Hugging Face Model Handler for RunInference

Ritesh Ghorse (riteshghorse@apache.org)
Last Updated Jun 23, 2023

Reviewers: Danny Mccormick, Robert Bradshaw
Status: Final

Overview

Implement a Hugging Face Model Handler in Apache Beam Python SDK to download and load models from Hugging Face Models Hub by just specifying the model name and model class. We would build different model handlers from different input types like `tensors` and `numpy`. The model handler for `torch.Tensor` and `tensorflow.Tensor` will be the same. Using this model handler, users would be able to use `RunInference` transform by loading the models directly from the Hugging Face. This is a nice usability win in the current scenario, we need to download the model first and then use `TensorFlow` / `PyTorch` model handler for inference.

Goals

- Implement `HuggingFaceModelHandler` to load models from Hugging Face Models Hub.
- Support for `tf.Tensor`, `torch.Tensor`, and `numpy` input types.

Non-goals

- Hugging Face pipeline() support is not part of this work

Background

This document uses the RunInference API described in RunInference: ML Inference in Beam to support a new Hugging Face Model Handler. Currently, we have support for model handlers for `Tensorflow`, `PyTorch`, `Sklearn`, `TensorRT`, `XGBoost`, and `ONNX`.

In the current scenario, if a user wants to use a model from Hugging Face with RunInference transform they would first need to download the model from the hub manually. Next, they have to use either Tensorflow or PyTorch model handlers for RunInference.

```
# Download and save model from Hugging Face Hub.
```

```
model = AutoModelForSeq2SeqLM.from_pretrained(
        model_name, torch_dtype=torch.bfloat16
    )
torch.save(model.state_dict(), state_dict_path)


# Pass this saved model path to the PyTorchModelHandler for RunInference
# along with model class and other config parameters.
model_handler = PytorchModelHandlerTensor(
            state_dict_path=state_dict_path,
            model_class=AutoModelForSeq2SeqLM.from_config,
            model_params={"config": AutoConfig.from_pretrained(model_name)}
            )

predictions = RunInference(model_handler)
```

While using Hugging Face in a ML pipeline without beam it looks like:

```
# load the model
model = AutoModelForSeq2SeqLM.from_pretrained(model_name, torch_dtype=torch.bfloat16)
predictions = model(input)
```

We can clearly see the amount of extra steps that we need to do to use models from Hugging Face Hub in RunInference as compared to a normal Hugging Face pipeline. Implementation of a dedicated model handler for Hugging Face would bridge this gap and users can directly provide the name of the model they want to use from Hugging Face and rest will be taken care of by the model handler.

With a Hugging Face model handler for RunInference it would look like:

```
model_handler = HuggingFaceModelHandlerTensor(model_name=model_name,
                                              model_class=AutoModelForSeq2SeqLM,
                                              load_model_args={"torch_dtype": torch.bfloat16})
prediction = RunInference(model_handler)
```

which is a huge usability win.

Implementation Details

Proposal:

To use models from Hugging Face for any kind of inferences, we need at least two pieces of information - `model_name` from the [Hugging Face Models Hub]() and `base_model_class`.

For our model handler we will allow users to pass the `model_class` they want to use and the `model_name`. This is similar to what we already do in our [`PyTorchModelHandler`]() where we ask the user the path to saved weights and model class. This is because the `model_class` is architecture to do a specific task. There can be different models/checkpoints trained for that specific task which could be loaded into that architecture.

model_class here can be anything from transformers library like AutoModelForMultipleChoice and the model name is the repository id from the Hugging Face Hub.

