tensorflow / **nmt**

TensorFlow Neural Machine Translation Tutorial

161 commits | 3 branches | 0 releases | 22 contributors | Apache-2.0

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

oahziur committed on 23 Dec 2017 Add option to use SampleEmbeddingHelper with a temperature. ... Latest commit 365e738 on 23 Dec 2017

nmt	Add option to use SampleEmbeddingHelper with a temperature.	a month ago
CONTRIBUTING.md	PiperOrigin-RevId: 157360504	9 months ago
LICENSE	PiperOrigin-RevId: 157360504	9 months ago
README.md	Correct other spelling issue	2 months ago

README.md

Neural Machine Translation (seq2seq) Tutorial

Authors: Thang Luong, Eugene Brevdo, Rui Zhao (Google Research Blogpost, Github)

This version of the tutorial requires TensorFlow Nightly. For using the stable TensorFlow versions, please consider other branches such as tf-1.4.

If make use of this codebase for your research, please cite this.

Introduction

Sequence-to-sequence (seq2seq) models (Sutskever et al., 2014, Cho et al., 2014) have enjoyed great success in a variety of tasks such as machine translation, speech recognition, and text summarization. This tutorial gives readers a full understanding of seq2seq models and shows how to build a competitive seq2seq model from scratch. We focus on the task of Neural Machine Translation (NMT) which was the very first testbed for seq2seq models with wild success. The included code is lightweight, high-quality, production-ready, and incorporated with the latest research ideas. We achieve this goal by:

1. Using the recent decoder / attention wrapper API, TensorFlow 1.2 data iterator
2. Incorporating our strong expertise in building recurrent and seq2seq models
3. Providing tips and tricks for building the very best NMT models and replicating Google's NMT (GNMT) system.

We believe that it is important to provide benchmarks that people can easily replicate. As a result, we have provided full experimental results and pretrained on models on the following publicly available datasets:

1. *Small-scale*: English-Vietnamese parallel corpus of TED talks (133K sentence pairs) provided by the IWSLT Evaluation Campaign.
2. *Large-scale*: German-English parallel corpus (4.5M sentence pairs) provided by the WMT Evaluation Campaign.

We first build up some basic knowledge about seq2seq models for NMT, explaining how to build and train a vanilla NMT model. The second part will go into details of building a competitive NMT model with attention mechanism. We then discuss tips and tricks to build the best possible NMT models (both in speed and translation quality) such as TensorFlow best practices (batching, bucketing), bidirectional RNNs, beam search, as well as scaling up to multiple GPUs using GNMT attention.

Basic

Background on Neural Machine Translation

Back in the old days, traditional phrase-based translation systems performed their task by breaking up source sentences into multiple chunks and then translated them phrase-by-phrase. This led to disfluency in the translation outputs and was not quite like how we, humans, translate. We read the entire source sentence, understand its meaning, and then produce a translation. Neural Machine Translation (NMT) mimics that!

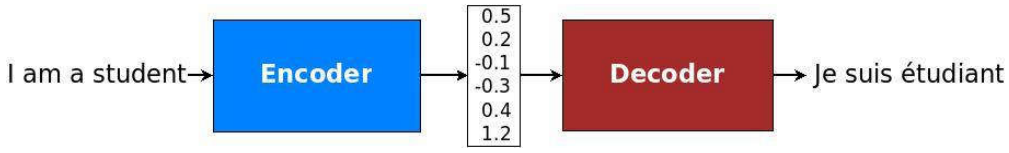

Figure 1. **Encoder-decoder architecture** – example of a general approach for NMT. An encoder converts a source sentence into a "meaning" vector which is passed through a *decoder* to produce a translation.

Specifically, an NMT system first reads the source sentence using an *encoder* to build a "thought" vector, a sequence of numbers that represents the sentence meaning; a *decoder*, then, processes the sentence vector to emit a translation, as illustrated in Figure 1. This is often referred to as the *encoder-decoder architecture*. In this manner, NMT addresses the local translation problem in the traditional phrase-based approach: it can capture *long-range dependencies* in languages, e.g., gender agreements; syntax structures; etc., and produce much more fluent translations as demonstrated by Google Neural Machine Translation systems.

NMT models vary in terms of their exact architectures. A natural choice for sequential data is the recurrent neural network (RNN), used by most NMT models. Usually an RNN is used for both the encoder and decoder. The RNN models, however, differ in terms of: (a) *directionality* – unidirectional or bidirectional; (b) *depth* – single- or multi-layer; and (c) *type* – often either a vanilla RNN, a Long Short-term Memory (LSTM), or a gated recurrent unit (GRU). Interested readers can find more information about RNNs and LSTM on this blog post.

In this tutorial, we consider as examples a *deep multi-layer RNN* which is unidirectional and uses LSTM as a recurrent unit. We show an example of such a model in Figure 2. In this example, we build a model to translate a source sentence "I am a student" into a target sentence "Je suis étudiant". At a high level, the NMT model consists of two recurrent neural networks: the *encoder* RNN simply consumes the input source words without making any prediction; the *decoder*, on the other hand, processes the target sentence while predicting the next words.

For more information, we refer readers to Luong (2016) which this tutorial is based on.

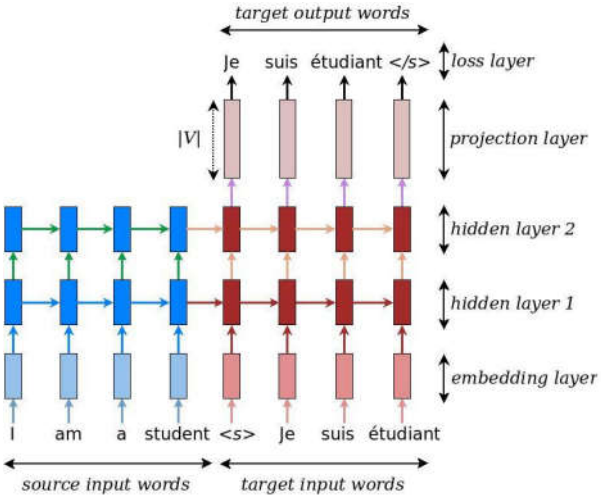

Figure 2. **Neural machine translation** – example of a deep recurrent architecture proposed by for translating a source sentence "I am a student" into a target sentence "Je suis étudiant". Here, "<s>" marks the start of the decoding process while "</s>" tells the decoder to stop.

