Adam Paszke

LSTM implementation explained

Aug 30, 2015

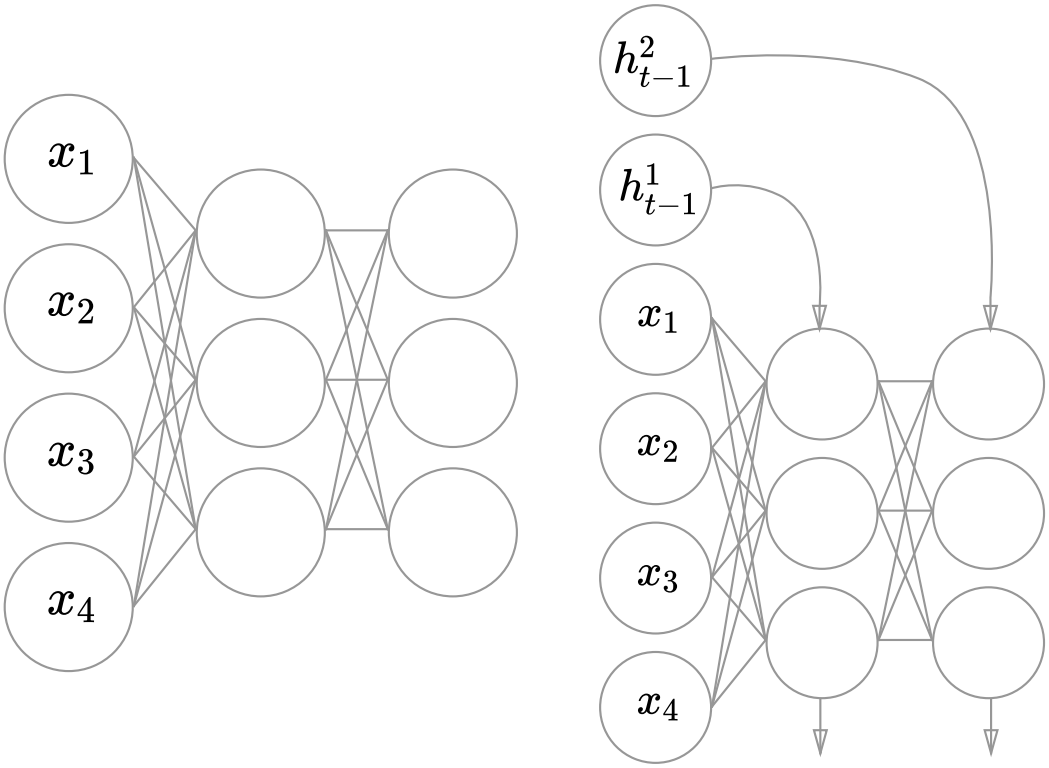
Preface

For a long time I've been looking for a good tutorial on implementing LSTM networks. They seemed to be complicated and I've never done anything with them before. Quick googling didn't help, as all I've found were some slides.

Fortunately, I took part in Kaggle EEG Competition and thought that it might be fun to use LSTMs and finally learn how they work. I based my solution and this post's code on char-rnn by Andrej Karpathy, which I highly recommend you to check out.

RNN misconception

There is one important thing that as I feel hasn't been emphasized strongly enough (and is the main reason why I couldn't get myself to do anything with RNNs). There isn't much difference between an RNN and feedforward network implementation. It's the easiest to implement an RNN just as a feedforward network with some parts of the input feeding into the middle of the stack, and a bunch of outputs coming out from there as well. There is no magic internal state kept in the network. It's provided as a part of the input!



The overall structure of RNNs is very similar to that of feedforward networks.

LSTM refresher

This section will cover only the formal definition of LSTMs. There are lots of other nice blog posts describing in detail how can you imagine and think of these equations.

