Machine Learning
Brendan Shillingford
July 6, 2016

Practical 6: LSTMs for language modelling

Logistics

Since this is the last week for practicals, it will be **extremely short** and does not require writing code, and is due by the end of the Friday's session (regardless of whether you are from the Wednesday or Friday session).

Technical aspects of practical (do this before reading)

Clone the practical **and** download the associated data:

```
git clone https://github.com/oxford-cs-ml-2015/practical6.git
cd practical6
wget http://www.cs.ox.ac.uk/people/brendan.shillingford/teaching/practical6-data.tar.gz
tar xvf practical6-data.tar.gz
```

and start training the model:

```
th train.lua -vocabfile vocab.t7 -datafile train.t7
```

Make note of the time at which you run the `train.lua` script. Every several iterations, the training script will save the current model (including its parameters) to a file called `model_autosave.t7`. You can make snapshots of this file if you want, but this is not required for the practical.

Running the sampler script

Run

```
th sample.lua -vocabfile vocab.t7 -model model_autosave.t7 -sample -primetext " "
```

to sample from the model named `model_autosave.t7`. This is just a serialized copy of the whole network (embedding layer, LSTM and its internal parameters, and the linear and softmax on the output). We discuss the network in more detail below.

Handin: take a look at the samples shortly after training, and observe their quality (i.e. how much they qualitatively look like English). Try again 20 minutes after or later, and compare. How do the samples differ? You may change the random seed using `-seed` if you want the sampling to be deterministic on each run of `sample.lua`.

UNIVERSITY OF OXFORD

Machine Learning
Brendan Shillingford
July 6, 2016

Introduction

This week, we will train a character-level LSTM language model on a small amount of news data (from the Billion Word Corpus, which in turn comes from the WMT 2011 News Crawl corpus: http://arxiv.org/abs/1312.3005).

In language modelling, the task is to model the probability of sequences of tokens, usually words but in this case characters (for time and computational resource constraints in the lab). More precisely, for a sequence of T words $w_1, \ldots, w_T$, we define its probability as

$$P(w_1, \ldots, w_T) = \prod_{t=1}^{T} P(w_t | w_{1:(t-1)})$$

where $1 : (t-1)$ denotes the sequence of indices 1 up to $t-1$. Hence we hope the model will assign high probabilities to character sequences like `the weather is` but low probabilities to character sequences like `lzzsjdrfzzzzz`.

Overview of LSTMs

As we have seen in the lecture, recurrent neural networks (RNNs) are a powerful model for sequential data but suffer from the *exploding and vanishing gradient* problem that makes them difficult to train in practice. More precisely, a sequence of derivatives less than 1 will vanish exponentially quickly with the length of the time lag, while a sequence of derivatives greater than 1 will cause the resultant gradient to explode. This may cause the gradient to decay faster than the duration between two related events in the input, such as the duration between an open and close parenthesis.

An LSTM is a RNN architecture that provides a solution to this problem: when derivatives values are back-propagated from the output, derivatives are able to propagate arbitrarily far back without decaying significantly, as we will see below. This will allow the model to learn longer-term dependencies than a traditional RNN.

An LSTM is described by the following equations:

- Input gate: Controls how much of the current input $\mathbf{x}_t$ and the previous output $\mathbf{h}_{t-1}$ will enter into the new cell
$$\begin{aligned}\mathbf{i}_t &= \sigma(\mathbf{W}_{xi}\mathbf{x}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i) \\ &= \sigma(\text{linear}_{xi}(\mathbf{x}_t) + \text{linear}_{hi}(\mathbf{h}_{t-1}))\end{aligned}$$
- Forget (reset) gate: Decides whether to erase (set to zero) or keep individual components of the memory
$$\mathbf{f}_t = \sigma(\mathbf{W}_{xf}\mathbf{x}_t + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f)$$
- Cell update (input) transformation: transforms the input and previous state to be taken into account into the current state
$$\mathbf{g}_t = \tanh(\mathbf{W}_{xg}\mathbf{x}_t + \mathbf{W}_{hg}\mathbf{h}_{t-1} + \mathbf{b}_g)$$

- Cell state update step: computes the next timestep's state using the gated previous state and the gated input
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t$$
- Output gate: Scales the output from the cell
$$\mathbf{o}_t = \sigma(\mathbf{W}_{xo}\mathbf{x}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o)$$
- Final output of the LSTM: Output of the LSTM scaled by a tanh (squashed) transformation of the current state
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

