Tensor2Tensor for Neural Machine Translation

Ashish Vaswani, **Samy Bengio**, **Eugene Brevdo**, **Francois Chollet**, **Aidan N. Gomez**, **Stephan Gouws**, **Llion Jones**, **Łukasz Kaiser**, **Nal Kalchbrenner**, **Niki Parmar**, **Ryan Sepassi**, **Noam Shazeer**, **Jakob Uszkoreit**

Abstract

Tensor2Tensor is a library for deep learning models that is very well-suited for neural machine translation and includes the reference implementation of the state-of-the-art Transformer model.

1 Neural Machine Translation Background

Machine translation using deep neural networks achieved great success with sequence-to-sequence models Sutskever et al. (2014); Bahdanau et al. (2014); Cho et al. (2014) that used recurrent neural networks (RNNs) with LSTM cells Hochreiter and Schmidhuber (1997). The basic sequence-to-sequence architecture is composed of an RNN encoder which reads the source sentence one token at a time and transforms it into a fixed-sized state vector. This is followed by an RNN decoder, which generates the target sentence, one token at a time, from the state vector.

While a pure sequence-to-sequence recurrent neural network can already obtain good translation results Sutskever et al. (2014); Cho et al. (2014), it suffers from the fact that the whole input sentence needs to be encoded into a single fixed-size vector. This clearly manifests itself in the degradation of translation quality on longer sentences and was partially overcome in Bahdanau et al. (2014) by using a neural model of attention.

Convolutional architectures have been used to obtain good results in word-level neural machine translation starting from Kalchbrenner and Blunsom (2013) and later in Meng et al. (2015). These early models used a standard RNN on top of the convolution to generate the output, which creates a bottleneck and hurts performance.

Fully convolutional neural machine translation without this bottleneck was first achieved in Kaiser and Bengio (2016) and Kalchbrenner et al. (2016). The model in Kaiser and Bengio (2016) (Extended Neural GPU) used a recurrent stack of gated convolutional layers, while the model in Kalchbrenner et al. (2016) (ByteNet) did away with recursion and used left-padded convolutions in the decoder. This idea, introduced in WaveNet van den Oord et al. (2016), significantly improves efficiency of the model. The same technique was improved in a number of neural translation models recently, including Gehring et al. (2017) and Kaiser et al. (2017).

2 Self-Attention

Instead of convolutions, one can use stacked self-attention layers. This was introduced in the Transformer model Vaswani et al. (2017) and has significantly improved state-of-the-art in machine translation and language modeling while also improving the speed of training. Research continues in applying the model in more domains and exploring the space of self-attention mechanisms. It is clear that self-attention is a powerful tool in general-purpose sequence modeling.

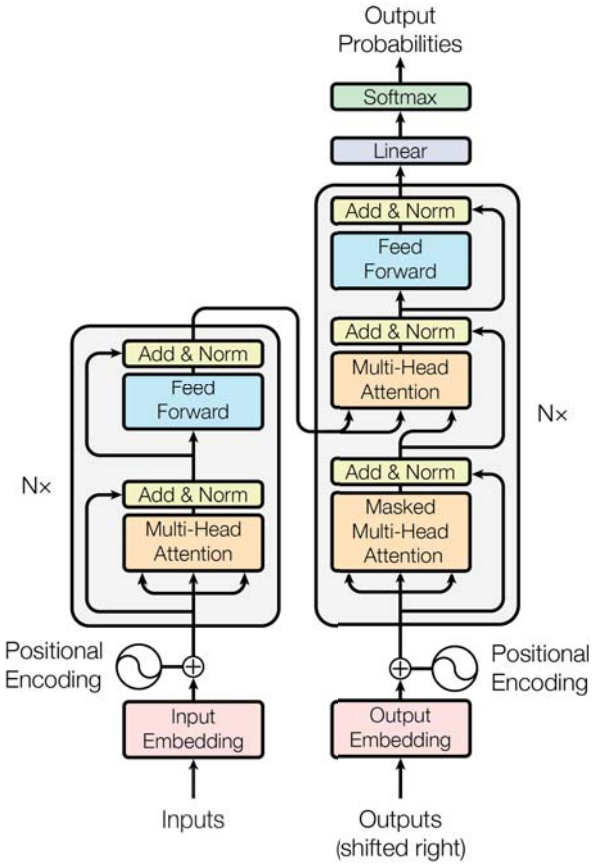

Figure 1: The Transformer model architecture.

While RNNs represent sequence history in their hidden state, the Transformer has no such fixed-size bottleneck. Instead, each timestep has full direct access to the history through the dot-product attention mechanism. This has the effect of both enabling the model to learn more distant temporal relationships, as well as speeding up training because there is no need to wait for a hidden state to propagate across time. This comes at the cost of memory usage, as the attention mechanism scales with t^2, where t is the length the sequence. Future work may reduce this scaling factor.

