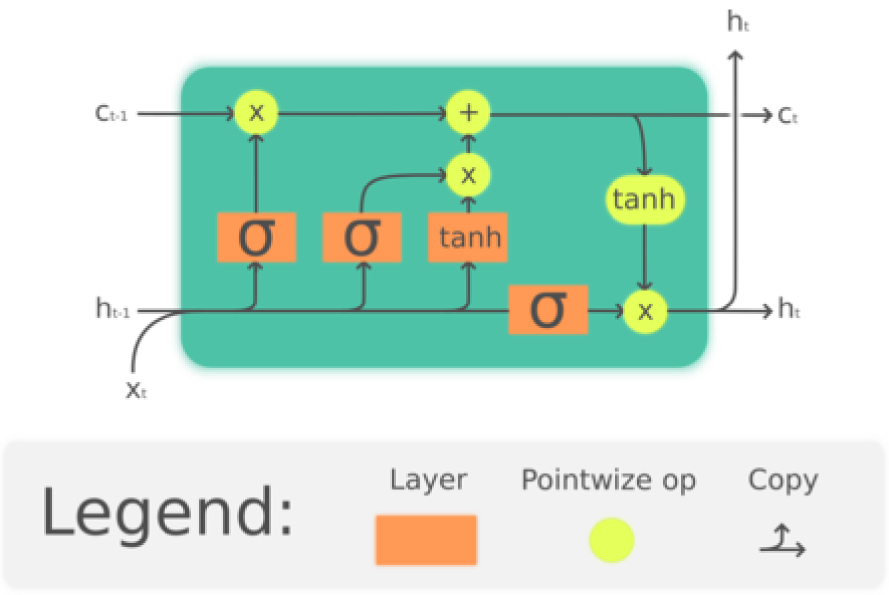
Building a LSTM by hand on PyTorch

Being able to build a LSTM cell from scratch enable you to make your own changes on the architecture and takes your studies to the next level.

Piero Esposito May 25, 2020 · 6 min read ★



LSTM cell illustration — Source : https://upload.wikimedia.org/wikipedia/commons/thumb/3/3b/The_LSTM_cell.png/300px-The_LSTM_cell.png — accessed in 2020–05–24

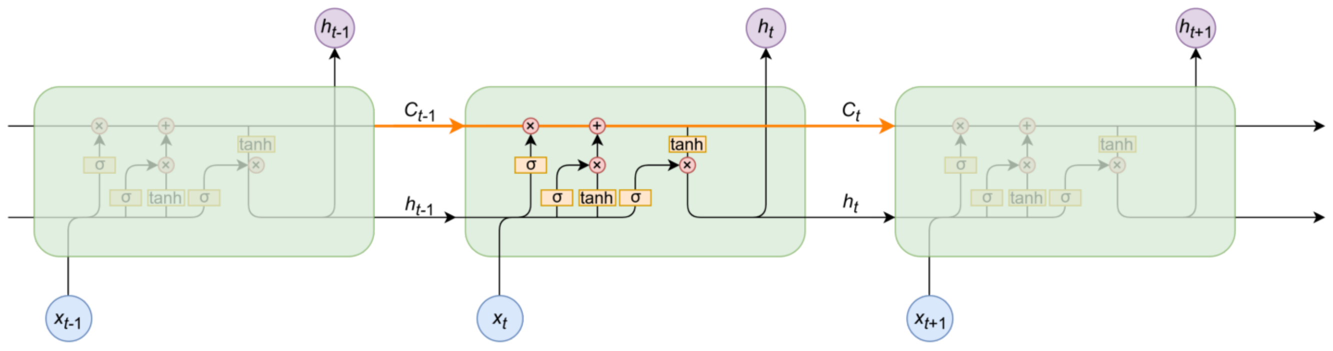
All the code mentioned are on the gists below or in our repo.

The LSTM cell is one of the most interesting architecture on the Recurrent Neural Networks study field on Deep Learning: Not only it enables the model to learn from long sequences, but it also creates a numerical abstraction for long and short term memories, being able o substitute one for another whenever needed.

Last but not least, we will show how to do minor tweaks on our implementation to implement some new ideas that do appear on the LSTM study-field, as the peephole connections.