LSTMs have many variations, but we'll stick to a simple one. One cell consists of three gates (input, forget, output), and a cell unit. Gates use a sigmoid activation, while input and cell state is often transformed with tanh. LSTM cell can be defined with a following set of equations:

Gates:

$$i_t = g(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$

$$f_t = g(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$$

$$o_t = g(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$$

Input transform:

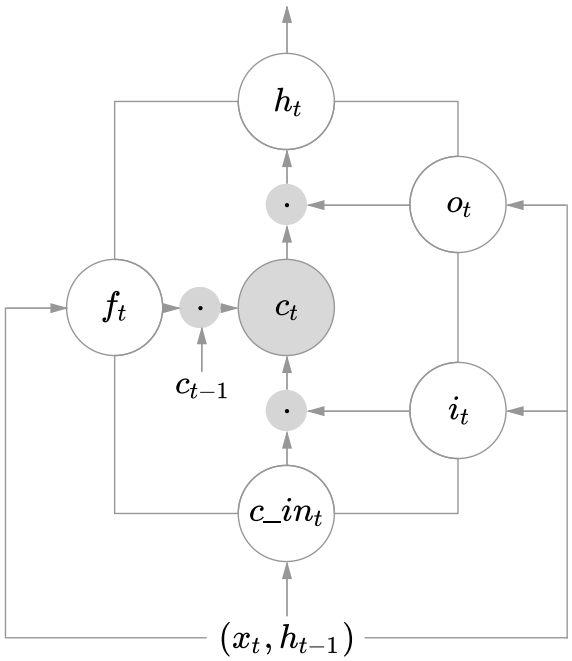
$$c\_in_t = tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_{c\_in})$$

State update:

$$c_t = f_t \cdot c_{t-1} + i_t \cdot c\_in_t$$

$$h_t = o_t \cdot tanh(c_t)$$

It can be pictured like this:



Because of the gating mechanism the cell can keep a piece of information for long periods of time during work and protect the gradient inside the cell from harmful changes during the training. Vanilla LSTMs don't have a forget gate and add unchanged cell state during the update (it can be seen as a recurrent connection with a constant weight of 1), what is often referred to as a Constant Error Carousel (CEC). It's called like that, because it solves a serious RNN training problem of vanishing and exploding gradients, which in turn makes it possible to learn long-term relationships.

Building your own LSTM layer

The code for this tutorial will be using Torch7. **Don't worry if you don't know it**. I'll explain everything, so you'll be able to implement the same algorithm in your favorite framework.

The network will be implemented as a `nngraph.gModule`, which basically means that we'll define a computation graph consisting of standard `nn` modules. We will need the following layers:

- `nn.Identity()` - passes on the input (used as a placeholder for input)
- `nn.Dropout(p)` - standard dropout module (drops with probability `1 - p`)
- `nn.Linear(in, out)` - an affine transform from `in` dimensions to `out` dims
- `nn.Narrow(dim, start, len)` - selects a subvector along `dim` dimension having `len` elements starting from `start` index
- `nn.Sigmoid()` - applies sigmoid element-wise

- `nn.Tanh()` - applies tanh element-wise
- `nn.CMulTable()` - outputs the product of tensors in forwarded table
- `nn.CAddTable()` - outputs the sum of tensors in forwarded table

Inputs

First, let's define the input structure. The array-like objects in lua are called tables. This network will accept a table of tensors like the one below:

$$\{\text{input}, c^1_{t-1}, h^1_{t-1}\}$$

```
local inputs = {}
table.insert(inputs, nn.Identity()())   -- network input
table.insert(inputs, nn.Identity()())   -- c at time t-1
table.insert(inputs, nn.Identity()())   -- h at time t-1
local input = inputs[1]
local prev_c = inputs[2]
local prev_h = inputs[3]
```

Identity modules will just copy whatever we provide to the network into the graph.

