benefit of clarity and allows for clear documentation; however, it is a backwards-incompatible change that requires users to add a `None` value into their `run_inference()` calls even if they do not want to replace the default inference function. Avoiding this would have the `model_inference_fn` arg placed as the 4th argument; however, this placement feels clunky (providing the lambda after the arguments to it) and that pattern potentially leads to adding too many optional arguments to the `run_inference()` function over time. There is also the issue of default values not being able to be set explicitly in the function signature, for the same model-instantiation reasons above. This makes the implementation much clunkier in practice.

Option 3 does not change the `run_inference()` function signature and instead establishes a pattern for providing an alternative function in the `inference_args` dictionary. This establishes a standard for how model handlers should accept custom inference arguments moving forward while also avoiding code changes for users who are currently using the default options or custom model handlers to suit their own purposes. Option 3 also avoids the pitfalls of the other two options by allowing users to provide the function at `run_inference()` call-time when the model is already instantiated, allowing use of model-specific functions. Option 3 is a clear preferred option, as it aligns with the proposed standard for adding additional and/or optional parameters to `RunInference`. The only drawback is a loss of type checking, as the `inference_args` dictionary is `string -> any`.

Providing the Arguments

The simplest routing for providing the arguments to the alternative inference function is to leverage the existing `inference_args` argument that is then passed as a `kwargs` argument to the default function (see the PyTorch example.) This keeps the current `run_inference()` usage and experience intact for users and requires no changes to the function signature.

Value-Type and Handling

With the mechanics of *how* the alternative inference function is passed, the other half of the equation is *what* is passed. The first distinctions are between a model attribute name and a lambda. If we accepted the string representation of an attribute on the model, we would be able to call `getattr(model, name)` and get the new inference function that way, like the following:

```
# Define a constant that will always be respected as the key for a custom inference function
ALT_MODEL_INFERENCE_FN = "run_inference_alternative_inference_function"
```

```
def run_inference(
        self,
        batch: Sequence[ExampleT],
        model: ModelT,
        inference_args: Optional[Dict[str, Any]] = None) -> Iterable[PredictionT]:

        inference_args = {} if not inference_args else inference_args

        inference_fn_name = inference_args.get(ALT_MODEL_INFERENCE_FN,
                                               default_inference_fn_name)
        inference_fn = getattr(model, inference_fn_name)

        if not inference_fn:
        raise ValueError("Could not find model attribute with given name.")
       else:
 # Remove the extra arg so it isn't passed to the inference function
 inference_args.pop(ALT_MODEL_INFERENCE_FN)

        predictions = inference_fn(batch, **inference_args)
        return [PredictionResult(x,y) for x, y in zip(batch, predictions)]
```

This approach has the benefit of simplicity, taking a string argument and not requiring any loading of the model beforehand; however this limits the alternative inference function to being an attribute of the model rather than a generic function and could feel restrictive to advanced users. It is unclear how many potential users this would impact, though.

Alternatively, the argument could be passed as a lambda to be invoked when `run_inference` is called:

```
# Define a constant that will always be respected as the key for a custom inference function
ALT_MODEL_INFERENCE_FN = "run_inference_alternative_inference_function"

def run_inference(
        self,
        batch: Sequence[ExampleT],
        model: ModelT,
        inference_args: Optional[Dict[str, Any]] = None) -> Iterable[PredictionT]:

        inference_args = {} if not inference_args else inference_args

        inference_fn = inference_args.get(ALT_MODEL_INFERENCE_FN, None)
        if not inference_fn:
                inference_fn = default_inference_fn
        else:
```

```
                # Remove the extra arg so it isn't passed to the inference function
                inference_args.pop(ALT_MODEL_INFERENCE_FN)

        predictions = inference_fn(batch, **inference_args)
        return [PredictionResult(x,y) for x, y in zip(batch, predictions)]
```

This has the benefit of flexibility, allowing for any alternative inference function to be used, not just ones defined as attributes of the model class; however, if a user is seeking to use an inference function that is an attribute of the model this may require serializing the model by passing a bound function. For sufficiently large models, this is not ideal.