We would then load the saved model from the model_name into the base class provided using the from_pretrained method provided by the Hugging Face.

```
base_class.from_pretrained(model_name)
```

Eg:
For a sequence classification task users could use AutoModelForSequenceClassification as the base class and bert-base-cased as the model_name. Now, in case of a custom checkpoint that a user may have for this model, they could instead specify that model name.

```
# loading pretrained model with bert-base-cased
model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased")

# loading custom checkpoint from Hugging Face Models Hub
model = AutoModelForSequenceClassification.from_pretrained("stevhliu/my_awesome_eli5_mlm_model")

# loading bert-base-cased checkpoint for masked language modeling
model = BertForMaskedLM.from_pretrained("bert-base-cased")

# loading image processor model with VisualBert
model = VisualBertModel.from_pretrained("uclanlp/visualbert-vqa-coco-pre")
```

Users could choose the kind of inference they want to do by providing the appropriate model class and the checkpoint for that model. It is possible to use model_class like VisualBertModel with different checkpoints/models available on Hugging Face Hub. To allow users to use it flexibly, we would need to get both the model_class and model_name on the hub from the users.

For the users it would looks like this:

```
model_handler = HuggingFaceModelHandlerTensor(model_name="repo/masked_language_model",
                                              model_class=AutoModelForMaskedLM)
prediction = RunInference(model_handler)
```

This is fairly simple to use and preserves the usability win that comes with HuggingFace.

We can allow additional kwargs for model specific config options users want to provide. These kwargs should be populated with the parameters of from_pretrained() from the Hugging Face.

Alternative 1:

Users will pass the repo-id and filename to download from the Models Hub, we can instead use the AutoModel classes of Transformer from Hugging Face. AutoModel for PyTorch and TFAutoModel for TensorFlow.

Users can then just pass the model name like `roberta-large` and we can do `TFAutoModel.from_pretrained(‘roberta-large’)` in our `load_model()` method.

This is a cleaner approach in terms of UI. The only cons is that we won’t be reusing existing model handlers.

For the users it would look like this

```
model_handler = HuggingFaceModelHandlerTensor(model_name="roberta-large")
```

In this approach, Hugging Face takes care of downloading and loading the model for inference, so you don't need to worry about `state_dict_path` for pytorch, `model_uri` for TensorFlow models, and other parameters required for their specific framework.

To use this approach, we will need to provide a Hugging Face auth token and the framework that we want to use. This is because Hugging Face uses different classes for PyTorch and TensorFlow models. However, we can avoid providing the framework argument if we only allow Tensors as input. In this case, Hugging Face will infer the framework from the type of Tensor - `torch.Tensor` and `tensorflow.Tensor`.

The alternative 1 seems more easy for users to use because they would have to remember less beam specific things and can mostly use the `kwargs` to provide Hugging Face specific args.

The disadvantage of using `AutoModel` is that when using `from_pretrained` it skips the `classifier` layer and `dropout`.

```
Some weights of the model checkpoint at stevhliu/my_awesome_eli5_mlm_model were not used when
initializing RobertaModel: ['lm_head.dense.weight', 'lm_head.dense.bias',
'lm_head.layer_norm.bias', 'lm_head.layer_norm.weight', 'lm_head.bias']
- This IS expected if you are initializing RobertaModel from the checkpoint of a model trained
on another task or with another architecture (e.g. initializing a BertForSequenceClassification
model from a BertForPreTraining model).
- This IS NOT expected if you are initializing RobertaModel from the checkpoint of a model that
you expect to be exactly identical (initializing a BertForSequenceClassification model from a
BertForSequenceClassification model).
```

Due to which, it gives the `last_hidden_state` and `pooler_ouput` tuple as the output (and other details like `attention_mask` for different kinds of models as configured).