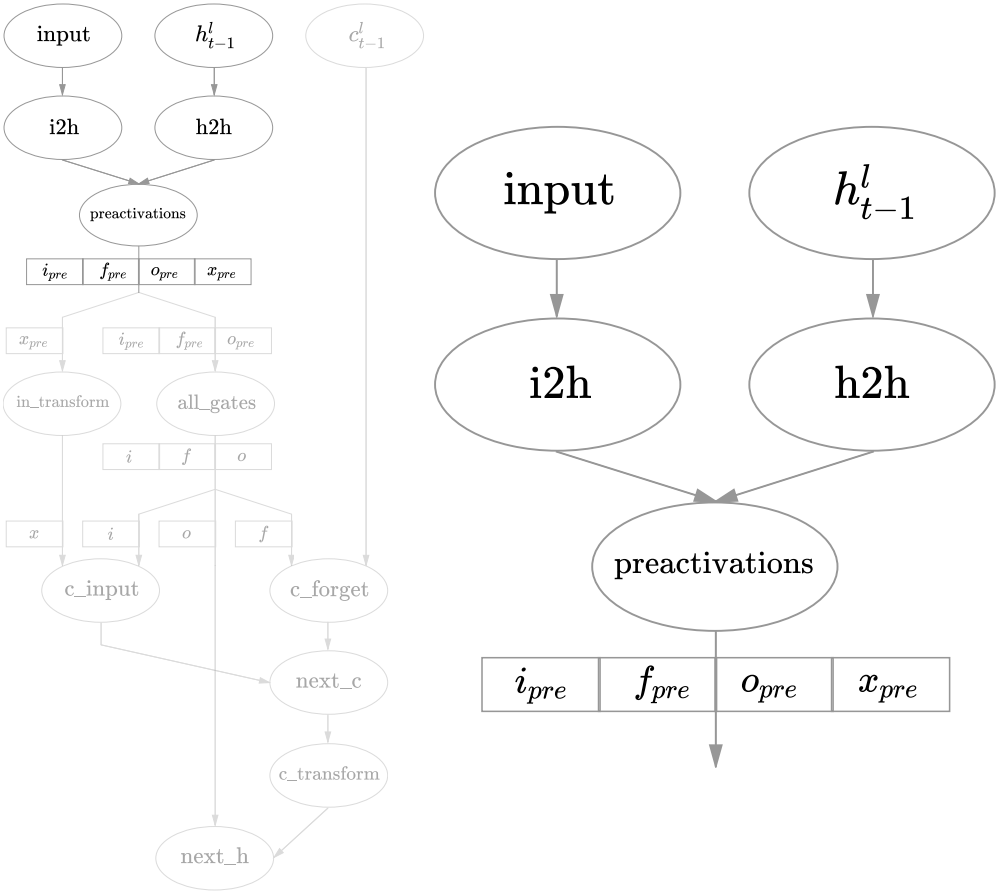
Computing gate values

To make our implementation faster we will be applying the transformations of the whole LSTM layer simultaneously.

```
local i2h = nn.Linear(input_size, 4 * rnn_size)(input)  -- input to hidc
local h2h = nn.Linear(rnn_size, 4 * rnn_size)(prev_h)   -- hidden to hic
local preactivations = nn.CAddTable()({i2h, h2h})       -- i2h + h2h
```

If you're unfamiliar with `nngraph` it probably seems strange that we're constructing a module and already calling it once more with a graph node. What actually happens is that the second call converts the `nn.Module` to `nngraph.gModule` and the argument specifies it's parent in the graph.

`preactivations` outputs a vector created by a linear transform of input and previous hidden state. These are raw values which will be used to compute the gate activations and the cell input. This vector is divided into 4 parts, each of size `rnn_size`. The first will be used for in gates, second for forget gates, third for out gates and the last one as a cell input (so the indices of respective gates and input of a cell number i are $\{i,\ \text{rnn\_size} + i,\ 2 \cdot \text{rnn\_size} + i,\ 3 \cdot \text{rnn\_size} + i\}$).