Installing the Tutorial

To install this tutorial, you need to have TensorFlow installed on your system. This tutorial requires TensorFlow Nightly. To install TensorFlow, follow the installation instructions here.

Once TensorFlow is installed, you can download the source code of this tutorial by running:

```
git clone https://github.com/tensorflow/nmt/
```

Training – How to build our first NMT system

Let's first dive into the heart of building an NMT model with concrete code snippets through which we will explain Figure 2 in more detail. We defer data preparation and the full code to later. This part refers to file **model.py**.

At the bottom layer, the encoder and decoder RNNs receive as input the following: first, the source sentence, then a boundary marker "<s>" which indicates the transition from the encoding to the decoding mode, and the target sentence. For *training*, we will feed the system with the following tensors, which are in time-major format and contain word indices:

- **encoder_inputs** [max_encoder_time, batch_size]: source input words.
- **decoder_inputs** [max_decoder_time, batch_size]: target input words.
- **decoder_outputs** [max_decoder_time, batch_size]: target output words, these are decoder_inputs shifted to the left by one time step with an end-of-sentence tag appended on the right.

Here for efficiency, we train with multiple sentences (batch_size) at once. Testing is slightly different, so we will discuss it later.

Embedding

Given the categorical nature of words, the model must first look up the source and target embeddings to retrieve the corresponding word representations. For this *embedding layer* to work, a vocabulary is first chosen for each language. Usually, a vocabulary size V is selected, and only the most frequent V words are treated as unique. All other words are converted to an "unknown" token and all get the same embedding. The embedding weights, one set per language, are usually learned during training.

```
# Embedding
embedding_encoder = variable_scope.get_variable(
    "embedding_encoder", [src_vocab_size, embedding_size], ...)
```

```
# Look up embedding:
#   encoder_inputs: [max_time, batch_size]
#   encoder_emb_inp: [max_time, batch_size, embedding_size]
encoder_emb_inp = embedding_ops.embedding_lookup(
    embedding_encoder, encoder_inputs)
```

Similarly, we can build *embedding_decoder* and *decoder_emb_inp*. Note that one can choose to initialize embedding weights with pretrained word representations such as word2vec or Glove vectors. In general, given a large amount of training data we can learn these embeddings from scratch.

Encoder

Once retrieved, the word embeddings are then fed as input into the main network, which consists of two multi-layer RNNs – an encoder for the source language and a decoder for the target language. These two RNNs, in principle, can share the same weights; however, in practice, we often use two different RNN parameters (such models do a better job when fitting large training datasets). The *encoder* RNN uses zero vectors as its starting states and is built as follows:

```
# Build RNN cell
encoder_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units)

# Run Dynamic RNN
#   encoder_outputs: [max_time, batch_size, num_units]
#   encoder_state: [batch_size, num_units]
encoder_outputs, encoder_state = tf.nn.dynamic_rnn(
    encoder_cell, encoder_emb_inp,
    sequence_length=source_sequence_length, time_major=True)
```

Note that sentences have different lengths to avoid wasting computation, we tell *dynamic_rnn* the exact source sentence lengths through *source_sequence_length*. Since our input is time major, we set *time_major=True*. Here, we build only a single layer LSTM, *encoder_cell*. We will describe how to build multi-layer LSTMs, add dropout, and use attention in a later section.

Decoder

The *decoder* also needs to have access to the source information, and one simple way to achieve that is to initialize it with the last hidden state of the encoder, *encoder_state*. In Figure 2, we pass the hidden state at the source word "student" to the decoder side.

```
# Build RNN cell
decoder_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units)


# Helper
helper = tf.contrib.seq2seq.TrainingHelper(
    decoder_emb_inp, decoder_lengths, time_major=True)
# Decoder
decoder = tf.contrib.seq2seq.BasicDecoder(
    decoder_cell, helper, encoder_state,
    output_layer=projection_layer)
# Dynamic decoding
outputs, _ = tf.contrib.seq2seq.dynamic_decode(decoder, ...)
logits = outputs.rnn_output
```

Here, the core part of this code is the *BasicDecoder* object, *decoder*, which receives *decoder_cell* (similar to encoder_cell), a *helper*, and the previous *encoder_state* as inputs. By separating out decoders and helpers, we can reuse different codebases, e.g., *TrainingHelper* can be substituted with *GreedyEmbeddingHelper* to do greedy decoding. See more in helper.py.

Lastly, we haven't mentioned *projection_layer* which is a dense matrix to turn the top hidden states to logit vectors of dimension V. We illustrate this process at the top of Figure 2.

```
projection_layer = layers_core.Dense(
    tgt_vocab_size, use_bias=False)
```

Loss

Given the *logits* above, we are now ready to compute our training loss:

```
crossent = tf.nn.sparse_softmax_cross_entropy_with_logits(
    labels=decoder_outputs, logits=logits)
train_loss = (tf.reduce_sum(crossent * target_weights) /
    batch_size)
```

Here, *target_weights* is a zero-one matrix of the same size as *decoder_outputs*. It masks padding positions outside of the target sequence lengths with values 0.

Important note: It's worth pointing out that we divide the loss by *batch_size*, so our hyperparameters are "invariant" to batch_size. Some people divide the loss by (*batch_size* * *num_time_steps*), which plays down the errors made on short sentences. More subtly, our hyperparameters (applied to the former way) can't be used for the latter way. For example, if both approaches use SGD with a learning of 1.0, the latter approach effectively uses a much smaller learning rate of 1 / *num_time_steps*.

Gradient computation & optimization

We have now defined the forward pass of our NMT model. Computing the backpropagation pass is just a matter of a few lines of code:

```
# Calculate and clip gradients
params = tf.trainable_variables()
gradients = tf.gradients(train_loss, params)
clipped_gradients, _ = tf.clip_by_global_norm(
    gradients, max_gradient_norm)
```

One of the important steps in training RNNs is gradient clipping. Here, we clip by the global norm. The max value, *max_gradient_norm*, is often set to a value like 5 or 1. The last step is selecting the optimizer. The Adam optimizer is a common choice. We also select a learning rate. The value of *learning_rate* can is usually in the range 0.0001 to 0.001; and can be set to decrease as training progresses.

```
# Optimization
optimizer = tf.train.AdamOptimizer(learning_rate)
update_step = optimizer.apply_gradients(
    zip(clipped_gradients, params))
```

In our own experiments, we use standard SGD (tf.train.GradientDescentOptimizer) with a decreasing learning rate schedule, which yields better performance. See the benchmarks.