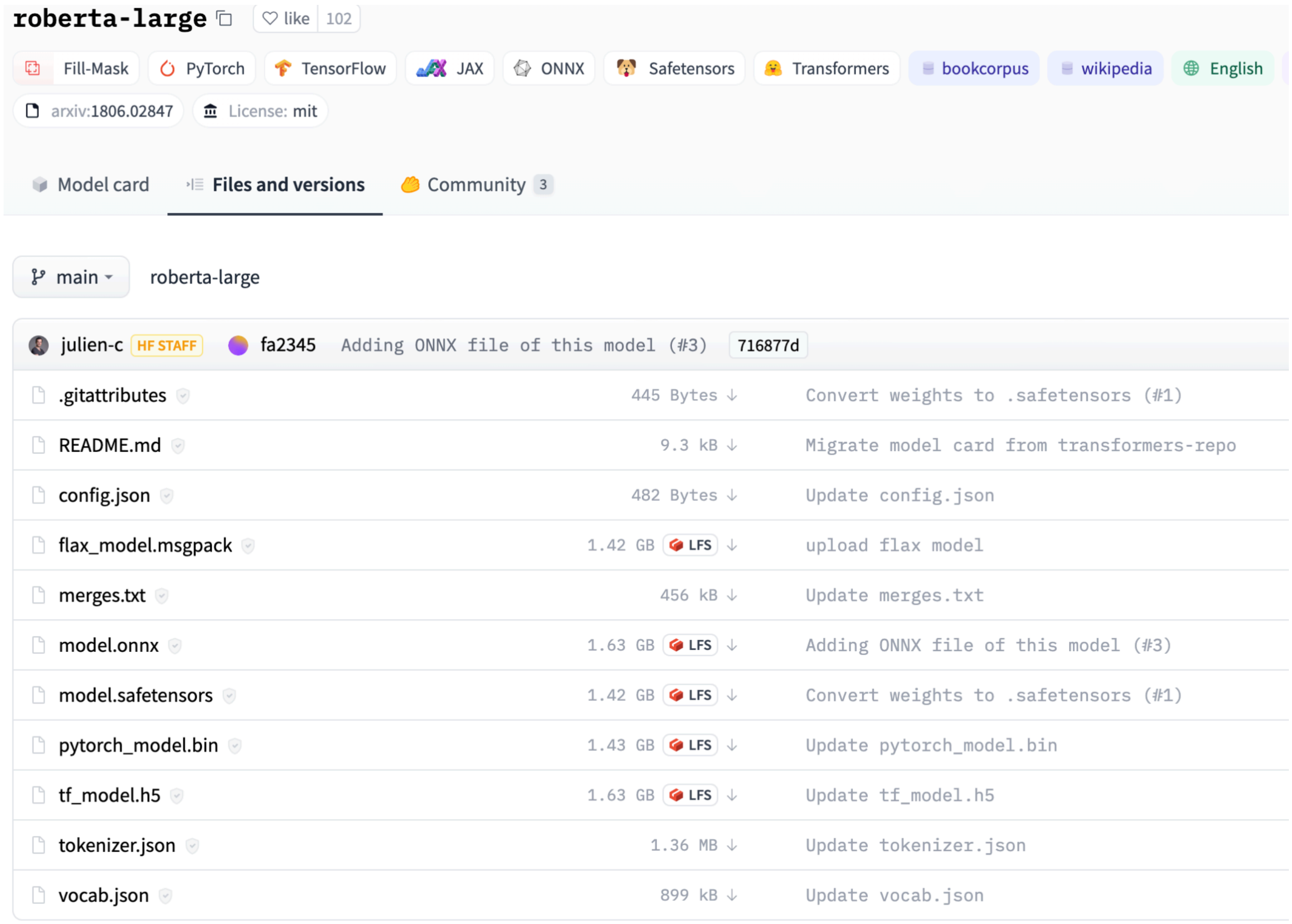
This means that users would have to write their own sort of output layer for the model which isn’t ideal and takes away the usability win of using Hugging Face in the first place.

Alternative 2:

Hugging Face model handlers make it easy for users to reuse pretrained models from the Hugging Face Models Hub. All the user needs to provide is the name of the repository and the file name for the pretrained model and an auth token.

We will use the `huggingface_hub` client library’s method `hf_hub_download()` to download the weight of models from the hub.