The LSTM Architecture

The LSTM has we is called a gated structure: a combination of some mathematical operations that make the information flow or be retained from that point on the computational graph. Because of that, it is able to "decide" between its long and short-term memory and output reliable predictions on sequence data:

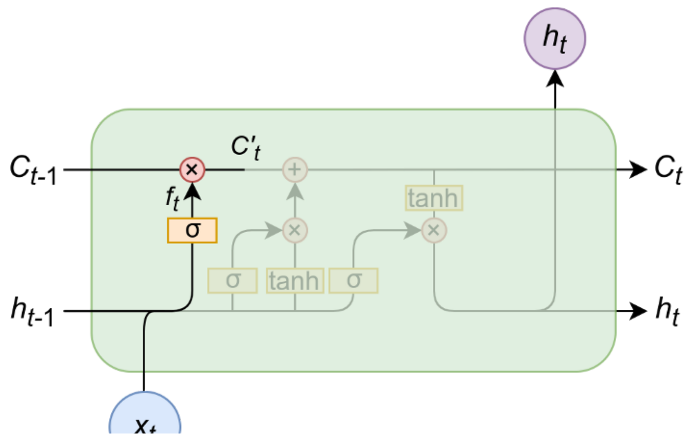


Sequence of predictions in a LSTM cell. Notice that it not only flow the predictions h_t, but also a c_t, which is the representant of the long-term memory. Source: https://medium.com/turing-talks/turing-talks-27-modelos-de-predi%C3%A7%C3%A3o-lstm-df85d87ad210. Accessed in 2020–05–24

We will go from it part-by-part:

The forget gate

The forget gate is the one where the input information is operated along with the candidate, as the long term memory. See that, on the first linear combination of the input, hidden state and bias, it is applied a sigmoid function:



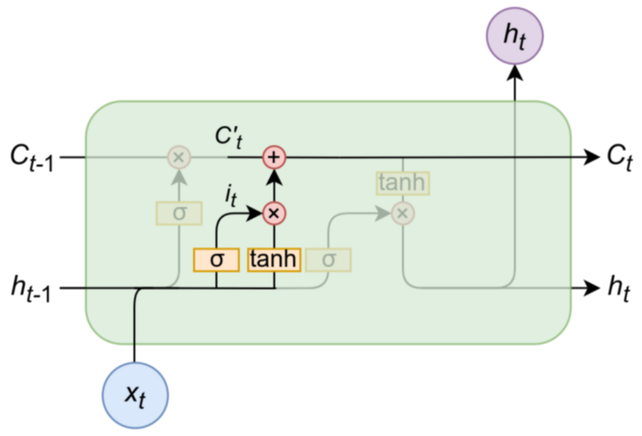
$$f_t = \sigma(U_f x_t + V_f h_{t-1} + b_f)$$
$$C'_t = f_t \cdot C_{t-1}$$

predi%C3%A7%C3%A3o-lstm-df85d87ad210. Accessed in 2020–05–24

That sigmoid “scales” the output of the forget gate from 0 to 1 — and, by multiplying it with the candidate, we can either set it to zero, which represent a “forgetting” from the long time memory, or by a bigger number, which represent “how much” we are remembering from that long-term memory.

The input gate and solution of the new long-term memory

The input gate is where the information contained on the input and hidden state is combined and then operated along with the candidate and partial candidate c’_t:



$$
\begin{aligned}
i_t &= \sigma(U_i\, x_t + V_i\, h_{t-1} + b_i) \\
C_t^+ &= \tanh(U_c\, x_t + V_c\, h_{t-1} + b_c) \\
C_t &= C_t' + i_t \cdot C_t^+
\end{aligned}
$$

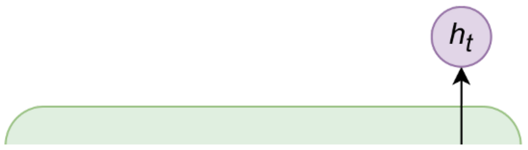
Input gate of LSTM cell. Source: https://medium.com/turing-talks/turing-talks-27-modelos-de-predi%C3%A7%C3%A3o-lstm-df85d87ad210. Accessed in 2020–05–24

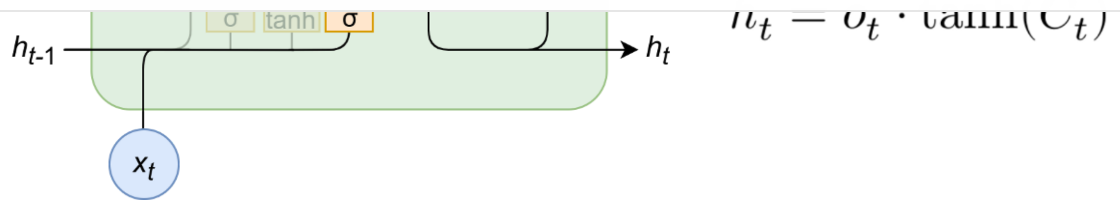
On those operations, it is decided how much of the new information will be introduced on the memory how it will change — that’s why we use a tanh function (“scale” from -1 to 1). We combine the partial candidate from the short-term and long-term memories and set it as the candidate.

And now we can proceed to the output gate.

