Recurrent Neural Networks

Introduction

Take a look at this great article (http://colah.github.io/posts/2015-08-Understanding-LSTMs/) for an introduction to recurrent neural networks and LSTMs in particular.

Language Modeling

In this tutorial we will show how to train a recurrent neural network on a challenging task of language modeling. The goal of the problem is to fit a probabilistic model which assigns probabilities to sentences. It does so by predicting next words in a text given a history of previous words. For this purpose we will use the Penn Tree Bank (https://catalog.ldc.upenn.edu/ldc99t42) (PTB) dataset, which is a popular benchmark for measuring the quality of these models, whilst being small and relatively fast to train.

Language modeling is key to many interesting problems such as speech recognition, machine translation, or image captioning. It is also fun -- take a look here (http://karpathy.github.io/2015/05/21/rnn-effectiveness/).

For the purpose of this tutorial, we will reproduce the results from Zaremba et al., 2014 (http://arxiv.org/abs/1409.2329) (pdf (http://arxiv.org/pdf/1409.2329.pdf)), which achieves very good quality on the PTB dataset.

Tutorial Files

This tutorial references the following files from `models/tutorials/rnn/ptb` in the TensorFlow models repo (https://github.com/tensorflow/models):

File	Purpose
`ptb_word_lm.py`	The code to train a language model on the PTB dataset.

File	Purpose
`reader.py`	The code to read the dataset.

Download and Prepare the Data

The data required for this tutorial is in the `data/` directory of the PTB dataset from Tomas Mikolov's webpage: http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz

The dataset is already preprocessed and contains overall 10000 different words, including the end-of-sentence marker and a special symbol (\<unk>) for rare words. In `reader.py`, we convert each word to a unique integer identifier, in order to make it easy for the neural network to process the data.

The Model

LSTM

The core of the model consists of an LSTM cell that processes one word at a time and computes probabilities of the possible values for the next word in the sentence. The memory state of the network is initialized with a vector of zeros and gets updated after reading each word. For computational reasons, we will process data in mini-batches of size `batch_size`.

The basic pseudocode is as follows:

```
lstm = tf.contrib.rnn.BasicLSTMCell(lstm_size)
# Initial state of the LSTM memory.
state = tf.zeros([batch_size, lstm.state_size])
probabilities = []
loss = 0.0
for current_batch_of_words in words_in_dataset:
    # The value of state is updated after processing each batch of words.
    output, state = lstm(current_batch_of_words, state)

    # The LSTM output can be used to make next word predictions
    logits = tf.matmul(output, softmax_w) + softmax_b
    probabilities.append(tf.nn.softmax(logits))
    loss += loss_function(probabilities, target_words)
```

Truncated Backpropagation

By design, the output of a recurrent neural network (RNN) depends on arbitrarily distant inputs. Unfortunately, this makes backpropagation computation difficult. In order to make the learning process tractable, it is common practice to create an "unrolled" version of the network, which contains a fixed number (`num_steps`) of LSTM inputs and outputs. The model is then trained on this finite approximation of the RNN. This can be implemented by feeding inputs of length `num_steps` at a time and performing a backward pass after each such input block.

Here is a simplified block of code for creating a graph which performs truncated backpropagation:

```
# Placeholder for the inputs in a given iteration.
words = tf.placeholder(tf.int32, [batch_size, num_steps])

lstm = tf.contrib.rnn.BasicLSTMCell(lstm_size)
# Initial state of the LSTM memory.
initial_state = state = tf.zeros([batch_size, lstm.state_size])

for i in range(num_steps):
    # The value of state is updated after processing each batch of words.
    output, state = lstm(words[:, i], state)

    # The rest of the code.
    # ...

final_state = state
```

And this is how to implement an iteration over the whole dataset:

```
# A numpy array holding the state of LSTM after each batch of words.
numpy_state = initial_state.eval()
total_loss = 0.0
for current_batch_of_words in words_in_dataset:
    numpy_state, current_loss = session.run([final_state, loss],
        # Initialize the LSTM state from the previous iteration.
        feed_dict={initial_state: numpy_state, words: current_batch_of_words})
    total_loss += current_loss
```

