Adam Paszke

LSTM implementation explained

Aug 30, 2015

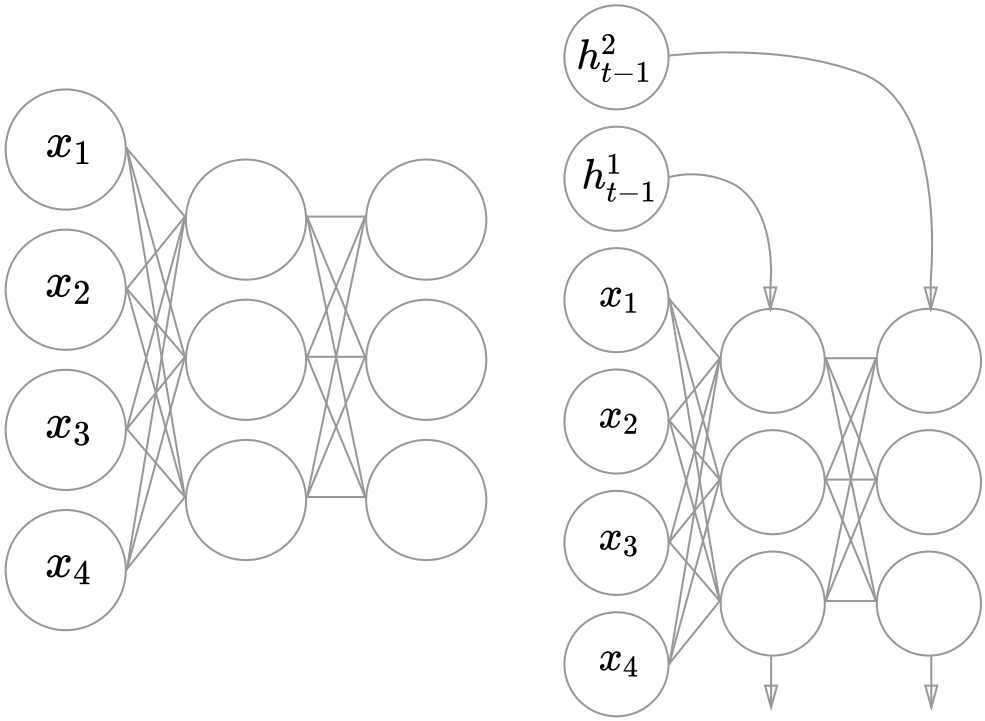
Preface

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:

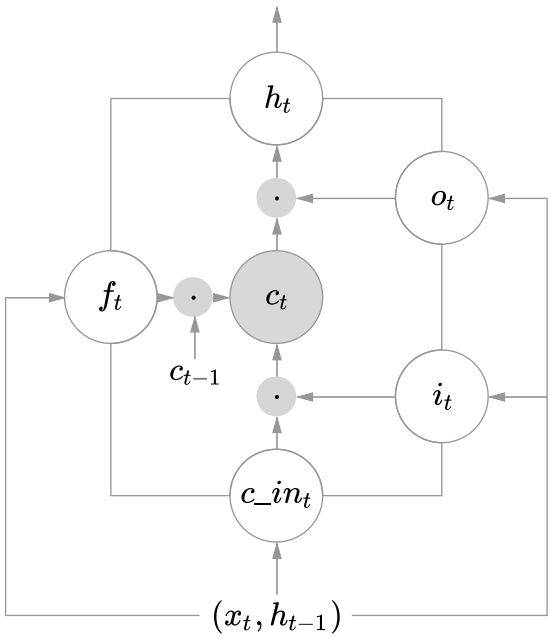
$$c\_in_t = tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_{c\_in})$$

