Exporting a pre-trained Encoder to TF Hub

Overview

This doc explains how to use TF-NLP's export_tfhub tool to export pre-trained Transformer encoders to SavedModels suitable for publication on TF Hub. (For the steps after that, see TF Hub's publisher guide.) For testing purposes, those SavedModels can also be used from their export locations on the filesystem.

On TF Hub, Transformer encoders for text come as a pair of SavedModels:

- The preprocessing model applies a tokenizer with a fixed vocab plus some additional logic to turn text into Transformer inputs.
- The encoder model (or "model" for short) applies the pre-trained Transformer encoder.

TF Hub defines Common APIs for all SavedModels of those two respective types, encapsulating the particular choice of preprocessing logic and Encoder architecture.

Exporting the Encoder

There is a choice between exporting just the encoder, or the encoder plus the prediction head for the masked language model (MLM) task from pre-training.

Exporting just the encoder suffices for many straightforward applications.

Exporting the Encoder alone

To export an encoder-only model, you can set `--export_type=model` and run the tool like this:

```
python official/nlp/tools/export_tfhub.py \
  --encoder_config_file=${BERT_DIR:?}/bert_encoder.yaml \
  --model_checkpoint_path=${BERT_DIR:?}/bert_model.ckpt \
  --vocab_file=${BERT_DIR:?}/vocab.txt \
  --export_type=model \
  --export_path=/tmp/bert_model
```

The flag `--encoder_config_file` refers to a YAML file representing the encoders.EncoderConfig dataclass, which supports multiple encoders (e.g., BERT, ALBERT). Instead of `--encoder_config_file`, you can set `--bert_config_file` to a legacy `bert_config.json` file to export a BERT model. If the model definition involves GIN, the flags `--gin_file` and `--gin_params` must be set accordingly, consistent with pre-training.

The `--model_checkpoint_path` refers to an object-based (TF2) checkpoint written by BertPretrainerV2, or any other checkpoint that can be restored to `tf.train.Checkpoint(encoder=encoder)` for the encoder defined by the config flags. Legacy checkpoints with `model=` instead of `encoder=` are also supported for now.

The exported SavedModel expects dict inputs and outputs as follows, implementing a specialization of the respective Common SavedModel API:

```
encoder = hub.load(...)
encoder_inputs = dict(
    input_word_ids=...,  # Shape [batch, seq_length], dtype=int32
    input_mask=...,      # Shape [batch, seq_length], dtype=int32
    input_type_ids=...,  # Shape [batch, seq_length], dtype=int32
)
encoder_outputs = encoder(encoder_inputs)
assert encoder_outputs.keys() == {
  "pooled_output",    # Shape [batch_size, width], dtype=float32
  "default",          # Alias for "pooled_output" (aligns with other models)
  "sequence_output",  # Shape [batch_size, seq_length, width], dtype=float32
  "encoder_outputs",  # List of Tensors with outputs of all transformer layers
}
```

The encoder's pooler layer is restored from the `--model_checkpoint_path`. However, unlike classic BERT, `BertPretrainerV2` does not train the pooler layer of the encoder. You have three options to handle that:

- Set flag `--copy_pooler_dense_to_encoder` to copy the pooling layer from the `ClassificationHead` passed to `BertPretrainerV2` for the next sentence prediction task. This mimicks classic BERT, but is not recommended for new models (see next item).
- Leave flag `--copy_pooler_dense_to_encoder` unset and export the untrained, randomly initialized pooling layer of the encoder. Folklore (as of 2020) has it that an untrained pooler gets fine-tuned better than a pre-trained pooler, so this is the default.
- Leave flag `--copy_pooler_dense_to_encoder` unset and perform your own initialization of the pooling layer before export. For example, Google's BERT Experts published in October 2020 initialize it to the identity map, reporting equal gains if fine-tuning, and more predictable behavior if not.

In any case, at this time, the export tool requires the encoder model to *have* a `pooled_output`, whether trained or not. (This can be revised in the future.)

The encoder model does not include any preprocessing logic, but for the benefit of users who take preprocessing into their own hands, the relevant information is attached from flags `--vocab_file` or `--sp_model_file`, resp., and `--do_lower_case`, which need to be set in exactly the same way as for the preprocessing model (see below).

The root object of the exported SavedModel stores the resulting values as attributes on the root object:

```
encoder = hub.load(...)
```

```
# Gets the filename of the respective tf.saved_model.Asset object.
if hasattr(encoder, "vocab_file"):
  print("Wordpiece vocab at", encoder.vocab_file.asset_path.numpy())
elif hasattr(encoder, "sp_model_file"):
  print("SentencePiece model at", encoder.sp_model_file.asset_path.numpy())
# Gets the value of a scalar bool tf.Variable.
print("...using do_lower_case =", encoder.do_lower_case.numpy())
```

New users are encouraged to ignore these attributes and use the preprocessing model instead. However, there are legacy users, and advanced users that require access to the full vocab.

Exporting the Encoder with a Masked Language Model head

To export an encoder and the masked language model it was trained with, first read the preceding section about exporting just the encoder. All the explanations there on setting the right flags apply here as well, up to the following differences.

The masked language model is added to the export by changing flag `--export_type` from `model` to `model_with_mlm`, so the export command looks like this:

```
python official/nlp/tools/export_tfhub.py \
  --encoder_config_file=${BERT_DIR:?}/bert_encoder.yaml \
  --model_checkpoint_path=${BERT_DIR:?}/bert_model.ckpt \
  --vocab_file=${BERT_DIR:?}/vocab.txt \
  --export_type=model_with_mlm \
  --export_path=/tmp/bert_model
```

The `--model_checkpoint_path` refers to an object-based (TF2) checkpoint written by BertPretrainerV2, or any other checkpoint that can be restored to `tf.train.Checkpoint(**BertPretrainerV2(...).checkpoint_items)` with the encoder defined by the config flags.

