Recurrent Neural Networks

Introduction

Take a look at this great article 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 probablities 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 (PTB) dataset, which is a popular benchmark for measuring 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, too -- take a look here.

For the purpose of this tutorial, we will reproduce the results from Zaremba et al., 2014, which achieves very good results on the PTB dataset.

Tutorial Files

This tutorial references the following files from `models/rnn/ptb`:

File	Purpose
`ptb_word_lm.py`	The code to train a language model on the PTB dataset.
`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. We convert all of them in the `reader.py` to unique integer identifiers to make it easy for the neural network to process.

The Model

LSTM

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

The basic pseudocode looks as follows:

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

loss = 0.0
for current_batch_of_words in words_in_dataset:
    # The value of state is updated after processing each batch of wo
rds.
    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 = tf.nn.softmax(logits)
    loss += loss_function(probabilities, target_words)
```

Truncated Backpropagation

In order to make the learning process tractable, it is a common practice to truncate the

gradients for backpropagation to a fixed number (`num_steps`) of unrolled steps. This is easy to implement by feeding inputs of length `num_steps` at a time and doing backward pass after each iteration.

A simplifed version of the code for the graph creation for truncated backpropagation:

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

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

for i in range(len(num_steps)):
    # The value of state is updated after processing each batch of wo
rds.
    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_o
f_words})
    total_loss += current_loss
```

Inputs

The word IDs will be embedded into a dense representation (see the Vectors

Representations Tutorial) 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 s
ize]
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 Fuction

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 seemless:

```
lstm = rnn_cell.BasicLSTMCell(lstm_size)
stacked_lstm = rnn_cell.MultiRNNCell([lstm] * number_of_layers)
```

```
initial_state = state = stacked_lstm.zero_state(batch_size, tf.float3
2)
for i in range(len(num_steps)):
    # The value of state is updated after processing each batch of wo
rds.
    output, state = stacked_lstm(words[:, i], state)

    # The rest of the code.
    # ...

final_state = state
```

Compile and Run the Code

First, the library needs to be built. To compile it on CPU:

```
bazel build -c opt tensorflow/models/rnn/ptb:ptb_word_lm
```

And if you have a fast GPU, run the following:

```
bazel build -c opt --config=cuda tensorflow/models/rnn/ptb:ptb_word_l
m
```

Now we can run the model:

```
bazel-bin/tensorflow/models/rnn/ptb/ptb_word_lm \
  --data_path=/tmp/simple-examples/data/ --alsologtostderr --model sm
all
```

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.