Mannual for Sequence-to-Sequence Generation

Yang Feng

December 15, 2016

1 Introduction

There are many open-source implementations of the sequence-to-sequence generation model, and in this document I will introduce the details of the version implemented with tensorflow. The main idea of the sequence-to-sequence model is based on encoder-decoder framework, that is, all the input words are feed to an encoder to get a tensor to express it together with a tensor for each input word for the attention calculation in decoder. Then the decoder uses the input vector and the attention of all the hidden vectors to generate target works one by one.

2 rnn handbook

2.1 translate.py

1. **train()**

```
get the vocabulary and convert the training set and dev set to ids;
create model;
randomly select a bucket;
randomly select the batch for the bucket for step(..);
step(..);
calculate the loss;
if checkpoint then
   then save the model;
end if
```

2. **read_data(source_path, target_path, max_size=None)**
   Read data from source and target files and put into buckets.

   Args:
   source_path: path to the files with token-ids for the source language.
   target_path: path to the file with token-ids for the target language; it must be aligned with the source file: n-th line contains the desired output for n-th line from the source_path.

max_size: maximum number of lines to read, all other will be ignored; if 0 or None, data files will be read completely (no limit).

Returns:
data_set: a list of length len(_buckets); data_set[n] contains a list of (source, target) pairs read from the provided data files that fit into the n-th bucket, i.e., such that len(source) ¡ _buckets[n][0] and len(target) ¡ _buckets[n][1]; source and target are lists of token-ids.

3. **create_model(session, forward_only)**

```
create an objection of class Seq2SeqModel;
if there exists checkpoint model then
    load it;
else
    initialize the parameters;
end if
```

2.2 data_utils.py

prepare_wmt_data(data_dir, en_vocabulary_size, fr_vocabulary_size, tokenizer=None)
Get WMT data into data_dir, create vocabularies and tokenize data.

Args:
data_dir: directory in which the data sets will be stored.
en_vocabulary_size: size of the English vocabulary to create and use.
fr_vocabulary_size: size of the French vocabulary to create and use.
tokenizer: a function to use to tokenize each data sentence; if None, basic_tokenizer will be used.

Returns:
A tuple of 6 elements:
(1) path to the token-ids for English training data-set,
(2) path to the token-ids for French training data-set,
(3) path to the token-ids for English development data-set,
(4) path to the token-ids for French development data-set,
(5) path to the English vocabulary file,
(6) path to the French vocabulary file.

2.3 seq2seq_model.py

1. **get_batch(self, data, bucket_id)**
Get a random batch of data from the specified bucket, prepare for step.

Args:
data: a tuple of size len(self.buckets) in which each element contains lists

of pairs of input and output data that we use to create a batch.
bucket_id: integer, which bucket to get the batch for.

Returns:
The triple (encoder_inputs, decoder_inputs, target_weights) for the constructed batch that has the proper format to call step(...) later. These three vectors are a list of batch-major vectors.
encoder_inputs, decoder_inputs: is set to the length of the bucket encoder size with pad id.
target_weights: are set to 1 for the real input element and 0 for pad element.

2. **__init__(self, source_vocab_size, target_vocab_size, buckets, size, num_layers, max_gradient_norm, batch_size, learning_rate, learning_rate_decay_factor, use_lstm=False, num_samples=512, forward_only=False)**
Create the model.

Args:
source_vocab_size: size of the source vocabulary.
target_vocab_size: size of the target vocabulary.
buckets: a list of pairs (I, O), where I specifies maximum input length that will be processed in that bucket, and O specifies maximum output length. Training instances that have inputs longer than I or outputs longer than O will be pushed to the next bucket and padded accordingly. We assume that the list is sorted, e.g., [(2, 4), (8, 16)].
size: number of units in each layer of the model.
num_layers: number of layers in the model.
max_gradient_norm: gradients will be clipped to maximally this norm.
batch_size: the size of the batches used during training; the model construction is independent of batch_size, so it can be changed after initialization if this is convenient, e.g., for decoding.
learning_rate: learning rate to start with.
learning_rate_decay_factor: decay learning rate by this much when needed.
use_lstm: if true, we use LSTM cells instead of GRU cells.
num_samples: number of samples for sampled softmax.
forward_only: if set, we do not construct the backward pass in the model.

Details:

```
determine whether to use tf.nn.sampled_softmax_loss(..);
determine which kind of cell to use, LSTM, GRU or MultiRNNCell;
if forward_only then
    create a seq2seq model with decoding;
    determine whether to use output_project(a linear wrapper of the output);
else
    create a seq2seq model without decoding;
    for each bucket do
        clip the gradient;
        update the parameters;
    end for
end if
```

3. **step(self, session, encoder_inputs, decoder_inputs, target_weights, bucket_id, forward_only)**
   Run a step of the model feeding the given inputs.