The output gate and hidden state (output) of the cell

After that, we can gather o_t as the output gate of the LSTM cell and then multiply it per the tanh of the candidate (long-term memory) which was already update with the proper operation. The output of network will be h_t.





Output gate of LSTM cell. Source: https://medium.com/turing-talks/turing-talks-27-modelos-de-predi%C3%A7%C3%A3o-lstm-df85d87ad210. Accessed in 2020–05–24

At the end, we have:

Equations of the LSTM cell:

$$f_t = \sigma(U_f\ x_t + V_f\ h_{t-1} + b_f)$$

$$i_t = \sigma(U_i\ x_t + V_i\ h_{t-1} + b_i)$$

$$o_t = \sigma(U_o\ x_t + V_o\ h_{t-1} + b_o)$$

$$g_t = \tanh\ (U_g\ x_t + V_g\ h_{t-1} + b_g) \text{ a.k.a. } \tilde{c}_t$$

$$c_t = f_t \circ c_{t-1} + i_t \circ g_t$$

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

Implementing it on PyTorch

To implement it on PyTorch, we will first do the proper imports.

```python
import math
import torch
import torch.nn as nn
```

lstm-post-imports.py hosted with ❤ by GitHub view raw

We will now create its class by inheriting from `nn.Module` , and then also instance its parameters and weight initialization, which you will see below (notice that its shapes are decided by the input size and output size of the network):

Setting the parameters

```python
class NaiveCustomLSTM(nn.Module):
        self.input_size = input_sz
        self.hidden_size = hidden_sz

        #i_t
        self.U_i = nn.Parameter(torch.Tensor(input_sz, hidden_sz))
        self.V_i = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz))
        self.b_i = nn.Parameter(torch.Tensor(hidden_sz))

        #f_t
        self.U_f = nn.Parameter(torch.Tensor(input_sz, hidden_sz))
        self.V_f = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz))
        self.b_f = nn.Parameter(torch.Tensor(hidden_sz))

        #c_t
        self.U_c = nn.Parameter(torch.Tensor(input_sz, hidden_sz))
        self.V_c = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz))
        self.b_c = nn.Parameter(torch.Tensor(hidden_sz))

        #o_t
        self.U_o = nn.Parameter(torch.Tensor(input_sz, hidden_sz))
        self.V_o = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz))
        self.b_o = nn.Parameter(torch.Tensor(hidden_sz))

        self.init_weights()
```

custom-lstm-init.py hosted with ❤ by GitHub view raw

To understand the shape of each of the weights, let's see:

The input shape of the matrix is (batch_size, sequence_length, feature_length) — and so the weight matrix that will multiply each element of the sequence must have the shape (feature_length, output_length).

The hidden state (a.k.a. output), for each of element on the sequence has the shape (batch_size, output_size), which results, at the end of the sequence processing, an output shape of (batch_size, sequence_length, output_size). — Because of that, the weight_matrix that will multiply it must have the shape (output_size, output_size) which corresponds to the parameter hidden_sz of our cell.

And here is the weight initialization, which we use as the same as the one in PyTorch default `nn.Module` s:

```
            weight.data.uniform_(-stdv, stdv)
```

lstm-weight-init.py hosted with ❤ by GitHub view raw

Feedforward operation

The feedforward operation receives the `init_states` parameter, which is a tuple with the (h_t, c_t) parameters of the equations above, which is set to zero if not introduced. We then perform the feedforward of the LSTM equations for each of the sequence elements preserving the (h_t, c_t), and introducing it as the states for the next element of the sequence.

At the end, we return the predictions and the last states tuple. Let's see how it happens:

```
def forward(self,
                x,
                init_states=None):

        """
        assumes x.shape represents (batch_size, sequence_size, input_size)
        """
        bs, seq_sz, _ = x.size()
        hidden_seq = []

        if init_states is None:
            h_t, c_t = (
                torch.zeros(bs, self.hidden_size).to(x.device),
                torch.zeros(bs, self.hidden_size).to(x.device),
            )
        else:
            h_t, c_t = init_states

        for t in range(seq_sz):
            x_t = x[:, t, :]

            i_t = torch.sigmoid(x_t @ self.U_i + h_t @ self.V_i + self.b_i)
            f_t = torch.sigmoid(x_t @ self.U_f + h_t @ self.V_f + self.b_f)
            g_t = torch.tanh(x_t @ self.U_c + h_t @ self.V_c + self.b_c)
            o_t = torch.sigmoid(x_t @ self.U_o + h_t @ self.V_o + self.b_o)
            c_t = f_t * c_t + i_t * g_t
            h_t = o_t * torch.tanh(c_t)

            hidden_seq.append(h_t.unsqueeze(0))
        hidden_seq = hidden_seq.transpose(0, 1).contiguous()
        return hidden_seq, (h_t, c_t)
```

ff-lstm.py hosted with ❤ by GitHub view raw

Now and optimized version

This LSTM is correct in terms of operations but not very optimized in terms of computational time: we perform 8 matrix multiplications separately, which is much slower than doing it in a vectorized way. We will now show how it could be done by reducing it to 2 matrix multiplications, which would make it much faster.