The other lambda-based approach is to add a level of indirection, instead making the routing for an alternative inference function route through a `Callable[[ModelT], Callable]` approach. This requires a little more boilerplate from a user in the form of a custom function; however, this allows for the use of general custom inference functions while avoiding the extra serialization step. This results in the following:

```
# Define a constant that will always be respected as the key for a custom inference function
ALT_MODEL_INFERENCE_FN = "run_inference_alternative_inference_function"

# Define a function that will produce the alternative inference fn at runtime
def custom_inference_fn_generator(model: ModelT) -> Callable:
        return model.alternate_fn

def run_inference(
        self,
        batch: Sequence[ExampleT],
        model: ModelT,
        inference_args: Optional[Dict[str, Any]] = None) -> Iterable[PredictionT]:

        inference_args = {} if not inference_args else inference_args

        inference_fn_generator = inference_args.get(ALT_MODEL_INFERENCE_FN, None)
        if not inference_fn:
                inference_fn = default_inference_fn
        else:
                inference_fn = inference_fn_generator(model)
                # Remove the extra arg so it isn't passed to the inference function
                inference_args.pop(ALT_MODEL_INFERENCE_FN)

        predictions = inference_fn(batch, **inference_args)
        return [PredictionResult(x,y) for x, y in zip(batch, predictions)]
```

The indirect lambda approach is arguably the most complex for a user to implement; however, the increase in complexity is relatively minor, gives some level of type checking with the generator function (albeit still limited

as an argument to `run_inference`,) and allows users to re-use the same model handler for different inference functions. These benefits all point towards this indirect approach being the preferred way for users to provide their custom inference functions.

Tweaking the form of the indirect function to be `function(model, batch, args) -> Iterable[Prediction]` allows for users to do more work within their function, having direct access to the model, batch, and arguments. This looks like the following:

```
# Define a constant that will always be respected as the key for a custom inference function
ALT_MODEL_INFERENCE_FN = "run_inference_alternative_inference_function"

# Define a custom inference fn that is executed at runtime
def custom_inference(model: ModelT, batch: Sequence[ExampleT],
                        inference_args: Optional[Dict[str, Any]] = None)
                        -> Iterable[PredictionT]:
        return model.alternate_fn(batch, **inference_args)

def run_inference(
        self,
        batch: Sequence[ExampleT],
        model: ModelT,
        inference_args: Optional[Dict[str, Any]] = None) -> Iterable[PredictionT]:

        inference_args = {} if not inference_args else inference_args

        custom_inference_fn = inference_args.get(ALT_MODEL_INFERENCE_FN, None)
        if not inference_fn:
                predictions = inference_fn(batch, **inference_args)
        else:
                # Remove the extra arg so it isn't passed to the inference function
                inference_args.pop(ALT_MODEL_INFERENCE_FN)
                predictions = custom_inference_fn(model, batch, inference_args)

        return [PredictionResult(x,y) for x, y in zip(batch, predictions)]
```

This removes one level of indirection (doing the inference directly instead of returning another function to call) and gives similar flexibility in terms of what can be done to customize their pipeline.

Implementation

The pairing of providing the custom inference function at model handler construction time and writing the “indirect” function returning the predictions is the best pairing of these options, providing methodology that gives users a large amount of flexibility while still having safeguards around type checking and large model serialization. This also keeps cross-language usage of `RunInference` up to par with native usage. This looks like the following when applied to the Sklearn numpy model handler:

```
def _default_numpy_inference_fn(
  model: BaseEstimator,
  batch: Sequence[numpy.ndarray],
  inference_args: Optional[Dict[str, Any]] = None) -> Any:
    # vectorize data for better performance
    vectorized_batch = numpy.stack(batch, axis=0)
    return model.predict(vectorized_batch)

class SklearnModelHandlerNumpy(ModelHandler[numpy.ndarray,
                                            PredictionResult,
                                            BaseEstimator]):
  def __init__(
      self,
      model_uri: str,
      model_file_type: ModelFileType = ModelFileType.PICKLE,
      inference_fn: Optional[Callable[[BaseEstimator, Sequence[numpy.ndarray],
                              Optional[Dict[str, Any]]], Any]] = _default_numpy_inference_fn):
    """ Implementation of the ModelHandler interface for scikit-learn
    using numpy arrays as input.

    Example Usage::

      pcoll | RunInference(SklearnModelHandlerNumpy(model_uri="my_uri"))

    Args:
      model_uri: The URI to where the model is saved.
      model_file_type: The method of serialization of the argument.
        default=pickle
      inference_fn: The inference function to use.
        default=_default_numpy_inference_fn
    """
    self._model_uri = model_uri
    self._model_file_type = model_file_type
    self._model_inference_fn = inference_fn

  def run_inference(
      self,
      batch: Sequence[numpy.ndarray],
      model: BaseEstimator,
      inference_args: Optional[Dict[str, Any]] = None
  ) -> Iterable[PredictionResult]:
    """Runs inferences on a batch of numpy arrays.

    Args:
      batch: A sequence of examples as numpy arrays. They should
        be single examples.
      model: A numpy model or pipeline. Must implement predict(X).
```

```
      Where the parameter X is a numpy array.
    inference_args: Any additional arguments for an inference.

  Returns:
    An Iterable of type PredictionResult.
  """
  predictions = self._model_inference_fn(model, batch, inference_args)

  return _convert_to_result(batch, predictions)
```

A draft pull request implementing this is available as #23642.