   Args:
   session: tensorflow session to use.
   encoder_inputs: list of numpy int vectors to feed as encoder inputs.
   decoder_inputs: list of numpy int vectors to feed as decoder inputs.
   target_weights: list of numpy float vectors to feed as target weights.
   bucket_id: which bucket of the model to use.
   forward_only: whether to do the backward step or only forward.

   Returns:
   A triple consisting of gradient norm (or None if we did not do backward), average perplexity, and the outputs.

   Raises:
   ValueError: if length of encoder_inputs, decoder_inputs, or target_weights disagrees with bucket size for the specified bucket_id.

2.4 seq2seq.py

embedding_attention_seq2seq(encoder_inputs, decoder_inputs, cell, num_encoder_symbols, num_decoder_symbols, embedding_size, num_heads=1, output_projection=None, feed_previous=False, dtype=dtypes.float32, scope=None, initial_state_attention=False)
Embedding sequence-to-sequence model with attention.

This model first embeds encoder_inputs by a newly created embedding (of shape [num_encoder_symbols x input_size]). Then it runs an RNN to encode embedded encoder_inputs into a state vector. It keeps the outputs of this RNN at every step to use for attention later. Next, it embeds decoder_inputs by another newly created embedding (of shape [num_decoder_symbols x input_size]). Then it runs attention decoder, initialized with the last encoder state, on embedded decoder_inputs and attending to encoder outputs.

Args: encoder_inputs: A list of 1D int32 Tensors of shape [batch_size].
decoder_inputs: A list of 1D int32 Tensors of shape [batch_size].
cell: rnn_cell. RNNCell defining the cell function and size.
num_encoder_symbols: Integer; number of symbols on the encoder side.
num_decoder_symbols: Integer; number of symbols on the decoder side.
embedding_size: Integer, the length of the embedding vector for each symbol.
num_heads: Number of attention heads that read from attention_states.
output_projection: None or a pair (W, B) of output projection weights and biases; W has shape [output_size x num_decoder_symbols] and B has shape [num_decoder_symbols]; if provided and feed_previous=True, each fed previous output will first be multiplied by W and added B.
feed_previous: Boolean or scalar Boolean Tensor; if True, only the first of decoder_inputs will be used (the "GO" symbol), and all other decoder inputs will be taken from previous outputs (as in embedding_rnn_decoder). If False, decoder_inputs are used as given (the standard decoder case).
dtype: The dtype of the initial RNN state (default: tf.float32).
scope: VariableScope for the created subgraph; defaults to "embedding_attention_seq2seq".
initial_state_attention: If False (default), initial attentions are zero. If True, initialize the attentions from the initial state and attention states.

Returns:
A tuple of the form (outputs, state), where: **outputs:** A list of the same length as decoder_inputs of 2D Tensors with shape [batch_size x num_decoder_symbols] containing the generated outputs.
state: The state of each decoder cell at the final time-step. It is a 2D Tensor of shape [batch_size x cell.state_size].

linear(args, output_size, bias, bias_start=0.0, scope=None) Linear map: $\sum_i (args[i] * W[i])$, where $W[i]$ is a variable.

Args:
args: a 2D Tensor or a list of 2D, batch x n, Tensors.
output_size: int, second dimension of W[i].
bias: boolean, whether to add a bias term or not.
bias_start: starting value to initialize the bias; 0 by default.
scope: (optional) Variable scope to create parameters in.

Returns:
A 2D Tensor with shape [batch x output_size] equal to $\sum_i (args[i] * W[i])$, where $W[i]s$ are newly created matrices.

Raises:
ValueError: if some of the arguments has unspecified or wrong shape.

3 tensorflow handbook

3.1 array_ops

concat(concat_dim, values, name='concat') Concatenates the list of tensors **values** along dimension **concat_dim**.

reshape(tensor, shape, name=None) Given **tensor**, this operation returns a tensor that has the same values as tensor with shape **shape**.

3.2 math_ops

reduce_sum(input_tensor, axis=None, keep_dims=False, name=None, reduction_indices=None)
Reduces input_tensor along the dimensions given in axis. Unless keep_dims is true, the rank of the tensor is reduced by 1 for each entry in axis. If keep_dims is true, the reduced dimensions are retained with length 1. If axis has no entries, all dimensions are reduced, and a tensor with a single element is returned.

3.3 control_flow_ops

cond(pred, fn1, fn2, name=None) Return either fn1() or fn2() based on the boolean predicate **pred**.