To do this operations, we set two matrixes, U and V, which have the weight contained on the 4 matrix multiplications each of they do. We then perform the gated operations on the matrixes that already passed per the linear combinations + bias operation.

With the vectorized operations, the equations of the LSTM cell would be:

$$A_t = Ux_t + Vh_t + b \text{ a.k.a. "Concatenated gates matrix" - runs vectorized}$$

$$i_t = \sigma(A_t[:, 0 : hs])$$

$$f_t = \sigma(A_t[:, hs : 2 \times hs])$$

$$o_t = \sigma(A_t[:, 2 \times hs : 3 \times hs])$$

$$g_t = \tanh(A_t[:, 3 \times hs : 4 \times hs])$$

$$c_t = f_t \circ c_{t-1} + i_t \circ g_t$$

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

And so it’s `nn.Module` class would be:

Optimized LSTM cell class

```
class CustomLSTM(nn.Module):
    def __init__(self, input_sz, hidden_sz):
        super().__init__()
        self.input_sz = input_sz
        self.hidden_size = hidden_sz
        self.init_weights()

    def init_weights(self):
        stdv = 1.0 / math.sqrt(self.hidden_size)
        for weight in self.parameters():
            weight.data.uniform_(-stdv, stdv)

    def forward(self, x,
                init_states=None):
        """Assumes x is of shape (batch, sequence, feature)"""
        bs, seq_sz, _ = x.size()
        hidden_seq = []
        if init_states is None:
            h_t, c_t = (torch.zeros(bs, self.hidden_size).to(x.device),
                        torch.zeros(bs, self.hidden_size).to(x.device))
        else:
            h_t, c_t = init_states

        HS = self.hidden_size
        for t in range(seq_sz):
            x_t = x[:, t, :]
            # batch the computations into a single matrix multiplication
            gates = x_t @ self.W + h_t @ self.U + self.bias
            i_t, f_t, g_t, o_t = (
                torch.sigmoid(gates[:, :HS]), # input
                torch.sigmoid(gates[:, HS:HS*2]), # forget
                torch.tanh(gates[:, HS*2:HS*3]),
                torch.sigmoid(gates[:, HS*3:]), # output
            )
            c_t = f_t * c_t + i_t * g_t
            h_t = o_t * torch.tanh(c_t)
            hidden_seq.append(h_t.unsqueeze(0))
        hidden_seq = torch.cat(hidden_seq, dim=0)
        # reshape from shape (sequence, batch, feature) to (batch, sequence, feature)
        hidden_seq = hidden_seq.transpose(0, 1).contiguous()
        return hidden_seq, (h_t, c_t)
```

optimized-lstm.py hosted with ❤ by GitHub view raw

Last but not least, we can show how easy would it be to tweak your implementation to use LSTM peephole connections.

LSTM peephole

$A_t = Ux_t + Vh_t + b$ a.k.a. "Concatenated gates matrix" - runs vectorized

$$i_t = \sigma(A_t[:, 0 : hs])$$

$$f_t = \sigma(A_t[:, hs : 2 \times hs])$$

$$o_t = \sigma(A_t[:, 2 \times hs : 3 \times hs])$$

$$c_t = f_t \circ c_{t-1} + i_t\ \sigma(U[:, 3 \times hs : 4 \times h_s]\ x_t)$$

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

It occurs that, by having a well implemented and optimized implementation of LSTM, we can add the options for peephole connection with some minor tweak on it:

And with that our LSTM is done. You may want to see it on our repo and test it with our LSTM text-sentiment analysis notebook which we made ready for testing and comparing with torch LSTM built-in layer.

Conclusion

We can take as a conclusion that, despite being kind of a Deep Learning tabus, if done in steps and with clean and good coding, it is actually easy to perform it's operations into a clean, easy to use `nn.Module` . We've also seen how easy would it be to change and tweak its connections to do operations like the peephole connection.

References

piEsposito/pytorch-lstm-by-hand

A small and simple tutorial on how to craft a LSTM nn.Module by hand on PyTorch. Remember to execute bash...

github.com

Understanding LSTM Networks

Posted on August 27, 2015 Humans don't start their thinking from scratch every second. As you read this essay, you...

colah.github.io

Redes Neurais | LSTM

Como lidar com perda de memória com Machine Learning

medium.com