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

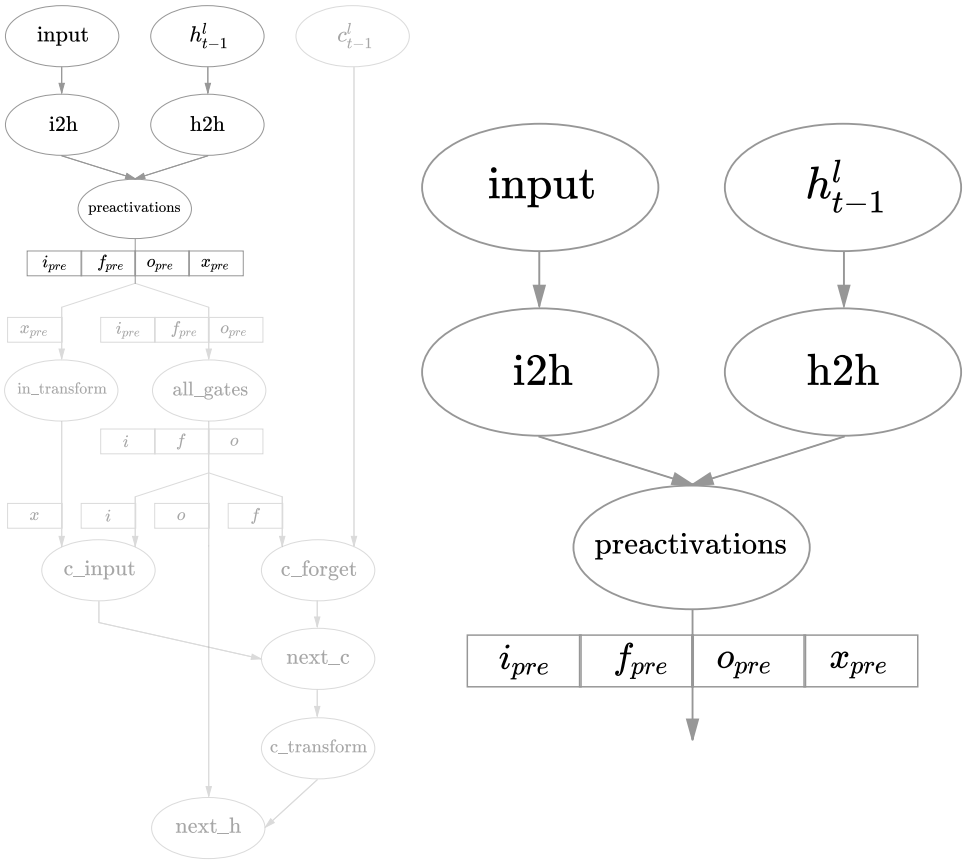
To make our implementation faster we will be applying the transformations of the whole LSTM layer simultaneously.

```
local i2h = nn.Linear(input_size, 4 * rnn_size)(input)  -- input to hidden
local h2h = nn.Linear(rnn_size, 4 * rnn_size)(prev_h)   -- hidden to hidden
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.

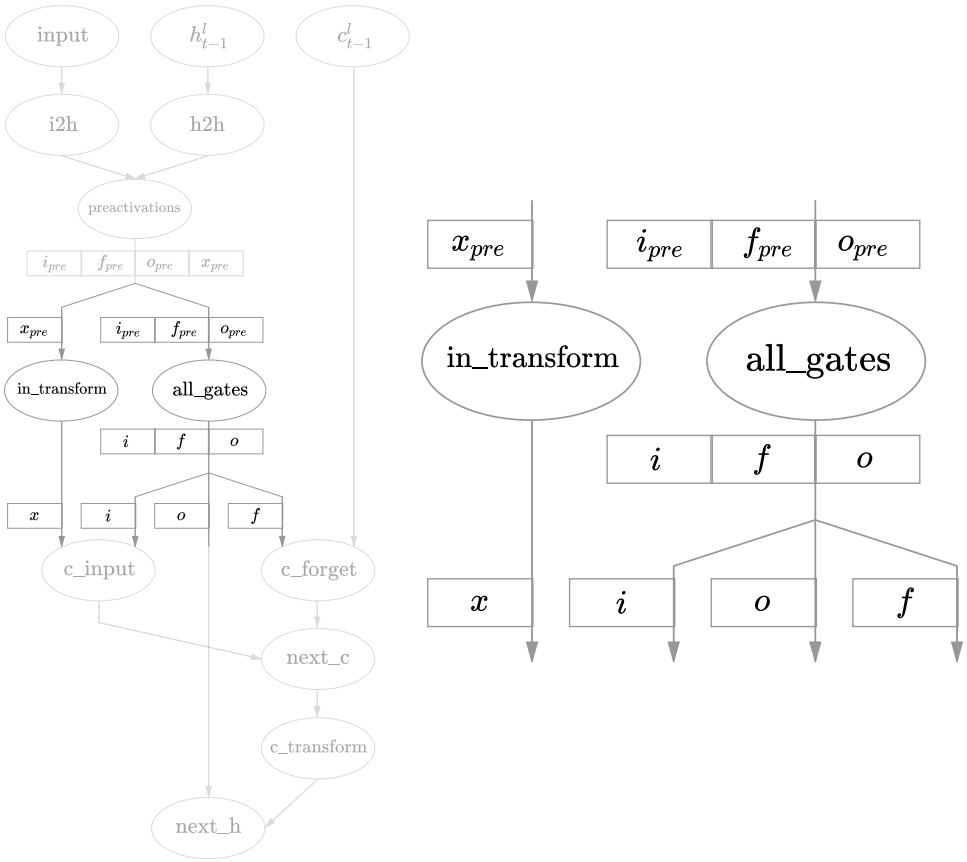
```
-- gates
local pre_sigmoid_chunk = nn.Narrow(2, 1, 3 * rnn_size)(preactivations)
local all_gates = nn.Sigmoid()(pre_sigmoid_chunk)