Hands-on – Let's train an NMT model

Let's train our very first NMT model, translating from Vietnamese to English! The entry point of our code is **nmt.py**.

We will use a *small-scale parallel corpus of TED talks* (133K training examples) for this exercise. All data we used here can be found at: https://nlp.stanford.edu/projects/nmt/. We will use tst2012 as our dev dataset, and tst2013 as our test dataset.

Run the following command to download the data for training NMT model:

```
nmt/scripts/download_iwslt15.sh /tmp/nmt_data
```

Run the following command to start the training:

```
mkdir /tmp/nmt_model
python -m nmt.nmt \
    --src=vi --tgt=en \
    --vocab_prefix=/tmp/nmt_data/vocab  \
    --train_prefix=/tmp/nmt_data/train \
    --dev_prefix=/tmp/nmt_data/tst2012  \
    --test_prefix=/tmp/nmt_data/tst2013 \
    --out_dir=/tmp/nmt_model \
    --num_train_steps=12000 \
    --steps_per_stats=100 \
    --num_layers=2 \
    --num_units=128 \
    --dropout=0.2 \
    --metrics=bleu
```

The above command trains a 2-layer LSTM seq2seq model with 128-dim hidden units and embeddings for 12 epochs. We use a dropout value of 0.2 (keep probability 0.8). If no error, we should see logs similar to the below with decreasing perplexity values as we train.

```
# First evaluation, global step 0
  eval dev: perplexity 17193.66
  eval test: perplexity 17193.27
# Start epoch 0, step 0, lr 1, Tue Apr 25 23:17:41 2017
  sample train data:
    src_reverse: </s> </s> Điều đó , dĩ nhiên , là câu chuyện trích ra từ học thuyết của Karl Marx .
    ref: That , of course , was the <unk> distilled from the theories of Karl Marx . </s> </s> </s>
  epoch 0 step 100 lr 1 step-time 0.89s wps 5.78K ppl 1568.62 bleu 0.00
  epoch 0 step 200 lr 1 step-time 0.94s wps 5.91K ppl 524.11 bleu 0.00
  epoch 0 step 300 lr 1 step-time 0.96s wps 5.80K ppl 340.05 bleu 0.00
  epoch 0 step 400 lr 1 step-time 1.02s wps 6.06K ppl 277.61 bleu 0.00
  epoch 0 step 500 lr 1 step-time 0.95s wps 5.89K ppl 205.85 bleu 0.00
```

See **train.py** for more details.

We can start Tensorboard to view the summary of the model during training:

```
tensorboard --port 22222 --logdir /tmp/nmt_model/
```

Training the reverse direction from English and Vietnamese can be done simply by changing:

```
--src=en --tgt=vi
```

Inference – How to generate translations

While you're training your NMT models (and once you have trained models), you can obtain translations given previously unseen source sentences. This process is called inference. There is a clear distinction between training and inference (*testing*): at inference time, we only have access to the source sentence, i.e., *encoder_inputs*. There are many ways to perform decoding. Decoding methods include greedy, sampling, and beam-search decoding. Here, we will discuss the greedy decoding strategy.

The idea is simple and we illustrate it in Figure 3:

1. We still encode the source sentence in the same way as during training to obtain an *encoder_state*, and this *encoder_state* is used to initialize the decoder.
2. The decoding (translation) process is started as soon as the decoder receives a starting symbol "<s>" (refer as *tgt_sos_id* in our code);
3. For each timestep on the decoder side, we treat the RNN's output as a set of logits. We choose the most likely word, the id associated with the maximum logit value, as the emitted word (this is the "greedy" behavior). For example in Figure 3, the word "moi" has the highest translation probability in the first decoding step. We then feed this word as input to the next timestep.
4. The process continues until the end-of-sentence marker "</s>" is produced as an output symbol (refer as *tgt_eos_id* in our code).

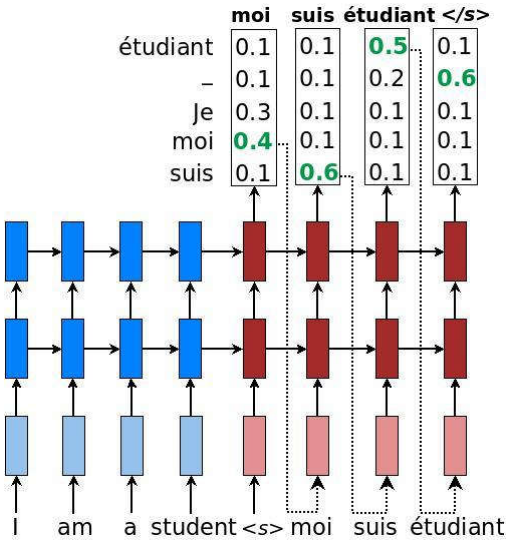

Figure 3. **Greedy decoding** – example of how a trained NMT model produces a translation for a source sentence "Je suis étudiant" using greedy search.

Step 3 is what makes inference different from training. Instead of always feeding the correct target words as an input, inference uses words predicted by the model. Here's the code to achieve greedy decoding. It is very similar to the training decoder.

```
# Helper
helper = tf.contrib.seq2seq.GreedyEmbeddingHelper(
    embedding_decoder,
    tf.fill([batch_size], tgt_sos_id), tgt_eos_id)

# Decoder
decoder = tf.contrib.seq2seq.BasicDecoder(
    decoder_cell, helper, encoder_state,
    output_layer=projection_layer)
# Dynamic decoding
outputs, _ = tf.contrib.seq2seq.dynamic_decode(
    decoder, maximum_iterations=maximum_iterations)
translations = outputs.sample_id
```

Here, we use *GreedyEmbeddingHelper* instead of *TrainingHelper*. Since we do not know the target sequence lengths in advance, we use *maximum_iterations* to limit the translation lengths. One heuristic is to decode up to two times the source sentence lengths.

```
maximum_iterations = tf.round(tf.reduce_max(source_sequence_length) * 2)
```

Having trained a model, we can now create an inference file and translate some sentences:

```
cat > /tmp/my_infer_file.vi
# (copy and paste some sentences from /tmp/nmt_data/tst2013.vi)

python -m nmt.nmt \
    --out_dir=/tmp/nmt_model \
    --inference_input_file=/tmp/my_infer_file.vi \
    --inference_output_file=/tmp/nmt_model/output_infer

cat /tmp/nmt_model/output_infer # To view the inference as output
```

Note the above commands can also be run while the model is still being trained as long as there exists a training checkpoint. See inference.py for more details.