This is a more comprehensive requirement on the checkpoint than for `--export_type=model`; not all Transformer encoders and not all pre-training techniques can satisfy it. For example, ELECTRA uses the BERT architecture but is pre-trained without an MLM task.

The root object of the exported SavedModel is called in the same way as above. In addition, the SavedModel has an `mlm` subobject that can be called as follows to output an `mlm_logits` tensor as well:

```
mlm_inputs = dict(
    input_word_ids=...,       # Shape [batch, seq_length], dtype=int32
    input_mask=...,           # Shape [batch, seq_length], dtype=int32
    input_type_ids=...,       # Shape [batch, seq_length], dtype=int32
    masked_lm_positions=...,  # Shape [batch, num_predictions], dtype=int32
)
```

```
mlm_outputs = encoder.mlm(mlm_inputs)
assert mlm_outputs.keys() == {
  "pooled_output",   # Shape [batch, width], dtype=float32
  "sequence_output", # Shape [batch, seq_length, width], dtype=float32
  "encoder_outputs", # List of Tensors with outputs of all transformer layers
  "mlm_logits"       # Shape [batch, num_predictions, vocab_size], dtype=float32
}
```

The extra subobject imposes a moderate size overhead.

Exporting from a TF1 BERT checkpoint

A BERT model trained with the original BERT implementation for TF1 can be exported after converting its checkpoint with the tf2_encoder_checkpoint_converter tool.

After that, run export_tfhub per the instructions above on the converted checkpoint. Do not set `--copy_pooler_dense_to_encoder`, because the pooler layer is part of the converted encoder. For `--vocab_file` and `--do_lower_case`, the values from TF1 BERT can be used verbatim.

Exporting the preprocessing model

You can skip this step if TF Hub already has a preprocessing model that does exactly what your encoder needs (same tokenizer, same vocab, same normalization settings (`do_lower_case`)). You can inspect its collection of Transformer Encoders for Text and click through to models with a similar input domain to find their preprocessing models.

To export the preprocessing model, set `--export_type=preprocessing` and run the export tool like this:

```
python official/nlp/tools/export_tfhub.py \
  --vocab_file=${BERT_DIR:?}/vocab.txt \
  --do_lower_case=True \
  --export_type=preprocessing \
  --export_path=/tmp/bert_preprocessing
```

Note: Set flag `--experimental_disable_assert_in_preprocessing` when exporting to users of the public TensorFlow releases 2.4.x to avoid a fatal ops placement issue when preprocessing is used within Dataset.map() on TPU workers. This is not an issue with TF2.3 and TF2.5+.

Flag `--vocab_file` specifies the vocab file used with BertTokenizer. For models that use the SentencepieceTokenizer, set flag `--sp_model_file` instead.

The boolean flag `--do_lower_case` controls text normalization (as in the respective tokenizer classes, so it's a bit more than just smashing case). If unset, do_lower_case will be enabled if 'uncased' appears in –vocab_file, or unconditionally if –sp_model_file is set, mimicking the conventions of BERT and

ALBERT, respectively. For programmatic use, or if in doubt, it's best to set `--do_lower_case` explicity.

If the definition of preprocessing involved GIN, the flags `--gin_file` and `--gin_params` would have to be set accordingly, consistent with pre-training. (At the time of this writing, no such GIN configurables exist in the code.)

The exported SavedModel can be called in the following way for a single segment input.

```
preprocessor = hub.load(...)
text_input = ... # Shape [batch_size], dtype=tf.string
encoder_inputs = preprocessor(text_input, seq_length=seq_length)
assert encoder_inputs.keys() == {
  "input_word_ids", # Shape [batch_size, seq_length], dtype=int32
  "input_mask",     # Shape [batch_size, seq_length], dtype=int32
  "input_type_ids"  # Shape [batch_size, seq_length], dtype=int32
}
```

Flag `--default_seq_length` controls the value of `seq_length` if that argument is omitted in the usage example above. The flag defaults to 128, because mutiples of 128 work best for Cloud TPUs, yet the cost of attention computation grows quadratically with `seq_length`.

Beyond this example, the exported SavedModel implements the full set interface from the preprocessor API for text embeddings with preprocessed inputs and with Transformer encoders from TF Hub's Common APIs for text.

Please see tfhub.dev/tensorflow/bert_en_uncased_preprocess for the full documentation of one preprocessing model exported with this tool, especially how custom trimming of inputs can happen between `.tokenize` and `.bert_pack_inputs`.

Using the `encoder.mlm()` interface requires masking of tokenized inputs by user code. The necessary information on the vocabulary encapsulated in the preprocessing model can be obtained like this (uniformly across tokenizers):

```
special_tokens_dict = preprocess.tokenize.get_special_tokens_dict()
vocab_size = int(special_tokens_dict["vocab_size"])
padding_id = int(special_tokens_dict["padding_id"])  # [PAD] or <pad>
start_of_sequence_id = int(special_tokens_dict["start_of_sequence_id"])  # [CLS]
end_of_segment_id = int(special_tokens_dict["end_of_segment_id"])  # [SEP]
mask_id = int(special_tokens_dict["mask_id"])  # [MASK]
```

Testing the exported models

Please test your SavedModels before publication by fine-tuning them on a suitable task and comparing performance and accuracy to a baseline experiment built from equivalent Python code. The trainer doc has instructions how to run BERT on MNLI and other tasks from the GLUE benchmark.