-- 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)
```



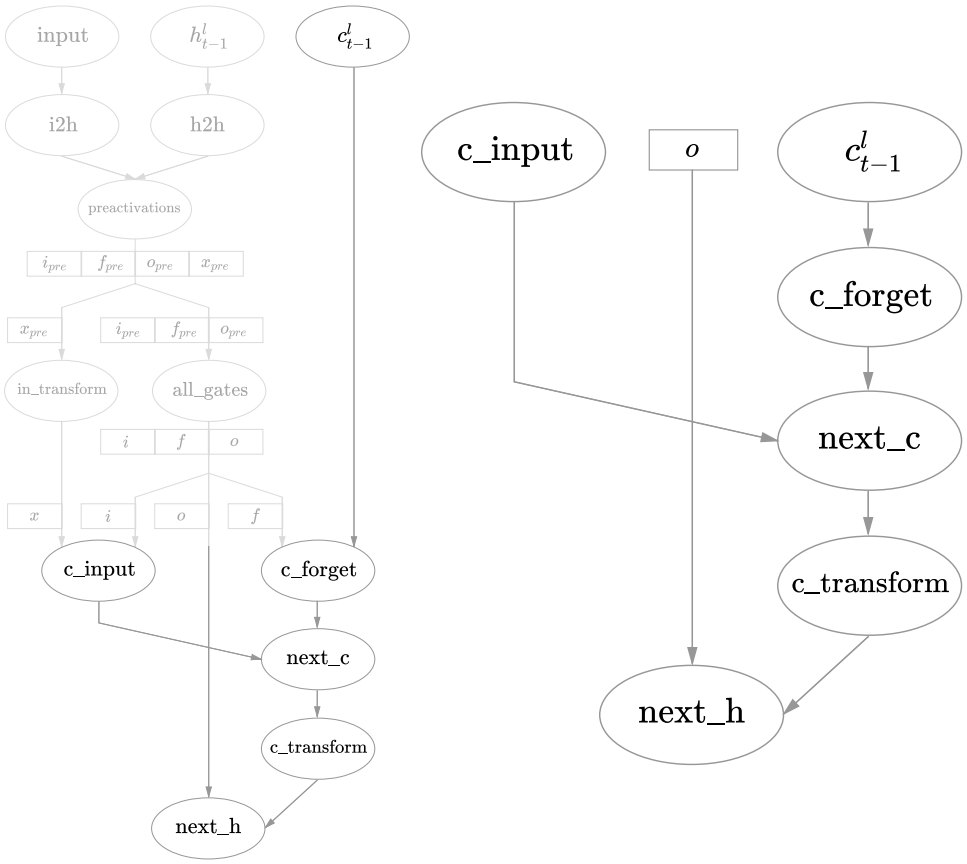
Cell and hidden state

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(), :backw
return nn.gModule(inputs, outputs)
```

Examples

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

```
th> LSTM = require 'LSTM.lua'
                                                                    [0.0224s]
th> layer = LSTM.create(3, 2)
                                                                    [0.0019s]
th> layer:forward({torch.randn(1,3), torch.randn(1,2), torch.randn(1,2)})
{
  1 : DoubleTensor - size: 1x2
  2 : DoubleTensor - size: 1x2
}
                                                                    [0.0005s]
```

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

Aravind Srinivas L • a year ago

Hi, can you please update this with the training code too?
Also, would be great if you could have another article explaining how to build LSTM networks using the 'rnn' package (with nn.Sequencer and stuff), and how it compares to using nngraph.

Thanks for the really good explanation about building the LSTM!

3 ^ v • Reply • Share ›

Adam Paszke Mod → Aravind Srinivas L • a year ago

Hi Aravind. Sorry for a late reply. I haven't used rnn package yet, because nngraph fitted nicely into my models so far. I'm planning to do some extensions to this post, which include adding a section on training LSTM networks.

^ v • Reply • Share ›

Bruce Zhang → Aravind Srinivas L • a year ago

I think there is no such thing as rnn package, you have to build RNN with nngraph.