Intermediate

Having gone through the most basic seq2seq model, let's get more advanced! To build state-of-the-art neural machine translation systems, we will need more "secret sauce": the *attention mechanism*, which was first introduced by Bahdanau et al., 2015, then later refined by Luong et al., 2015 and others. The key idea of the attention mechanism is to establish direct short-cut connections between the target and the source by paying "attention" to relevant source content as we translate. A nice byproduct of the attention mechanism is an easy-to-visualize alignment matrix between the source and target sentences (as shown in Figure 4).

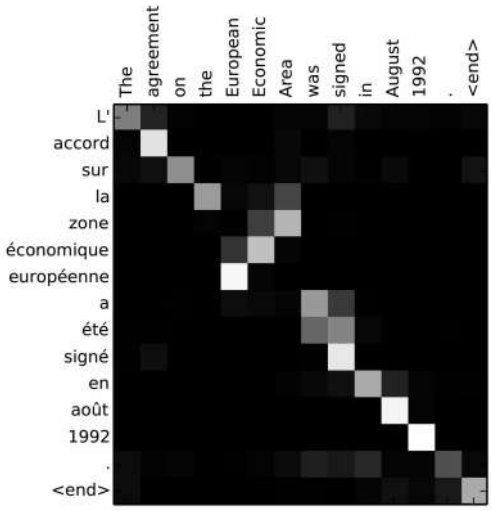

Figure 4. **Attention visualization** – example of the alignments between source and target sentences. Image is taken from (Bahdanau et al., 2015).

Remember that in the vanilla seq2seq model, we pass the last source state from the encoder to the decoder when starting the decoding process. This works well for short and medium-length sentences; however, for long sentences, the single fixed-size hidden state becomes an information bottleneck. Instead of discarding all of the hidden states computed in the source RNN, the attention mechanism provides an approach that allows the decoder to peek at them (treating them as a dynamic memory of the source information). By doing so, the attention mechanism improves the translation of longer sentences. Nowadays, attention mechanisms are the defacto standard and have been successfully applied to many other tasks (including image caption generation, speech recognition, and text summarization).

Background on the Attention Mechanism

We now describe an instance of the attention mechanism proposed in (Luong et al., 2015), which has been used in several state-of-the-art systems including open-source toolkits such as OpenNMT and in the TF seq2seq API in this tutorial. We will also provide connections to other variants of the attention mechanism.

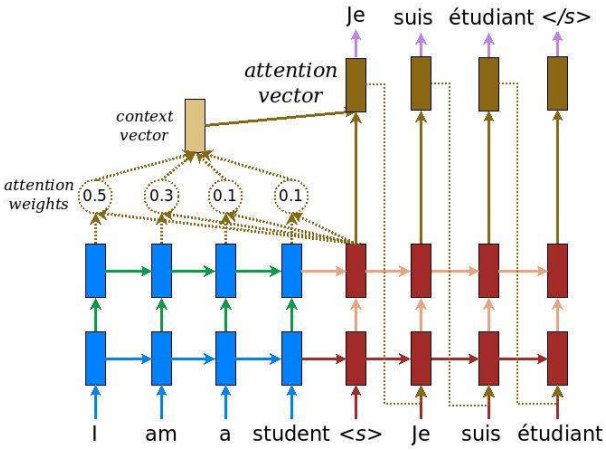

Figure 5. **Attention mechanism** – example of an attention-based NMT system as described in (Luong et al., 2015) . We highlight in detail the first step of the attention computation. For clarity, we don't show the embedding and projection layers in Figure (2).

As illustrated in Figure 5, the attention computation happens at every decoder time step. It consists of the following stages:

1. The current target hidden state is compared with all source states to derive *attention weights* (can be visualized as in Figure 4).
2. Based on the attention weights we compute a *context vector* as the weighted average of the source states.

3. Combine the context vector with the current target hidden state to yield the final *attention vector*
4. The attention vector is fed as an input to the next time step (*input feeding*). The first three steps can be summarized by the equations below:

$$\alpha_{ts} = \frac{\exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'=1}^{S} \exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)} \quad \text{[Attention weights]} \quad (1)$$

$$\boldsymbol{c}_t = \sum_s \alpha_{ts} \bar{\boldsymbol{h}}_s \quad \text{[Context vector]} \quad (2)$$

$$\boldsymbol{a}_t = f(\boldsymbol{c}_t, \boldsymbol{h}_t) = \tanh(\boldsymbol{W}_{\boldsymbol{c}}[\boldsymbol{c}_t; \boldsymbol{h}_t]) \quad \text{[Attention vector]} \quad (3)$$

Here, the function `score` is used to compared the target hidden state $$h_t$$ with each of the source hidden states $$\overline{h}_s$$, and the result is normalized to produced attention weights (a distribution over source positions). There are various choices of the scoring function; popular scoring functions include the multiplicative and additive forms given in Eq. (4). Once computed, the attention vector $$a_t$$ is used to derive the softmax logit and loss. This is similar to the target hidden state at the top layer of a vanilla seq2seq model. The function `f` can also take other forms.

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \boldsymbol{W} \bar{\boldsymbol{h}}_s & \text{[Luong's multiplicative style]} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W}_{\boldsymbol{1}} \boldsymbol{h}_t + \boldsymbol{W}_{\boldsymbol{2}} \bar{\boldsymbol{h}}_s\right) & \text{[Bahdanau's additive style]} \end{cases} \quad (4)$$

Various implementations of attention mechanisms can be found in attention_wrapper.py.

