Contents

1 Transformer Small

1.1 Data Generation

1.1.1 vocabulary & tokenization

$\text{vocab}_{\text{type}}$: (`text_problems.py vocab_type()`)

‘VocabType‘s:

- ‘SUBWORD‘: ‘SubwordTextEncoder‘, an invertible wordpiece vocabulary. Must provide ‘$\text{self.approx}_{\text{vocabsize}}$‘. Generates the vocabulary based on the training data. To limit the number of samples the vocab generation looks at, override ‘self.max$_{\text{samplesforvocab}}$‘. Recommended and default.

- ‘CHARACTER‘: ‘ByteTextEncoder‘, encode raw bytes.
- ‘TOKEN‘: ‘TokenTextEncoder‘, vocabulary based on a file. Must provide a vocabulary file yourself (‘TokenTextEncoder.store$_{\text{tofile}}$‘) because one will not be generated for you. The vocab file should be stored in ‘data$_{\text{dir}}$/‘ with the name specified by ‘self.vocab$_{\text{filename}}$‘.

1.1.2 Registry Gotcha

- github: `https://github.com/tensorflow/tensor2tensor/blob/master/docs/new_problem.md`
- blog: `https://cloud.google.com/blog/big-data/2018/02/cloud-poetry-training-and-hyp`
- tutorial: `https://github.com/GoogleCloudPlatform/training-data-analyst/blob/master/courses/machine_learning/deepdive/09_sequence/poetry.ipynb`

You can test data generation of your a problem in your own project with:

```
PROBLEM=poetry_line_problem
DATA_DIR=$HOME/t2t_data
TMP_DIR=/tmp/t2t_datagen
mkdir -p $DATA_DIR $TMP_DIR

t2t-datagen \
  --t2t_usr_dir=$PATH_TO_YOUR_PROBLEM_DIR \
  --data_dir=$DATA_DIR \
  --tmp_dir=$TMP_DIR \
  --problem=$PROBLEM
```

```
@registry.register_problem
class PoetryLineProblem(text_problems.Text2TextProblem):
    pass
```

Where:

- The Python Filename is `snake_case` of problem name
- `PROBLEM` is the name of the class that was registered with `@registry.register_problem()`, but converted from `CamelCase` to `snake_case`

1.1.3 MISC

1.2 Problems base classes

- blog: `https://cloud.google.com/blog/big-data/2018/02/cloud-poetry-training-and-hyp`
- Models that use a sequence as an input, but are essentially classifiers or regressors—a spam filter or sentiment identifier are canonical examples of such a model.
- Models that take a single entity as input but produce a text sequence as output—image captioning is an example of such a model, since given an image, the model needs to produce a sequence of words that describes the image.
- Models that take sequences as input and produce sequences as outputs. Language translation, question-answering, and text summarization are all examples of this third type.

1.3 Modality (Not Interfaces)

1.3.1 Definition

`utils/modality.py`

> An abstract class representing modalities for transforming data to a space interpretable by T2T models. It has 4 functions:
>
> - bottom: called on inputs entering the model.
> - $\text{targets}_{\text{bottom}}$: called on targets entering the model (e.g., the decoder).
> - top: called on model outputs to generate predictions (e.g., logits).
> - loss: called on predictions (outputs of top) and targets.
>
> For example, think about a modality for images:
>
> - 'bottom' represents the part of the model applied to an incoming image, e.g., an entry flow of a convolutional network.
> - 'top' represents the top part of a model that is generating images, e.g., a PixelCNN network.

- 'targets$_{\text{bottom}}$' represents the auto-regressive part of the network. It is applied to the already-generated part of an image, which is given to the decoder to generate the next part. In some cases, e.g., for text, it is the same as the 'bottom' function, and that is the default we use. But, e.g., for images, a different function might be needed to regress properly.
- 'loss' would compare the generated image to the target image and score it.

All the functions have simple and sharded versions. A sub-class only needs to implement the simple version, the default sharding will be used then.