^ v • Reply • Share ›

Adam Paszke Mod → Bruce Zhang • a year ago

In fact there is an rnn package by Nicholas Léonard, but it's not included in torch core. See this repository.

^ v • Reply • Share ›

soshiant • a year ago

hi great stuff.
i'm gonno ask this just to be sure :
you have 3 inputs for forward pass
so f i want to us it in multiple steps i have to give this layer the resuls of the last step for two of those inputs right?
this is making it too hard could you just take the input and somehow insert the ht-1 ?

1 ^ v • Reply • Share ›

Adam Paszke Mod → soshiant • a year ago

Yes, two of the outputs are inputs in the next time step. Unfortunately, I don't think it can be done in a simpler way.

^ v • Reply • Share ›

Igor Chernobaev • 9 days ago

Hi, i'm trying to implement own LSTM for better understanding. Unfortunately my backprop algorithm doesn't pass gradient check. Could you kindly help me?

^ v • Reply • Share ›

Faiyaz Lalani • 4 months ago

Excellent post - very helpful. Much appreciated Adam!

Reply • Share ›

fnm • 5 months ago

I am interested in knowing that how lstm model is trained.

Reply • Share ›

Yitzhak Spielberg • 8 months ago

It would be cool to have the training procedure as well

Reply • Share ›

Yitzhak Spielberg • 8 months ago

Hi, how do I build a model with multiple lstm layers? for ordinary neural nets I used nn.sequential to build the model and model:forward(x) to calculate the output. How does it work in this case? A simple example, let's say with 2 lstm layers would be really helpful.

Reply • Share ›

> **Yitzhak Spielberg** ➚ Yitzhak Spielberg • 8 months ago
>
> solved. just found the multilayer example above.
>
> Reply • Share ›

Mohammed Jabreel • 8 months ago

Hi, Thank you for this nice post about LSTM, I would like to explain me how we could implement the LSTM attention.

Reply • Share ›

simon • 9 months ago

Nice LSTM post, but I have a question,how the gate units can protect the gradient inside the cell from harmful changes during the training, if there existes some explanation in math?

Reply • Share ›

Luca Naterop • 9 months ago

Noob question: How can I compute the RNN output with this? layer:forward() only gives the updated hidden and cell state...but how do I get the networks predictions?

Reply • Share ›

Gaurav Pandey • a year ago

Hi, nice article. The LSTM obtained above won't have tied weights. Hence, I had 2 questions.

1) Is it more common to have tied weights between different LSTM units.
2) Can this simply be achieved by getting the parameters pointer of the first unit, and setting the parameters of all successive units to point to that pointer (Similar for gradient)

Reply • Share ›

Mahati • a year ago

Hi. Thanks for this! It would be great if you could also update the training code. This has been very helpful for me in learning about LSTM.

Reply • Share ›

Adam Paszke Mod → Mahati • a year ago

Hi Mahati. Great to hear that it was helpful for you. I am planning to expand this post in the near future and there will be some training examples provided.

Reply • Share ›

Marco Damonte • a year ago

Once you have your "layer" variable, can you use it as a traditional nn.Module and start the training?

Reply • Share ›

Adam Paszke Mod → Marco Damonte • a year ago

Yes! gModule has the same API as all nn modules.

Reply • Share ›

Aiqun Huang • a year ago

@Adam Paszke, what is the dimension of preactivations? I think it is just a 1D vector of size 4 * rnn_size, but why the first argument of nn.Narrow(2, 1, 3 * rnn_size) is 2?

Reply • Share ›

Aiqun Huang → Aiqun Huang • a year ago

@Adam Paszke, I think I got it. The 1st argument of nn.Narrow should be 2, which means along the column dimension choosing 3*rnn_size columns, and in the case of preactivations, each column has only 1 element.

Reply • Share ›

Adam Paszke Mod → Aiqun Huang • a year ago

Yeah. This module runs in batch mode only, so even if you're only doing one step at a time, you should input a 1xN matrix. The activations for respective cells are in the columns :)

In this way you can train or predict with LSTMs on several inputs in parallel (if you add more rows).

1 Reply • Share ›

Maxim Ivanov • a year ago

Thanks **@Adam Paszke** for such detailed step-by-step LSTM explanation. I have a question. In original paper and other sources there is nothing like rnn_size (basically, rnn_size equals to 1 there). Having rnn_size > 1 just sounds like having rnn_size number of all the gates (so, instead of 4 gates it will be 4*rnn_size). The question is how did you come up with that param? Did you have a chance to see the affect of different rnn_size in prediction quality? any link is helpful, thanks!

any link is helpful, thanks!