What matters in the attention mechanism?

As hinted in the above equations, there are many different attention variants. These variants depend on the form of the scoring function and the attention function, and on whether the previous state $$h_{t-1}$$ is used instead of $$h_t$$ in the scoring function as originally suggested in (Bahdanau et al., 2015). Empirically, we found that only certain choices matter. First, the basic form of attention, i.e., direct connections between target and source, needs to be present. Second, it's important to feed the attention vector to the next timestep to inform the network about past attention decisions as demonstrated in (Luong et al., 2015). Lastly, choices of the scoring function can often result in different performance. See more in the benchmark results section.

Attention Wrapper API

In our implementation of the AttentionWrapper, we borrow some terminology from (Weston et al., 2015) in their work on *memory networks*. Instead of having readable & writable memory, the attention mechanism presented in this tutorial is a *read-only* memory. Specifically, the set of source hidden states (or their transformed versions, e.g., $$W\overline{h}_s$$ in Luong's scoring style or $$W_2\overline{h}_s$$ in Bahdanau's scoring style) is referred to as the "*memory*". At each time step, we use the current target hidden state as a "*query*" to decide on which parts of the memory to read. Usually, the query needs to be compared with keys corresponding to individual memory slots. In the above presentation of the attention mechanism, we happen to use the set of source hidden states (or their transformed versions, e.g., $$W_1h_t$$ in Bahdanau's scoring style) as "keys". One can be inspired by this memory-network terminology to derive other forms of attention!

Thanks to the attention wrapper, extending our vanilla seq2seq code with attention is trivial. This part refers to file **attention_model.py**

First, we need to define an attention mechanism, e.g., from (Luong et al., 2015):

```
# attention_states: [batch_size, max_time, num_units]
attention_states = tf.transpose(encoder_outputs, [1, 0, 2])

# Create an attention mechanism
attention_mechanism = tf.contrib.seq2seq.LuongAttention(
    num_units, attention_states,
    memory_sequence_length=source_sequence_length)
```

In the previous Encoder section, *encoder_outputs* is the set of all source hidden states at the top layer and has the shape of *[max_time, batch_size, num_units]* (since we use *dynamic_rnn* with *time_major* set to *True* for efficiency). For the attention mechanism, we need to make sure the "memory" passed in is batch major, so we need to transpose *attention_states*. We pass *source_sequence_length* to the attention mechanism to ensure that the attention weights are properly normalized (over non-padding positions only).

Having defined an attention mechanism, we use *AttentionWrapper* to wrap the decoding cell:

```
decoder_cell = tf.contrib.seq2seq.AttentionWrapper(
    decoder_cell, attention_mechanism,
    attention_layer_size=num_units)
```

The rest of the code is almost the same as in the Section Decoder!

Hands-on – building an attention-based NMT model

To enable attention, we need to use one of `luong`, `scaled_luong`, `bahdanau` or `normed_bahdanau` as the value of the `attention` flag during training. The flag specifies which attention mechanism we are going to use. In addition, we need to create a new directory for the attention model, so we don't reuse the previously trained basic NMT model.

Run the following command to start the training:

```
mkdir /tmp/nmt_attention_model

python -m nmt.nmt \
    --attention=scaled_luong \
    --src=vi --tgt=en \
    --vocab_prefix=/tmp/nmt_data/vocab  \
    --train_prefix=/tmp/nmt_data/train \
    --dev_prefix=/tmp/nmt_data/tst2012  \
    --test_prefix=/tmp/nmt_data/tst2013 \
    --out_dir=/tmp/nmt_attention_model \
    --num_train_steps=12000 \
    --steps_per_stats=100 \
    --num_layers=2 \
    --num_units=128 \
    --dropout=0.2 \
    --metrics=bleu
```

After training, we can use the same inference command with the new out_dir for inference:

```
python -m nmt.nmt \
    --out_dir=/tmp/nmt_attention_model \
    --inference_input_file=/tmp/my_infer_file.vi \
    --inference_output_file=/tmp/nmt_attention_model/output_infer
```

Tips & Tricks

Building Training, Eval, and Inference Graphs

When building a machine learning model in TensorFlow, it's often best to build three separate graphs:

- The Training graph, which:
  - Batches, buckets, and possibly subsamples input data from a set of files/external inputs.
  - Includes the forward and backprop ops.
  - Constructs the optimizer, and adds the training op.
- The Eval graph, which:
  - Batches and buckets input data from a set of files/external inputs.
  - Includes the training forward ops, and additional evaluation ops that aren't used for training.
- The Inference graph, which:
  - May not batch input data.
  - Does not subsample or bucket input data.
  - Reads input data from placeholders (data can be fed directly to the graph via *feed_dict* or from a C++ TensorFlow serving binary).
  - Includes a subset of the model forward ops, and possibly additional special inputs/outputs for storing state between session.run calls.

Building separate graphs has several benefits:

- The inference graph is usually very different from the other two, so it makes sense to build it separately.
- The eval graph becomes simpler since it no longer has all the additional backprop ops.
- Data feeding can be implemented separately for each graph.
- Variable reuse is much simpler. For example, in the eval graph there's no need to reopen variable scopes with *reuse=True* just because the Training model created these variables already. So the same code can be reused without sprinkling *reuse=* arguments everywhere.
- In distributed training, it is commonplace to have separate workers perform training, eval, and inference. These need to build their own graphs anyway. So building the system this way prepares you for distributed training.

The primary source of complexity becomes how to share Variables across the three graphs in a single machine setting. This is solved by using a separate session for each graph. The training session periodically saves checkpoints, and the eval session and the infer session restore parameters from checkpoints. The example below shows the main differences between the two approaches.