1.3.2 Usage Example

In `SubclassProblem` `def hparams():` (`text_problems.py` `Text2TextProblem` as an example)

```
def hparams(self, defaults, unused_model_hparams):
  p = defaults
  p.stop_at_eos = int(True)

  if self.has_inputs:
    source_vocab_size = self._encoders["inputs"].vocab_size
    p.input_modality = {
        "inputs": (registry.Modalities.SYMBOL, source_vocab_size)
    }
  target_vocab_size = self._encoders["targets"].vocab_size
  p.target_modality = (registry.Modalities.SYMBOL, target_vocab_size)
  if self.vocab_type == VocabType.CHARACTER:
    p.loss_multiplier = 2.0

  if self.packed_length:
    identity = (registry.Modalities.GENERIC, None)
    if self.has_inputs:
      p.input_modality["inputs_segmentation"] = identity
      p.input_modality["inputs_position"] = identity
    p.input_modality["targets_segmentation"] = identity
    p.input_modality["targets_position"] = identity
```

1.4 Input Pipeline

1.4.1 batch generation

`https://github.com/tensorflow/tensor2tensor/blob/master/docs/overview.md#batching`

Variable length Problems are bucketed by sequence length and then batched out of those buckets. This significantly improves performance over a naive batching scheme for variable length sequences because each example in a batch must be padded to match the example with the maximum length in the batch.

1. Implementation

    - In `data_generators/problems.py` `Problem.input_fn()`

    ```
    # sg: GPU batch size / buckets are generated here
    dataset = data_reader.bucket_by_sequence_length(
        dataset, data_reader.example_length, batching_scheme["boundaries"],
        batching_scheme["batch_sizes"])
    # bucket_by_sequence_length using tf.contrib.data.group_by_window()
    ```

    `https://www.tensorflow.org/api_docs/python/tf/contrib/data/group_by_window`

    A transformation that groups windows of elements by key and reduces them.

    This transformation maps each consecutive element in a dataset to a key using key_{func} and groups the elements by key. It then applies $\text{reduce}_{\text{func}}$ to at most $\text{window}_{\text{sizefunc}}(\text{key})$ elements matching the same key. All execpt the final window for each key will contain $\text{window}_{\text{sizefunc}}(\text{key})$ elements; the final window may be smaller.

1.5 T2TModel

1.5.1 Overview

`top()` `body()` `bottom()` `loss()`

- `body()` should be overridden when subclass `T2TModel`
- `top()` `bottom()` `loss()` should be overridden when declaring new `Modality` in `layers/modalities.py`

From https://github.com/tensorflow/tensor2tensor/blob/master/docs/overview.md#building-the-model

> At this point, the input features typically have "inputs" and "targets", each of which is a batched 4-D Tensor (e.g. of shape [batch_size, sequence_length, 1, 1] for text input or [batch_size, height, width, 3] for image input).
>
> The Estimator model function is created by T2TModel.estimator_model_fn, which may be overridden in its entirety by subclasses if desired. Typically, subclasses only override T2TModel.body.

- estimator_model_fn is a @classmethod function, which is used as an override of the original constructor. This acts like a factory function return subclass instances of T2TModel class

> The model function constructs a T2TModel, calls it, and then calls T2TModel.{estimator_spec_train, estimator_spec_eval, estimator_spec_predict} depending on the mode.
>
> A call of a T2TModel internally calls bottom, body, top, and loss, all of which can be overridden by subclasses (typically only body is).
>
> The default implementations of bottom, top, and loss depend on the Modality specified for the input and target features (e.g. SymbolModality.bottom embeds integer tokens and SymbolModality.loss is softmax_cross_entropy).

1.5.2 hparams

When subclass T2TModel, there are many hparams defined in layers/common_hparams.py. transformer.py initiated basic_params1() in that file.

hparams can also be overridden in subclasses.