The Transformer model is illustrated in Figure 1. It uses stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, shown in the left and right halves of Figure 1 respectively.

Encoder: The encoder is composed of a stack of identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, positionwise fully connected feed-forward network.

Decoder: The decoder is also composed of a stack of identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack.

More details about multi-head attention and overall architecture can be found in Vaswani et al. (2017).

3 Tensor2Tensor

Tensor2Tensor (T2T) is a library of deep learning models and datasets designed to make deep learning research faster and more accessible. T2T uses TensorFlow, Abadi et al. (2016), throughout and there is a strong focus on performance as well as usability. Through its use of TensorFlow and various T2T-specific abstractions, researchers can train models on CPU, GPU (single or multiple), and TPU, locally and in the cloud, usually with no or minimal device-specific code or configuration.

Development began focused on neural machine translation and so Tensor2Tensor includes many of the most successful NMT models and standard datasets. It has since added support for other task types as well across multiple media (text, images, video, audio). Both the number of models and datasets has grown significantly.

Usage is standardized across models and problems which makes it easy to try a new model on multiple problems or try multiple models on a single problem. See Example Usage (appendix B) to see some of the usability benefits of standardization of commands and unification of datasets, models, and training, evaluation, decoding procedures.

Development is done in the open on GitHub (http://github.com/tensorflow/tensor2tensor) with many contributors inside and outside Google.

4 System Overview

There are five key components that specify a training run in Tensor2Tensor:

1. Datasets: The `Problem` class encapsulate everything about a particular dataset. A `Problem` can generate the dataset from scratch, usually downloading data from a public source, building a vocabulary, and writing encoded samples to disk. `Problem`s also produce input pipelines for training and evaluation as well as any necessary additional information per feature (for example, its type, vocabulary size, and an encoder able to convert samples to and from human and machine-readable representations).

2. Device configuration: the type, number, and location of devices. TensorFlow and Tensor2Tensor currently support CPU, GPU, and TPU in single and multi-device configurations. Tensor2Tensor also supports both synchronous and asynchronous data-parallel training.

3. Hyperparameters: parameters that control the instantiation of the model and training procedure (for example, the number of hidden layers or the optimizer's learning rate). These are specified in code and named so they can be easily shared and reproduced.

4. Model: the model ties together the preceding components to instantiate the parameterized transformation from inputs to targets, compute the loss and evaluation metrics, and construct the optimization procedure.

5. `Estimator` and `Experiment`: These classes that are part of TensorFlow handle instantiating the runtime, running the training loop, and executing basic support services like model checkpointing, logging, and alternation between training and evaluation.

These abstractions enable users to focus their attention only on the component they're interested in experimenting with. Users that wish to try models on a new problem usually only have to define a new problem. Users that wish to create or modify models only have to create a model or edit hyperparameters. The other components remain untouched, out of the way, and available for use, all of which reduces mental load and allows users to more quickly iterate on their ideas at scale.

Appendix A contains an outline of the code and appendix B contains example usage.

5 Library of research components

Tensor2Tensor provides a vehicle for research ideas to be quickly tried out and shared. Components that prove to be very useful can be committed to more widely-used libraries like TensorFlow, which contains many standard layers, optimizers, and other higher-level components.

Tensor2Tensor supports library usage as well as script usage so that users can reuse specific components in their own model or system. For example, multiple researchers are continuing work on extensions and variations of the attention-based Transformer model and the availability of the attention building blocks enables that work.

Some examples:

- The Image Transformer Parmar et al. (2018) extends the Transformer model to images. It relies heavily on many of the attention building blocks in Tensor2Tensor and adds many of its own.
- `tf.contrib.layers.rev_block`, implementing a memory-efficient block of reversible layers as presented in Gomez et al. (2017), was first implemented and exercised in Tensor2Tensor.
- The Adafactor optimizer (pending publication), which significantly reduces memory requirements for second-moment estimates, was developed within Tensor2Tensor and tried on various models and problems.
- `tf.contrib.data.bucket_by_sequence_length` enables efficient processing of sequence inputs on GPUs in the new `tf.data.Dataset` input pipeline API. It was first implemented and exercised in Tensor2Tensor.

6 Reproducibility and Continuing Development

Continuing development on a machine learning codebase while maintaining the quality of models is a difficult task because of the expense and randomness of model training. Freezing a codebase to maintain a certain configuration, or moving to an append-only process has enormous usability and development costs.