For example, let’s look at the roberta-large model on the hub:
It has a tab for `Files and versions` where you can see all the files for pretrained models.



julien-c HF STAFF fa2345 Adding ONNX file of this model (#3) 716877d

.gitattributes	445 Bytes	Convert weights to .safetensors (#1)
README.md	9.3 kB	Migrate model card from transformers-repo
config.json	482 Bytes	Update config.json
flax_model.msgpack	1.42 GB LFS	upload flax model
merges.txt	456 kB	Update merges.txt
model.onnx	1.63 GB LFS	Adding ONNX file of this model (#3)
model.safetensors	1.42 GB LFS	Convert weights to .safetensors (#1)
pytorch_model.bin	1.43 GB LFS	Update pytorch_model.bin
tf_model.h5	1.63 GB LFS	Update tf_model.h5
tokenizer.json	1.36 MB	Update tokenizer.json
vocab.json	899 kB	Update vocab.json

In this case, the repository id is `roberta-large` and the filename would be `tf_model.h5` for TensorFlow model.

The `HuggingFaceModelHandler` will have a class variable that will be instantiated with the inferred framework’s model handler.

But the problem with this approach is that each model handler have different input parameter requirements - for example, in `TFModelHandlerNumpy`, we need `model_uri, create_model_fn, and ModelType` in one of the ways of loading model while `PyTorchModelHandlerTensor` needs an `uri` and `model_class`. There are other optional parameters as well which may differ. This makes it difficult for the `HuggingFaceModelHandler` to generalize different kinds of input to be passed to the underlying model handler.

To resolve this we have two options:

1. Use a class-like object as a parameter to the `HuggingFaceModelHandler` which defines the framework specific details. For example, `PytorchModelHandlerTensor` has `state_dict_path`, `model_class`, `model_params` as inputs with other optional types. We can bundle them up in a single class like

```
class PyTorchMetadata(NamedTuple):
  model_class: torch.Module
  model_params: Dict[str, Any]
  …
  …
```

The advantage of this approach is that the users can know the details they would like to provide by doing `PyTorchMetadata.model_class`.
The cons of this approach is that we are introducing additional data structure for users to look at.

2. Use a `kwargs` dictionary similar to what we use in `load_model` and `run_inference` methods.
   In this approach, users would have to look at the desired framework's model handler and pass in the required parameters except the one where they need to pass the path to model weights.

In addition to this, Hugging Face auth token needs to be passed when required.

Extension to Base Model Handler

Hugging Face Pipeline Support

Hugging Face Pipelines are a higher level abstraction of a series of steps - tokenizing, loading the model, running inference, de-tokenizing - for specific tasks. It enables users to do inference in just a few lines.

Example: To do a text-classification task, we just need to do

```
>> from transformers import pipeline
>> model = pipeline("text-classification")
>> model("It is a good day")
[{'label': 'POSITIVE', 'score': 0.99}]
```

In the above case, the pipeline used a default model for text-classification task defined in the Hugging Face. To specify a different model, it looks like

```
model = pipeline(task="text-classification", model="roberta-large-mnli")
```

Some models on Hugging Face Models Hub already define the task that they are trained for. In such cases there is no need to specify the task separately. You can just write it as

```
model = pipeline(model="roberta-large-mnli")
```

To create a model handler for pipeline support, we would need either of these two - `model` or `task` to load the model. In addition to this, there are other optional parameters like `config`, `tokenizer`, `framework`, etc which could be passed to the `pipeline()`. We can pipe these through using the `load_pipeline_args` dictionary. We will need to validate that either a task or model is passed to the class while instantiating.

`pipeline()` return the model instance which can be returned from the `load_model()` function of the model handler.

The input type for the model handler would be just string as it is for usual Hugging Face Pipelines usage. String input works nicely for text related tasks. For images and audio related tasks, users can just pass a string path to the files and the pipeline wrapper takes care of the rest of the processing.

Examples for Image Segmentation task with Hugging Face Pipelines:

```
classifier = pipeline(model="facebook/detr-resnet-50")
output = classifier("gs://gcs-bucket/bird.png")
```

Similarly, we could provide a path to the audio file to do audio related tasks.

We can specify the framework to use by using the `load_pipeline_args` dict. `pipeline()` provides a parameter for specifying it - `framework: ‘pt’ for PyTorch or ‘tf’ for TensorFlow`. If nothing is specified, it will default to the one that is installed. If both are installed then it will default to the framework of the model or to PyTorch if no model is specified.