Inputs

The word IDs will be embedded into a dense representation (see the Vector Representations Tutorial (https://www.tensorflow.org/tutorials/word2vec)) before feeding to the LSTM. This allows the model to efficiently represent the knowledge about particular words. It is also easy to write:

```
# embedding_matrix is a tensor of shape [vocabulary_size, embedding size]
word_embeddings = tf.nn.embedding_lookup(embedding_matrix, word_ids)
```

The embedding matrix will be initialized randomly and the model will learn to differentiate the meaning of words just by looking at the data.

Loss Function

We want to minimize the average negative log probability of the target words:

$$\text{loss} = -\frac{1}{N}\sum_{i=1}^{N} \ln p_{\text{target}_i}$$

It is not very difficult to implement but the function `sequence_loss_by_example` is already available, so we can just use it here.

The typical measure reported in the papers is average per-word perplexity (often just called perplexity), which is equal to

$$e^{-\frac{1}{N}\sum_{i=1}^{N} \ln p_{\text{target}_i}} = e^{\text{loss}}$$

and we will monitor its value throughout the training process.

Stacking multiple LSTMs

To give the model more expressive power, we can add multiple layers of LSTMs to process the data. The output of the first layer will become the input of the second and so on.

We have a class called `MultiRNNCell` that makes the implementation seamless:

```
lstm = tf.contrib.rnn.BasicLSTMCell(lstm_size, state_is_tuple=False)
stacked_lstm = tf.contrib.rnn.MultiRNNCell([lstm] * number_of_layers,
    state_is_tuple=False)
```

```
initial_state = state = stacked_lstm.zero_state(batch_size, tf.float32)
for i in range(num_steps):
    # The value of state is updated after processing each batch of words.
    output, state = stacked_lstm(words[:, i], state)

    # The rest of the code.
    # ...

final_state = state
```

Run the Code

Start by cloning the TensorFlow models repo (https://github.com/tensorflow/models) from GitHub. You'll also need to download the PTB dataset, as discussed at the beginning of this tutorial; we'll assume the dataset is located in **`/tmp/simple-examples/data`**.

Run the following commands:

```
cd models/tutorials/rnn/ptb
python ptb_word_lm.py --data_path=/tmp/simple-examples/data/ --model=small
```

There are 3 supported model configurations in the tutorial code: "small", "medium" and "large". The difference between them is in size of the LSTMs and the set of hyperparameters used for training.

The larger the model, the better results it should get. The **`small`** model should be able to reach perplexity below 120 on the test set and the **`large`** one below 80, though it might take several hours to train.

What Next?

There are several tricks that we haven't mentioned that make the model better, including:

- decreasing learning rate schedule,
- dropout between the LSTM layers.

Study the code and modify it to improve the model even further.

tf.nn.dynamic_rnn

tf.nn.dynamic_rnn

```
dynamic_rnn(
    cell,
    inputs,
    sequence_length=None,
    initial_state=None,
    dtype=None,
    parallel_iterations=None,
    swap_memory=False,
    time_major=False,
    scope=None
)
```

Defined in **tensorflow/python/ops/rnn.py** (https://www.github.com/tensorflow/tensorflow/blob/r1.1/tensorflow/python/ops/rnn.py).

See the guide: Neural Network > Recurrent Neural Networks (https://www.tensorflow.org/api_guides/python/nn#Recurrent_Neural_Networks)

Creates a recurrent neural network specified by RNNCell **cell**.

This function is functionally identical to the function **rnn** above, but performs fully dynamic unrolling of **inputs**.