We attempt to mitigate the impact of ongoing development on historical reproducibility through 3 mechanisms:

1. Named and versioned hyperparameter sets in code
2. End-to-end regression tests that run on a regular basis for important model-problem pairs and verify that certain quality metrics are achieved.
3. Setting random seeds on multiple levels (Python, numpy, and TensorFlow) to mitigate the effects of randomness (though this is effectively impossible to achieve in full in a multithreaded, distributed, floating-point system).

If necessary, because the code is under version control on GitHub (http://github.com/tensorflow/tensor2tensor), we can always recover the exact code that produced certain experiment results.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. (2016). Tensorflow: A system for large-scale machine

learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283.

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473.

Cho, K., van Merrienboer, B., Gulcehre, C., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078.

Gehring, J., Auli, M., Grangier, D., Yarats, D., and Dauphin, Y. N. (2017). Convolutional sequence to sequence learning. *CoRR*, abs/1705.03122.

Gomez, A. N., Ren, M., Urtasun, R., and Grosse, R. B. (2017). The reversible residual network: Backpropagation without storing activations. *CoRR*, abs/1707.04585.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Kaiser, Ł. and Bengio, S. (2016). Can active memory replace attention? In *Advances in Neural Information Processing Systems*, pages 3781–3789.

Kaiser, L., Gomez, A. N., and Chollet, F. (2017). Depthwise separable convolutions for neural machine translation. *CoRR*, abs/1706.03059.

Kalchbrenner, N. and Blunsom, P. (2013). Recurrent continuous translation models. In *Proceedings EMNLP 2013*, pages 1700–1709.

Kalchbrenner, N., Espeholt, L., Simonyan, K., van den Oord, A., Graves, A., and Kavukcuoglu, K. (2016). Neural machine translation in linear time. *CoRR*, abs/1610.10099.

Meng, F., Lu, Z., Wang, M., Li, H., Jiang, W., and Liu, Q. (2015). Encoding source language with convolutional neural network for machine translation. In *ACL*, pages 20–30.

Parmar, N., Vaswani, A., Uszkoreit, J., Kaiser, Ł., Shazeer, N., and Ku, A. (2018). Image Transformer. *ArXiv e-prints*.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112.

van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. (2016). WaveNet: A generative model for raw audio. *CoRR*, abs/1609.03499.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. *CoRR*.

A Tensor2Tensor Code Outline

- Create `HParams`
- Create `RunConfig` specifying devices
  - Create and include the `Parallelism` object in the `RunConfig` which enables data-parallel duplication of the model on multiple devices (for example, for multi-GPU synchronous training).

- Create `Experiment`, including training and evaluation hooks which control support services like logging and checkpointing
- Create `Estimator` encapsulating the model function
  - `T2TModel.estimator_model_fn`
    - `model(features)`
      - `model.bottom`: This uses feature type information from the `Problem` to transform the input features into a form consumable by the model body (for example, embedding integer token ids into a dense float space).
      - `model.body`: The core of the model.
      - `model.top`: Transforming the output of the model body into the target space using information from the `Problem`
      - `model.loss`
    - When training: `model.optimize`
    - When evaluating: `create_evaluation_metrics`
- Create input functions
  - `Problem.input_fn`: produce an input pipeline for a given mode. Uses TensorFlow's `tf.data.Dataset` API.
    - `Problem.dataset` which creates a stream of individual examples
    - Pad and batch the examples into a form ready for efficient processing
- Run the `Experiment`
  - `estimator.train`
    - `train_op = model_fn(input_fn(mode=TRAIN))`
    - Run the `train_op` for the number of training steps specified
  - `estimator.evaluate`
    - `metrics = model_fn(input_fn(mode=EVAL))`
    - Accumulate the metrics across the number of evaluation steps specified

B Example Usage

Tensor2Tensor usage is standardized across problems and models. Below you'll find a set of commands that generates a dataset, trains and evaluates a model, and produces decodes from that trained model. Experiments can typically be reproduced with the (problem, model, hyperparameter set) triple.

The following train the attention-based Transformer model on WMT data translating from English to German:

```
pip install tensor2tensor

PROBLEM=translate_ende_wmt32k
MODEL=transformer
HPARAMS=transformer_base

# Generate data
t2t-datagen \
  --problem=$PROBLEM \
```

```
  --data_dir=$DATA_DIR \
  --tmp_dir=$TMP_DIR

# Train and evaluate
t2t-trainer \
--problems=$PROBLEM \
--model=$MODEL \
--hparams_set=$HPARAMS \
--data_dir=$DATA_DIR \
--output_dir=$OUTPUT_DIR \
--train_steps=250000

# Translate lines from a file
t2t-decoder \
  --data_dir=$DATA_DIR \
  --problems=$PROBLEM \
  --model=$MODEL \
  --hparams_set=$HPARAMS \
  --output_dir=$OUTPUT_DIR \
  --decode_from_file=$DECODE_FILE \
  --decode_to_file=translation.en
```