Note that this solves the vanishing gradient problem, for $k < t$, as

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_k} = \frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} \frac{\partial \mathbf{c}_{t-1}}{\partial \mathbf{c}_{t-2}} \cdots \frac{\partial \mathbf{c}_{k+1}}{\partial \mathbf{c}_k} = \operatorname{diag}(\mathbf{f}_t) \cdot \operatorname{diag}(\mathbf{f}_{t-1}) \cdots \operatorname{diag}(\mathbf{f}_{k+1}) = \operatorname{diag}(\mathbf{f}_t \odot \mathbf{f}_{t-1} \odot \cdots \odot \mathbf{f}_{k+1})$$

This shows there is a clear path for the gradient to flow mostly uninterrupted (i.e. without decaying towards 0), except for the constant (wrt $\mathbf{c}_{t-1}$) multiplicative factor from the forget gate which will usually be near $\mathbf{1}$ and hence not cause much decay, except when the forget gate resets the cell state during the forward pass, in which case the gradient should not flow back anyway.

In contrast, a traditional RNN's main path for the gradient to flow will involve derivatives of sigmoids and derivatives of weight matrices, both of which are much less likely to be close to 1. A good exercise is to write this derivative out and verify this for yourself.

Forward pass

We will now describe how a forward pass takes place in our LSTM model.

Our model has 3 components:

1. the LSTM module,
2. an embedding module to map words (or in this case, characters) to continuous vectors, and
3. a linear module followed by a softmax, mapping the LSTM's output $\mathbf{h}_t$ to a probability distribution.

Preparation: we unroll the LSTM to T timesteps, with each copy of the LSTM timestep sharing weights (i.e. W_{xi}, W_{xo}, W_{xf} etc.) with the other timesteps. We do the same for the linear, softmax and embedding layers to produce T copies that share weights.

Forward pass 1. pass the inputs through the embedding layers, to get T embeddings 2. for $t = 1..T$ do: 1. run a timestep of the LSTM, using the embedding at time t as input 2. get the output of the softmax at time t, which predicts which symbol should be output at time t

A backward pass is now easy: we just take the forward pass and perform it completely in reverse, calling `backward` instead of `forward`, and keeping track of the `gradInput` that each `backward()` call returns, and passing it to the next `backward` call. See `train.lua` for implementation details.

Sampling from the LSTM

At each time step the LSTM outputs a probability distribution over the characters, via the softmax.

Given each timestep's distribution, there are several methods to obtain a single character, which gets embedded and fed back into the LSTM as input to the next timestep: 1. taking the maximum at the current timestep (try running `sample.lua` without `-sample` to see this) 2. sampling from the distribution given by the softmax 3. taking the top k maxima and doing a beam search, which is not implemented here.

Evaluating the quality of the samples

The easiest method: visualize the data! Words like "the" or "and" will appear frequently even after a very short amount of training. As training continues, the samples will (noticeably!) look increasingly like English. This is what we asked you to do in the first section, and is **sufficient for this practical's handin.**

For a more formal evaluation of the performance one can measure the **perplexity** on an unseen test set. Perplexity is the inverse probability of the test set, normalized by the number of words. Minimizing perplexity is the same as maximizing the probability, thus a lower perplexity model better describes the data. Perplexity is a frequently used measure for evaluating language models and has the following formula:

$$PP = P(w_1 w_2 ... w_N)^{\frac{-1}{N}} = \left(\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 w_2 ... w_{i-1})} \right)^{\frac{1}{N}}$$

where $w_1, \ldots, w_N$ is the entire sequence of data that we are training on; we treat the whole thing as one sequence. In a bigram model, the conditions are truncated,

$$PP = \left(\prod_{i=1}^{N} \frac{1}{P(w_i | w_{i-1})} \right)^{\frac{1}{N}}$$

but in the recurrent net we depend on the entire past as in the first formula.

In the practical, however, we will not measure perplexity as we don't have a test set, and we are training a character-level model. Perplexity is usually used on word-level language models.

Handin

Handin: see first section above

If you finish early

Nothing! :) Unless you're interested in using LSTMs in your own work, in which case you should keep reading.

But if you want to do something, start by understanding the code in detail. One thing to understand well is how to sample from the LSTM, or produce a sequence taking the maximum at each step. Once you understand the forward pass, the backward pass is easy: we just reverse it.

If you want a further exercise, try modelling some other kind of sequence, and perhaps turn it into a larger project. Or, perhaps you may try making changes to the architecture of the LSTM to improve its performance, for instance http://arxiv.org/abs/1502.02367. We don't have any test evaluation procedure here, so that would need to be added first.