1.5.3 Decoder Notes

- has_input=False, no encoder

1. Pre & Post Process

   For example, if sequence=="dna", then the output is previous_value + normalize(dropout(x))

```
hparams.layer_preprocess_sequence = "n"
# normalize(x)
hparams.layer_postprocess_sequence = "da"
# previous_value + dropout(x)
```

1.5.4 Embedding

- Embedding Size (Hidden Size / Input Length)

```
# /utils/modality.py
def _body_input_length(self):
  return self._model_hparams.hidden_size
```

- Implementation
  - `SymbolModality`: `simple_body()` calls `_get_weights()`

1.6 Estimator Funcs

- Involves `t2t_model` `subclass_T2TModel` `problem` `common_hparams`

Estimator funcs related code & logic are mainly implemented in `utils/t2t_model.py` `estimator_model_fn()` and `?(eval_metrics etc)`

1.6.1 Train

- return `train_op` & `loss`
- optimizer: defined in `hparams.optimizer`
  - common hparams are defined in `common_hparams.py`
  - subclasses can overridden this hparam

1.6.2 Eval

- return `predictions` & `eval_metric_ops` & `loss`
- metrics: defined by `problem.py (and subclasses)` `eval_metrics()`

1.6.3 Predict

- return `predictions`
- call `self.infer()` in which calls `_greedy_infer()` which should be implemented in subclass. Otherwise a slower version of `infer()` will be used

1.7 Function Flow

`bottom`, `top`, and `loss` are specified in `hparams.problems`
from `features` to `logtis, losses`

1. features flow into model: $\text{t2t}_{\text{model.py}}$ `T2TModel.estimator_model_fn()` calls `logits, losses_dict = model(features)`

2. `model(features)` calls `Layers.base.Layer.__call__()` which is overridden by `T2TModel.call()`

3. `T2TModel.call()` calls `model_fn_sharded(sharded_features)`

4. `model_fn_sharded()` calls `model_fn(datashard_to_features)`

5. `model_fn()` calls `bottom()`, `body()` `top()` and `loss()` in order. `bottom()`, `top()` and `loss()` are defined in corresponding problems' `def hparams()` method which use functions defined in `layers/modalities.py`

   - `body()` can return `losses` or not
     - If return, then it must contains `(logits, losses)` in a `tuple` then training will skip `top()` and `loss()` functions. This means `body()` need to implement `top()` and `loss()` by itself
     - Otherwise, it simply return `logits` as output. Then `model_fn()` will use `top()` to calculate `logits` and `loss()` to calculate `loss()`

       `t2t_model.py` loss(): $\text{model}_{\text{body}}$ must return a dictionary of logits when $\text{problem}_{\text{hparams.target modality}}$ is a dict
   - `bottom()`: transform features to feed into body
     - `input`: embedding inputs
     - `output`: transformed features
     - `SymbolModality`
       (a) ensure 3D dimension of input feature `x`
       (b) added dropout to input features `x`
       (c) created weight matrix `ret` as embedding matrix
       (d) sampling `ret` using dropped out `x`
   - `top()`: generating logits. `body_output` to `logits`

- SymbolModality `top()`: [batch, p0, p1, $\text{body}_{\text{inputdepth}}$] -> [batch, p0, p1, ?, $\text{vocab}_{\text{size}}$] ([batch, p0, p1, 1, $\text{vocab}_{\text{size}}$] in small ptb)

- `loss()`: Default in `utils/modality.py` `loss(top_out, targets)` which uses `softmax-cross-entropy`
  - `SymbolModality` doesn't override this method, so it use cross entropy as default
  - Should be overridden when define new modality in `layers/modality.py`

6. `return logtis, losses` in `model_fn()`

1.8 Encoder-Decoder Diffs

1.8.1 self-attention

No computational differences

- layer name is different (same name with format string / variable)
- bias is different
- decoder has layer cache but currently un-implemented

1.9 MISC

problem.py spaceID? $\text{ZH}_{\text{TOK}} = 16$