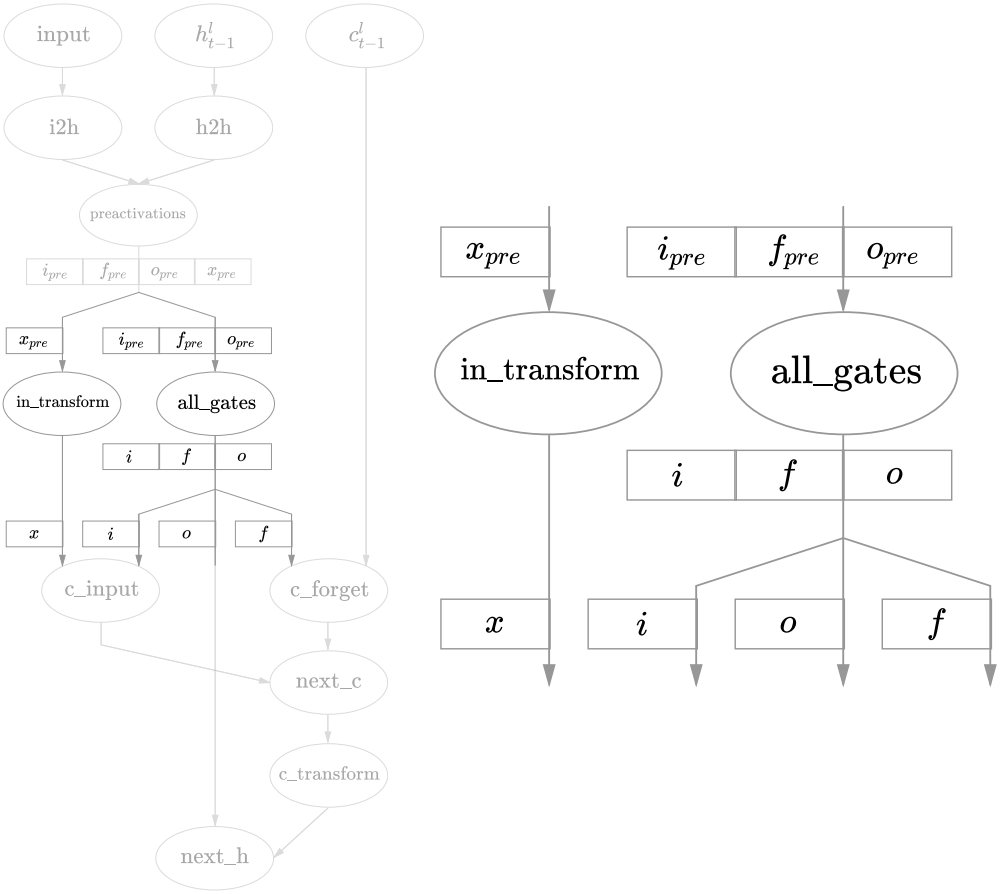
Next, we have to apply a nonlinearity, but while all the gates use the sigmoid, we will use a tanh for the input preactivation. Because of this, we will place two `nn.Narrow` modules, which will select appropriate parts of the preactivation vector.

```
-- gates
local pre_sigmoid_chunk = nn.Narrow(2, 1, 3 * rnn_size)(preactivations)
local all_gates = nn.Sigmoid()(pre_sigmoid_chunk)

-- input
local in_chunk = nn.Narrow(2, 3 * rnn_size + 1, rnn_size)(preactivations
local in_transform = nn.Tanh()(in_chunk)
```

After the nonlinearities we have to place a couple more `nn.Narrow`s and we have the gates done!

```
local in_gate = nn.Narrow(2, 1, rnn_size)(all_gates)
local forget_gate = nn.Narrow(2, rnn_size + 1, rnn_size)(all_gates)
local out_gate = nn.Narrow(2, 2 * rnn_size + 1, rnn_size)(all_gates)
```