Before: Three models in a single graph and sharing a single Session

```
with tf.variable_scope('root'):
  train_inputs = tf.placeholder()
  train_op, loss = BuildTrainModel(train_inputs)
  initializer = tf.global_variables_initializer()

with tf.variable_scope('root', reuse=True):
  eval_inputs = tf.placeholder()
  eval_loss = BuildEvalModel(eval_inputs)

with tf.variable_scope('root', reuse=True):
  infer_inputs = tf.placeholder()
  inference_output = BuildInferenceModel(infer_inputs)

sess = tf.Session()

sess.run(initializer)

for i in itertools.count():
  train_input_data = ...
  sess.run([loss, train_op], feed_dict={train_inputs: train_input_data})

  if i % EVAL_STEPS == 0:
    while data_to_eval:
      eval_input_data = ...
      sess.run([eval_loss], feed_dict={eval_inputs: eval_input_data})

  if i % INFER_STEPS == 0:
    sess.run(inference_output, feed_dict={infer_inputs: infer_input_data})
```

After: Three models in three graphs, with three Sessions sharing the same Variables

```
train_graph = tf.Graph()
eval_graph = tf.Graph()
infer_graph = tf.Graph()

with train_graph.as_default():
  train_iterator = ...
  train_model = BuildTrainModel(train_iterator)
  initializer = tf.global_variables_initializer()

with eval_graph.as_default():
  eval_iterator = ...
  eval_model = BuildEvalModel(eval_iterator)

with infer_graph.as_default():
  infer_iterator, infer_inputs = ...
  infer_model = BuildInferenceModel(infer_iterator)

checkpoints_path = "/tmp/model/checkpoints"

train_sess = tf.Session(graph=train_graph)
eval_sess = tf.Session(graph=eval_graph)
infer_sess = tf.Session(graph=infer_graph)
```

```
train_sess.run(initializer)
train_sess.run(train_iterator.initializer)

for i in itertools.count():

  train_model.train(train_sess)

  if i % EVAL_STEPS == 0:
    checkpoint_path = train_model.saver.save(train_sess, checkpoints_path, global_step=i)
    eval_model.saver.restore(eval_sess, checkpoint_path)
    eval_sess.run(eval_iterator.initializer)
    while data_to_eval:
      eval_model.eval(eval_sess)

  if i % INFER_STEPS == 0:
    checkpoint_path = train_model.saver.save(train_sess, checkpoints_path, global_step=i)
    infer_model.saver.restore(infer_sess, checkpoint_path)
    infer_sess.run(infer_iterator.initializer, feed_dict={infer_inputs: infer_input_data})
    while data_to_infer:
      infer_model.infer(infer_sess)
```

Notice how the latter approach is "ready" to be converted to a distributed version.

One other difference in the new approach is that instead of using *feed_dicts* to feed data at each *session.run* call (and thereby performing our own batching, bucketing, and manipulating of data), we use stateful iterator objects. These iterators make the input pipeline much easier in both the single-machine and distributed setting. We will cover the new input data pipeline (as introduced in TensorFlow 1.2) in the next section.

Data Input Pipeline

Prior to TensorFlow 1.2, users had two options for feeding data to the TensorFlow training and eval pipelines:

1. Feed data directly via *feed_dict* at each training *session.run* call.
2. Use the queueing mechanisms in *tf.train* (e.g. *tf.train.batch*) and *tf.contrib.train*.
3. Use helpers from a higher level framework like *tf.contrib.learn* or *tf.contrib.slim* (which effectively use #2).

The first approach is easier for users who aren't familiar with TensorFlow or need to do exotic input modification (i.e., their own minibatch queueing) that can only be done in Python. The second and third approaches are more standard but a little less flexible; they also require starting multiple python threads (queue runners). Furthermore, if used incorrectly queues can lead to deadlocks or opaque error messages. Nevertheless, queues are significantly more efficient than using *feed_dict* and are the standard for both single-machine and distributed training.

Starting in TensorFlow 1.2, there is a new system available for reading data into TensorFlow models: dataset iterators, as found in the **tf.data** module. Data iterators are flexible, easy to reason about and to manipulate, and provide efficiency and multithreading by leveraging the TensorFlow C++ runtime.

A **dataset** can be created from a batch data Tensor, a filename, or a Tensor containing multiple filenames. Some examples:

```
# Training dataset consists of multiple files.
train_dataset = tf.data.TextLineDataset(train_files)

# Evaluation dataset uses a single file, but we may
# point to a different file for each evaluation round.
eval_file = tf.placeholder(tf.string, shape=())
eval_dataset = tf.data.TextLineDataset(eval_file)

# For inference, feed input data to the dataset directly via feed_dict.
infer_batch = tf.placeholder(tf.string, shape=(num_infer_examples,))
infer_dataset = tf.data.Dataset.from_tensor_slices(infer_batch)
```

All datasets can be treated similarly via input processing. This includes reading and cleaning the data, bucketing (in the case of training and eval), filtering, and batching.

To convert each sentence into vectors of word strings, for example, we use the dataset map transformation:

```
dataset = dataset.map(lambda string: tf.string_split([string]).values)
```

We can then switch each sentence vector into a tuple containing both the vector and its dynamic length:

```
dataset = dataset.map(lambda words: (words, tf.size(words))
```

Finally, we can perform a vocabulary lookup on each sentence. Given a lookup table object table, this map converts the first tuple elements from a vector of strings to a vector of integers.

```
dataset = dataset.map(lambda words, size: (table.lookup(words), size))
```

Joining two datasets is also easy. If two files contain line-by-line translations of each other and each one is read into its own dataset, then a new dataset containing the tuples of the zipped lines can be created via:

```
source_target_dataset = tf.data.Dataset.zip((source_dataset, target_dataset))
```

Batching of variable-length sentences is straightforward. The following transformation batches *batch_size* elements from *source_target_dataset*, and respectively pads the source and target vectors to the length of the longest source and target vector in each batch.

```
batched_dataset = source_target_dataset.padded_batch(
        batch_size,
        padded_shapes=((tf.TensorShape([None]),  # source vectors of unknown size
                        tf.TensorShape([])),     # size(source)
                       (tf.TensorShape([None]),  # target vectors of unknown size
                        tf.TensorShape([]))),    # size(target)
        padding_values=((src_eos_id,  # source vectors padded on the right with src_eos_id
                         0),          # size(source) -- unused
                        (tgt_eos_id,  # target vectors padded on the right with tgt_eos_id
                         0)))         # size(target) -- unused
```

Values emitted from this dataset will be nested tuples whose tensors have a leftmost dimension of size *batch_size*. The structure will be:

- iterator[0][0] has the batched and padded source sentence matrices.
- iterator[0][1] has the batched source size vectors.
- iterator[1][0] has the batched and padded target sentence matrices.
- iterator[1][1] has the batched target size vectors.