We could write a model handler for this as

```
class HuggingFacePipelineModelHandler(ModelHandler[str,
                                                   PredictionResult,
                                                   Pipeline]):

  def __init__(
      self,
      task: str = None,
      model=None,
      *,
      inference_fn: Optional[Callable[..., Iterable[PredictionT]]] = None,
      load_pipeline_args: Optional[Dict[str, Any]] = None,
      inference_args: Optional[Dict[str, Any]] = None,
      min_batch_size: Optional[int] = None,
      max_batch_size: Optional[int] = None,
      large_model: bool = False,
      **kwargs):
        …
        …
```

With the approach, the `RunInference` call looks like

```
model_handler = HuggingFacePipelineModelHandler(task="text-classification",
   load_pipeline_args={'framework': 'pt'})

output = pipeline
                  | beam.Create(["It is a good day"])
                  | RunInference(model_handler)
```

ModelOutput Return Type

Hugging Face models output the prediction in the form of `ModelOutput` class. It is just `OrderedDict` that defines various output values like `loss`, `score`, etc.
For example, in case of a masked language model

The Hugging Face pipeline is -

```
model = AutoModelForMaskedLM.from_pretrained("stevhliu/my_awesome_eli5_mlm_model")
prediction = model(**input)
```

with prediction as

```
MaskedLMOutput(loss=None, logits=tensor([[[-0.1248,  0.0000,  0.0524,  ..., -0.1736, -0.1838,
-0.3950],
         [ 0.6857,  0.0000,  0.3641,  ..., -0.3146,  0.6410,  0.8473],
         [-0.5218,  0.0000, -0.1220,  ..., -0.1641, -0.0437, -0.1125],
         ...,
         [-0.3393,  0.0000, -0.1739,  ..., -0.3465,  0.5032,  0.4917],
         [ 0.4458,  0.0000, -0.1290,  ...,  0.5043,  0.2980,  0.2150],
         [-0.4792,  0.0000,  0.7597,  ..., -0.5056, -0.6125,  0.7951]]],
       grad_fn=<ViewBackward0>), hidden_states=None, attentions=None)
```

With our model handler,

```
model_handler = HuggingFaceModelHandlerTensor(model_uri="stevhliu/my_awesome_eli5_mlm_model",
                                              model_class=AutoModelForMaskedLM)

prediction = (pipeline | beam.Create(["The capital of France is Paris."])
                       | RunInference(model_handler)
```

the `ModelOutput` type prediction is assigned to the inference field of `PredictionResult` as

```
PredictionResult(

example={'input_ids': tensor([    0,   970,    32,   195, 29472,   261,  7046,    15,    42,
14262,
        50264,     4,     2]), 'attention_mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1])},

inference=MaskedLMOutput(loss=None, logits=tensor([[[20.4482, -4.8258, 17.9683,  ..., -0.2959,
1.8380,  9.0381],
         [ 3.2331, -5.4753, 16.6446,  ..., -0.7833, -0.8888,  2.2015],
         [-1.1801, -5.6766,  9.9048,  ...,  1.3851, -3.0148,  0.2140],
         ...,
         [-5.1801, -5.1109,  5.8191,  ..., -5.9792, -4.8173,  0.6445],
         [ 8.8481, -5.5120, 17.3786,  ..., -2.0276, -0.2673,  2.8984],
         [ 5.7127, -5.6760, 25.9026,  ..., -3.1510, -4.7130,  4.8339]]]), hidden_states=None,
attentions=None),

model_id=None)
```

We already have a return type `PredictionResult` in place for our model handlers. We could just wrap the model output object to the inference field of `PredictionResult`. Having a specific output type for

`HuggingFaceModelHandler` is not adding too much additional value. Most importantly, it aligns with all our model handlers as well.

Mass Model Download

In general, there is no explicit rate-limiting specified for downloading the models from Hugging Face Models. But for the Inference API, there is a limit of 10k requests suddenly. The time frame for the rate is not clearly defined here.

Hugging Face provides a way to save the loaded models locally using `save_pretrained()`