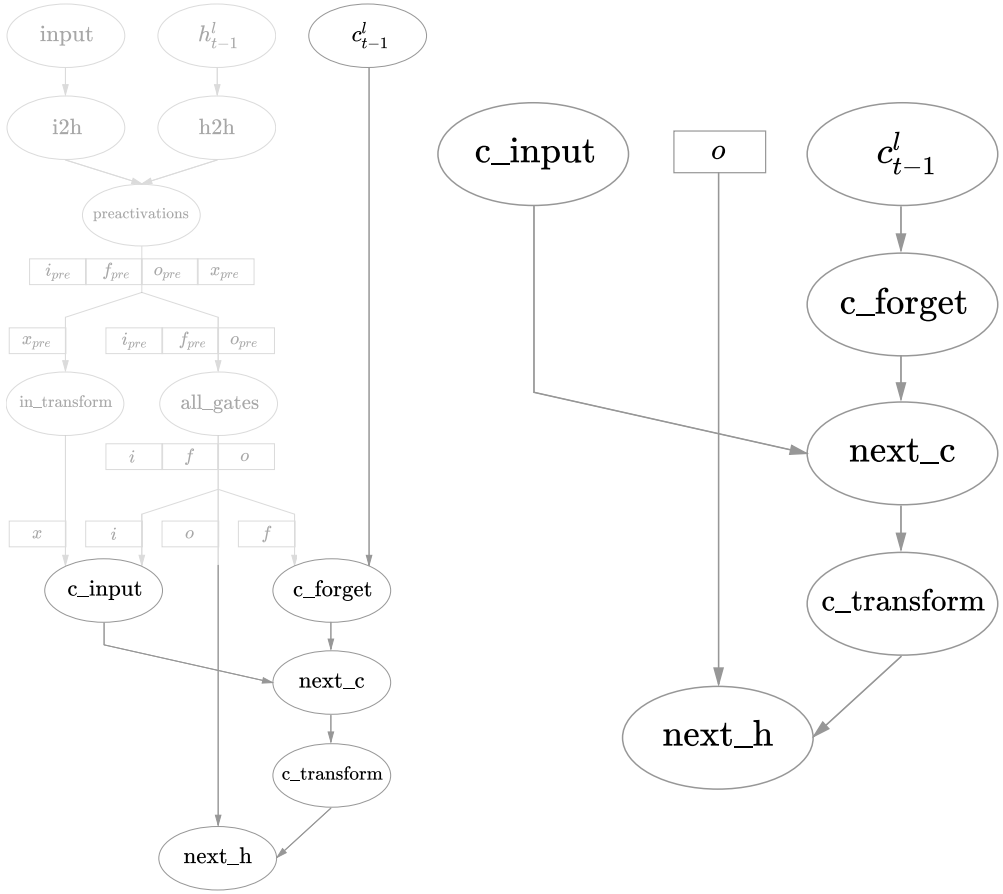
Cell and hidden state

Having computed the gate values we can now calculate the current cell state. All that's required are just two `nn.CMulTable` modules (one for $f \cdot c^l_{t-1}$ and one for $i \cdot x$), and a `nn.CAddTable` to sum them up to a current cell state.

```
-- previous cell state contribution
local c_forget = nn.CMulTable()({forget_gate, prev_c})
-- input contribution
local c_input = nn.CMulTable()({in_gate, in_transform})
-- next cell state
local next_c = nn.CAddTable()({
  c_forget,
  c_input
})
```

It's finally time to implement hidden state calculation. It's the simplest part, because it just involves applying tanh to current cell state (`nn.Tanh`) and multiplying it with an output gate (`nn.CMulTable`).

```
local c_transform = nn.Tanh()(next_c)
local next_h = nn.CMulTable()({out_gate, c_transform})
```



Defining the module

Now, if you want to export the whole graph as a standalone module you can wrap it like that:

```
-- module outputs
outputs = {}
table.insert(outputs, next_c)
table.insert(outputs, next_h)

-- packs the graph into a convenient module with standard API (:forward(
return nn.gModule(inputs, outputs)
```

Examples

LSTM layer implementation is available here. You can use it like that:

```
th> LSTM = require 'LSTM.lua'
                                                                    [0
th> layer = LSTM.create(3, 2)
                                                                    [0
th> layer:forward({torch.randn(1,3), torch.randn(1,2), torch.randn(1,2)})
{
```

```
  1 : DoubleTensor - size: 1x2
  2 : DoubleTensor - size: 1x2
}
                                                                    [0
```

To make a multi-layer LSTM network you can forward subsequent layers in a for loop, taking `next_h` from previous layer as next layer's input. You can check this example.

Training

If you're interested please leave a comment and I'll try to expand this post!

That's it!

That's it. It's quite easy to implement any RNN when you understand how to deal with the hidden state. After connecting several layers just put a regular MLP on top and connect it to last layer's hidden state and you're done!

Here are some nice papers on RNNs if you're interested:

- Visualizing and Understanding Recurrent Networks
- An Empirical Exploration of Recurrent Network Architectures
- Recurrent Neural Network Regularization
- Sequence to Sequence Learning with Neural Networks

Ayana Altaqin · 3 days ago

Well thanks for the sharing, I just started to learn these RNN,LSTM,Lua and Torch things... And they are really hard to understand.. Your work is very helpful and I do really appreciate it .. Yes , thank you ...

Ozan Çağlayan · 18 days ago

I think CMulTable() should be the product while the CAddTable() the sum in their descriptions.

Thanks :)

Adam Paszke Mod → Ozan Çağlayan · 16 days ago

You've got me, thanks! ;)

joe · 19 days ago

The figures is very clear and this page makes the LSTM very easy to understand, I certainly got lot of things from this article, it makes sense. What's more, I'm also curious about the bp procedure of LSTM, maybe there will be some supplyment posts later ^ ^

Adam Paszke Mod → joe · 16 days ago

Hey! Sorry for the late reply!

It's great to hear that you liked it! Saying 'bp procedure' do you mean the exact derivations or just the training code? It's really easy to train these models with a few calls like :backward(). There are some examples in torch docs e.g.: https://github.com/torch/nn/bl...