Finally, bucketing that batches similarly-sized source sentences together is also possible. Please see the file utils/iterator_utils.py for more details and the full implementation.

Reading data from a Dataset requires three lines of code: create the iterator, get its values, and initialize it.

```
batched_iterator = batched_dataset.make_initializable_iterator()

((source, source_lengths), (target, target_lengths)) = batched_iterator.get_next()

# At initialization time.
session.run(batched_iterator.initializer, feed_dict={...})
```

Once the iterator is initialized, every *session.run* call that accesses source or target tensors will request the next minibatch from the underlying dataset.

Other details for better NMT models

Bidirectional RNNs

Bidirectionality on the encoder side generally gives better performance (with some degradation in speed as more layers are used). Here, we give a simplified example of how to build an encoder with a single bidirectional layer:

```
# Construct forward and backward cells
forward_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units)
backward_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units)
```

```
bi_outputs, encoder_state = tf.nn.bidirectional_dynamic_rnn(
    forward_cell, backward_cell, encoder_emb_inp,
    sequence_length=source_sequence_length, time_major=True)
encoder_outputs = tf.concat(bi_outputs, -1)
```

The variables *encoder_outputs* and *encoder_state* can be used in the same way as in Section Encoder. Note that, for multiple bidirectional layers, we need to manipulate the encoder_state a bit, see model.py, method *_build_bidirectional_rnn()* for more details.

Beam search

While greedy decoding can give us quite reasonable translation quality, a beam search decoder can further boost performance. The idea of beam search is to better explore the search space of all possible translations by keeping around a small set of top candidates as we translate. The size of the beam is called *beam width*; a minimal beam width of, say size 10, is generally sufficient. For more information, we refer readers to Section 7.2.3 of Neubig, (2017). Here's an example of how beam search can be done:

```
# Replicate encoder infos beam_width times
decoder_initial_state = tf.contrib.seq2seq.tile_batch(
    encoder_state, multiplier=hparams.beam_width)

# Define a beam-search decoder
decoder = tf.contrib.seq2seq.BeamSearchDecoder(
        cell=decoder_cell,
        embedding=embedding_decoder,
        start_tokens=start_tokens,
        end_token=end_token,
        initial_state=decoder_initial_state,
        beam_width=beam_width,
        output_layer=projection_layer,
        length_penalty_weight=0.0)

# Dynamic decoding
outputs, _ = tf.contrib.seq2seq.dynamic_decode(decoder, ...)
```

Note that the same *dynamic_decode()* API call is used, similar to the Section Decoder. Once decoded, we can access the translations as follows:

```
translations = outputs.predicted_ids
# Make sure translations shape is [batch_size, beam_width, time]
if self.time_major:
   translations = tf.transpose(translations, perm=[1, 2, 0])
```

See model.py, method *_build_decoder()* for more details.

Hyperparameters

There are several hyperparameters that can lead to additional performances. Here, we list some based on our own experience [Disclaimers: others might not agree on things we wrote!].

Optimizer: while Adam can lead to reasonable results for "unfamiliar" architectures, SGD with scheduling will generally lead to better performance if you can train with SGD.

Attention: Bahdanau-style attention often requires bidirectionality on the encoder side to work well; whereas Luong-style attention tends to work well for different settings. For this tutorial code, we recommend using the two improved variants of Luong & Bahdanau-style attentions: *scaled_luong* & *normed bahdanau*.

Multi-GPU training

Training a NMT model may take several days. Placing different RNN layers on different GPUs can improve the training speed. Here's an example to create RNN layers on multiple GPUs.

```
cells = []
for i in range(num_layers):
  cells.append(tf.contrib.rnn.DeviceWrapper(
      tf.contrib.rnn.LSTMCell(num_units),
      "/gpu:%d" % (num_layers % num_gpus)))
cell = tf.contrib.rnn.MultiRNNCell(cells)
```

In addition, we need to enable the `colocate_gradients_with_ops` option in `tf.gradients` to parallelize the gradients computation.

You may notice the speed improvement of the attention based NMT model is very small as the number of GPUs increases. One major drawback of the standard attention architecture is using the top (final) layer's output to query attention at each time step. That means each decoding step must wait its previous step completely finished; hence, we can't parallelize the decoding process by simply placing RNN layers on multiple GPUs.

The GNMT attention architecture parallelizes the decoder's computation by using the bottom (first) layer's output to query attention. Therefore, each decoding step can start as soon as its previous step's first layer and attention computation finished. We implemented the architecture in GNMTAttentionMultiCell, a subclass of *tf.contrib.rnn.MultiRNNCell*. Here's an example of how to create a decoder cell with the *GNMTAttentionMultiCell*.

```
cells = []
for i in range(num_layers):
  cells.append(tf.contrib.rnn.DeviceWrapper(
      tf.contrib.rnn.LSTMCell(num_units),
      "/gpu:%d" % (num_layers % num_gpus)))
attention_cell = cells.pop(0)
attention_cell = tf.contrib.seq2seq.AttentionWrapper(
    attention_cell,
    attention_mechanism,
    attention_layer_size=None,  # don't add an additional dense layer.
    output_attention=False,)
cell = GNMTAttentionMultiCell(attention_cell, cells)
```

Benchmarks

IWSLT English-Vietnamese

Train: 133K examples, vocab=vocab.(vi|en), train=train.(vi|en) dev=tst2012.(vi|en), test=tst2013.(vi|en), download script.

Training details. We train 2-layer LSTMs of 512 units with bidirectional encoder (i.e., 1 bidirectional layers for the encoder), embedding dim is 512. LuongAttention (scale=True) is used together with dropout keep_prob of 0.8. All parameters are uniformly. We use SGD with learning rate 1.0 as follows: train for 12K steps (~ 12 epochs); after 8K steps, we start halving learning rate every 1K step.

Results.

Below are the averaged results of 2 models (model 1, model 2).
We measure the translation quality in terms of BLEU scores (Papineni et al., 2002).