Reply • Share ›

Maxim Ivanov ➚ Maxim Ivanov • a year ago

oh, I see. LSTM.create(3,2) can be considered as layer with 2 LSTM units. Each of them is original LSTM unit with 4 gates. Makes sense?

Reply • Share ›

Adam Paszke Mod ➚ Maxim Ivanov • a year ago

Exactly! This would be the layer that accepts an input vector of length 3 and uses 2 LSTM units to generate output of length 2. Second parameter (let's call it x) determines how many units will be used (so there will be 4*x gates).

Regarding Your question about the prediction quality, I'd say it depends on a problem. Bigger sizes allow the layer to be more expressive and exhibit more complex behaviour, but can also lead to overfitting (just like with the regular Linear layers). In the 'Visualizing and Understanding Recurrent Networks' paper You can find a table that lists some benchmark results for LSTM networks with varying number of layers and sizes (on top of page 4).

Reply • Share ›

Peter Uherek • a year ago

Really good article. I am glad that someone explain the LSTM. It is hard to understand without good examples. I want to ask you, if you have plan to expand this article about training the LSTM?

Reply • Share ›

Adam Paszke Mod ➚ Peter Uherek • a year ago

I'll try to, if only I'll have some time :)

It's very similar to training regular NN modules. Do you know the API? You can read more in here and here (just the exmaple of manual training). I hope this helps at least a bit!

1 Reply • Share ›

Bibekananda Kundu • a year ago

How to use

layer:backward()

Reply • Share ›

Bibekananda Kundu • a year ago

Can you please provide the code for training the LSTM ?

Reply • Share ›

Andrew Morgan • a year ago

I really liked your article, and it's encouraged me to have a go building LSTMs too. I forked your examples and have been testing it, to try and do learning with it on some data I have. But, I think I'm misunderstand how training fits into your example.

My code is on github along with test data. https://github.com/bytesumo...

Maybe you could glance at it, offer a pointer to fixing the training process?
Many thanks, Andrew.

Reply • Share ›

Adam Paszke Mod → Andrew Morgan • a year ago

The link you posted gives me a 404 :/
Did you solve the issue? I can't see LSTM implementation in your code.

Reply • Share ›

Andrew Morgan → Adam Paszke • a year ago

yeah ... so my initial enthusiasm ebbed as I realised my attempt was rubbish! So ditched it and settled on trying some experiments with a working library rnn. My experimental code is here. https://github.com/bytesumo...
Still not sure it learns anything, but most bugs ironed out now.

Reply • Share ›

Viking Ender • a year ago

Hi, this model may better have a updateOutput() & backward() since it behaves different from Containers such as Sequential & Concat. While it's exactly what you need to define in training, but putting things together (combine all layers in a complete Module) permits you convenience to optimize parameters as a whole. Cheers.

Reply • Share ›

Adam Paszke Mod → Viking Ender • a year ago

Uh, you could combine the layers in some kind of container, but I haven't tried it yet. Anyway, it's a thing worth adding here! Thanks!

Reply • Share ›

Viking Ender → Adam Paszke • a year ago

I have an implementation here, the LSTM module mainly was copied from yours. So your code should work with mine as well.
https://github.com/KHN190/t...

Reply • Share ›

Adam Paszke Mod → Viking Ender • a year ago

I see. It might be convenient, but you code is too big to put it in here. And I'm not sure if it wouldn't be simpler to write another nngraph module wrapping individual layers, or your own function to construct and connect all the layers into a single network in advance.

Reply • Share ›

Viking Ender ➚ Adam Paszke • a year ago

I don't see a way to escape from crafting the parameters updating. In my case, it's implemented in the partner functions of boneRecurrent. In your case, it must be somewhere during the training. For reusability it's better to create a Container. For one-go experiment, you need to modify training code a bit from side to side. But on one thing you are right, nn.gModule forces a lot of inconvenience in coding (e.x. I can't do resizeAs in a module, simply because the input is no more a tensor! It's nngraph.Node then). I think it's a fault of Torch.

Reply • Share ›

Ayana Altaqin • a year 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 ...

Reply • Share ›

Ozan Çağlayan • 2 years ago

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

Thanks :)

Reply • Share ›

Adam Paszke Mod ➚ Ozan Çağlayan • 2 years ago

You've got me, thanks! ;)

Reply • Share ›

joe • 2 years 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 ^ ^

Reply • Share ›

Adam Paszke Mod ➚ joe • 2 years ago

apaszke

apaszke