Unlike **rnn**, the input **inputs** is not a Python list of **Tensors**, one for each frame. Instead, **inputs** may be a single **Tensor** where the maximum time is either the first or second dimension (see the parameter **time_major**). Alternatively, it may be a (possibly nested) tuple of Tensors, each of them having matching batch and time dimensions. The corresponding output is either a single **Tensor** having the same number of time steps and batch size, or a (possibly nested) tuple of such tensors, matching the nested structure of **cell.output_size**.

The parameter **sequence_length** is optional and is used to copy-through state and zero-out outputs when past a batch element's sequence length. So it's more for correctness than performance, unlike in rnn().

Args:

- **`cell`**: An instance of RNNCell.
- **`inputs`**: The RNN inputs.

  If **`time_major == False`** (default), this must be a **`Tensor`** of shape: **`[batch_size, max_time, ...]`**, or a nested tuple of such elements.

  If **`time_major == True`**, this must be a **`Tensor`** of shape: **`[max_time, batch_size, ...]`**, or a nested tuple of such elements.

  This may also be a (possibly nested) tuple of Tensors satisfying this property. The first two dimensions must match across all the inputs, but otherwise the ranks and other shape components may differ. In this case, input to **`cell`** at each time-step will replicate the structure of these tuples, except for the time dimension (from which the time is taken).

  The input to **`cell`** at each time step will be a **`Tensor`** or (possibly nested) tuple of Tensors each with dimensions **`[batch_size, ...]`**. **`sequence_length`***: (optional) An int32/int64 vector sized* **`[batch_size]`**. **`initial_state`**: (optional) An initial state for the RNN. If **`cell.state_size`** is an integer, this must be a **`Tensor`** of appropriate type and shape **`[batch_size, cell.state_size]`**. If **`cell.state_size`** is a tuple, this should be a tuple of tensors having shapes **`[batch_size, s] for s in cell.state_size`**. **`dtype`***: (optional) The data type for the initial state and expected output. Required if initial_state is not provided or RNN state has a heterogeneous dtype.*
  `parallel_iterations`: (Default: 32). The number of iterations to run in parallel. Those operations which do not have any temporal dependency and can be run in parallel, will be. This parameter trades off time for space. Values >> 1 use more memory but take less time, while smaller values use less memory but computations take longer. **`swap_memory`***: Transparently swap the tensors produced in forward inference but needed for back prop from GPU to CPU. This allows training RNNs which would typically not fit on a single GPU, with very minimal (or no) performance penalty.* **`time_major`**: The shape format of the **`inputs`** and **`outputs`** Tensors. If true, these **`Tensors`** must be shaped **`[max_time, batch_size, depth]`**. If false, these **`Tensors`** must be shaped **`[batch_size, max_time, depth]`**. Using **`time_major = True`** is a bit more efficient because it avoids transposes at the beginning and end of the RNN calculation. However, most TensorFlow data is batch-major, so by default this function accepts input and emits output in batch-major form. * **`scope`**: VariableScope for the created subgraph; defaults to "rnn".

Returns:

A pair (outputs, state) where:

```
outputs: The RNN output `Tensor`.

  If time_major == False (default), this will be a `Tensor` shaped:
    `[batch_size, max_time, cell.output_size]`.

  If time_major == True, this will be a `Tensor` shaped:
    `[max_time, batch_size, cell.output_size]`.

  Note, if `cell.output_size` is a (possibly nested) tuple of integers
  or `TensorShape` objects, then `outputs` will be a tuple having the
  same structure as `cell.output_size`, containing Tensors having shapes
  corresponding to the shape data in `cell.output_size`.

state: The final state.  If `cell.state_size` is an int, this
  will be shaped `[batch_size, cell.state_size]`.  If it is a
  `TensorShape`, this will be shaped `[batch_size] + cell.state_size`.
  If it is a (possibly nested) tuple of ints or `TensorShape`, this will
  be a tuple having the corresponding shapes.
```

Raises:

- **`TypeError`**: If `cell` is not an instance of RNNCell.
- **`ValueError`**: If inputs is None or an empty list.

Last updated March 8, 2017.