Systems	tst2012 (dev)	test2013 (test)
NMT (greedy)	23.2	25.5
NMT (beam=10)	23.8	**26.1**
(Luong & Manning, 2015)	-	23.3

Training Speed: (0.37s step-time, 15.3K wps) on *K40m* & (0.17s step-time, 32.2K wps) on *TitanX*.
Here, step-time means the time taken to run one mini-batch (of size 128). For wps, we count words on both the source and target.

WMT German-English

Train: 4.5M examples, vocab=vocab.bpe.32000.(de|en), train=train.tok.clean.bpe.32000.(de|en), dev=newstest2013.tok.bpe.32000.(de|en), test=newstest2015.tok.bpe.32000.(de|en), download script

Training details. Our training hyperparameters are similar to the English-Vietnamese experiments except for the following details. The data is split into subword units using BPE (32K operations). We train 4-layer LSTMs of 1024 units with bidirectional encoder (i.e., 2 bidirectional layers for the encoder), embedding dim is 1024. We train for 350K steps (~ 10 epochs); after 170K steps, we start halving learning rate every 17K step.

Results.

The first 2 rows are the averaged results of 2 models (model 1, model 2). Results in the third row is with GNMT attention (model) ; trained with 4 GPUs.

Systems	newstest2013 (dev)	newstest2015
NMT (greedy)	27.1	27.6
NMT (beam=10)	28.0	28.9
NMT + GNMT attention (beam=10)	29.0	**29.9**
WMT SOTA	-	29.3

These results show that our code builds strong baseline systems for NMT.
(Note that WMT systems generally utilize a huge amount monolingual data which we currently do not.)

Training Speed: (2.1s step-time, 3.4K wps) on *Nvidia K40m* & (0.7s step-time, 8.7K wps) on *Nvidia TitanX* for standard models. To see the speed-ups with GNMT attention, we benchmark on *K40m* only:

Systems	1 gpu	4 gpus	8 gpus
NMT (4 layers)	2.2s, 3.4K	1.9s, 3.9K	-
NMT (8 layers)	3.5s, 2.0K	-	2.9s, 2.4K
NMT + GNMT attention (4 layers)	2.6s, 2.8K	1.7s, 4.3K	-
NMT + GNMT attention (8 layers)	4.2s, 1.7K	-	1.9s, 3.8K

These results show that without GNMT attention, the gains from using multiple gpus are minimal.
With GNMT attention, we obtain from 50%-100% speed-ups with multiple gpus.

WMT English-German — Full Comparison

The first 2 rows are our models with GNMT attention: model 1 (4 layers), model 2 (8 layers).

Systems	newstest2014	newstest2015
Ours — NMT + GNMT attention (4 layers)	23.7	26.5
Ours — NMT + GNMT attention (8 layers)	24.4	**27.6**
WMT SOTA	20.6	24.9
OpenNMT (Klein et al., 2017)	19.3	-
tf-seq2seq (Britz et al., 2017)	22.2	25.2
GNMT (Wu et al., 2016)	**24.6**	-

The above results show our models are very competitive among models of similar architectures.
[Note that OpenNMT uses smaller models and the current best result (as of this writing) is 28.4 obtained by the Transformer network (Vaswani et al., 2017) which has a significantly different architecture.]

Standard HParams

We have provided a set of standard hparams for using pre-trained checkpoint for inference or training NMT architectures used in the Benchmark.

We will use the WMT16 German-English data, you can download the data by the following command.

```
nmt/scripts/wmt16_en_de.sh /tmp/wmt16
```

Here is an example command for loading the pre-trained GNMT WMT German-English checkpoint for inference.

```
python -m nmt.nmt \
    --src=de --tgt=en \
    --ckpt=/path/to/checkpoint/translate.ckpt \
    --hparams_path=nmt/standard_hparams/wmt16_gnmt_4_layer.json \
    --out_dir=/tmp/deen_gnmt \
    --vocab_prefix=/tmp/wmt16/vocab.bpe.32000 \
    --inference_input_file=/tmp/wmt16/newstest2014.tok.bpe.32000.de \
    --inference_output_file=/tmp/deen_gnmt/output_infer \
    --inference_ref_file=/tmp/wmt16/newstest2014.tok.bpe.32000.en
```

Here is an example command for training the GNMT WMT German-English model.

```
python -m nmt.nmt \
    --src=de --tgt=en \
    --hparams_path=nmt/standard_hparams/wmt16_gnmt_4_layer.json \
    --out_dir=/tmp/deen_gnmt \
    --vocab_prefix=/tmp/wmt16/vocab.bpe.32000 \
    --train_prefix=/tmp/wmt16/train.tok.clean.bpe.32000 \
    --dev_prefix=/tmp/wmt16/newstest2013.tok.bpe.32000 \
    --test_prefix=/tmp/wmt16/newstest2015.tok.bpe.32000
```

Other resources

For deeper reading on Neural Machine Translation and sequence-to-sequence models, we highly recommend the following materials by Luong, Cho, Manning, (2016); Luong, (2016); and Neubig, (2017).

There's a wide variety of tools for building seq2seq models, so we pick one per language:
Stanford NMT https://nlp.stanford.edu/projects/nmt/ *[Matlab]*
tf-seq2seq https://github.com/google/seq2seq *[TensorFlow]*
Nemantus https://github.com/rsennrich/nematus *[Theano]*
OpenNMT http://opennmt.net/ *[Torch]*
OpenNMT-py https://github.com/OpenNMT/OpenNMT-py *[PyTorch]*

Acknowledgment

We would like to thank Denny Britz, Anna Goldie, Derek Murray, and Cinjon Resnick for their work bringing new features to TensorFlow and the seq2seq library. Additional thanks go to Lukasz Kaiser for the initial help on the seq2seq codebase; Quoc Le for the suggestion to replicate GNMT; Yonghui Wu and Zhifeng Chen for details on the GNMT systems; as well as the Google Brain team for their support and feedback!

References

- Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. ICLR.
- Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. EMNLP.
- Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. NIPS.

BibTex

```
@article{luong17,
  author  = {Minh{-}Thang Luong and Eugene Brevdo and Rui Zhao},
  title   = {Neural Machine Translation (seq2seq) Tutorial},
  journal = {https://github.com/tensorflow/nmt},
  year    = {2017},
}
```