LSTM Structure

May 31, 2020

1 LSTM Structure and Hidden State

We know that RNNs are used to maintain a kind of memory by linking the output of one node to the input of the next. In the case of an LSTM, for each piece of data in a sequence (say, for a word in a given sentence), there is a corresponding *hidden state* h_t. This hidden state is a function of the pieces of data that an LSTM has seen over time; it contains some weights and, represents both the short term and long term memory components for the data that the LSTM has already seen.

So, for an LSTM that is looking at words in a sentence, **the hidden state of the LSTM will change based on each new word it sees. And, we can use the hidden state to predict the next word in a sequence** or help identify the type of word in a language model, and lots of other things!

1.0.1 Exercise Repository

Note that most exercise notebooks can be run locally on your computer, by following the directions in the Github Exercise Repository.

1.1 LSTMs in Pytorch

To create and train an LSTM, you have to know how to structure the inputs, and hidden state of an LSTM. In PyTorch an LSTM can be defined as: `lstm = nn.LSTM(input_size=input_dim, hidden_size=hidden_dim, num_layers=n_layers)`.

In PyTorch, an LSTM expects all of its inputs to be 3D tensors, with dimensions defined as follows: >* `input_dim` = the number of inputs (a dimension of 20 could represent 20 inputs) >* `hidden_dim` = the size of the hidden state; this will be the number of outputs that each LSTM cell produces at each time step. >* `n_layers` = the number of hidden LSTM layers to use; this is typically a value between 1 and 3; a value of 1 means that each LSTM cell has one hidden state. This has a default value of 1.

1.1.1 Hidden State

Once an LSTM has been defined with input and hidden dimensions, we can call it and retrieve the output and hidden state at every time step. `out, hidden = lstm(input.view(1, 1, -1), (h0, c0))`

The inputs to an LSTM are `(input, (h0, c0))`. >* `input` = a Tensor containing the values in an input sequence; this has values: (seq_len, batch, input_size) >* `h0` = a Tensor containing the initial hidden state for each element in a batch >* `c0` = a Tensor containing the initial cell memory for each element in the batch

`h0` nd `c0` will default to 0, if they are not specified. Their dimensions are: (n_layers, batch, hidden_dim).

These will become clearer in the example in this notebook. This and the following notebook are modified versions of this PyTorch LSTM tutorial.

Let's take a simple example and say we want to process a single sentence through an LSTM. If we want to run the sequence model over one sentence "Giraffes in a field", our input should look like this `1x4` row vector of individual words:

$$\begin{bmatrix}\text{Giraffes in a field}\end{bmatrix} \tag{1}$$

In this case, we know that we have **4 inputs words** and we decide how many outputs to generate at each time step, say we want each LSTM cell to generate **3 hidden state values**. We'll keep the number of layers in our LSTM at the default size of 1.

The hidden state and cell memory will have dimensions (n_layers, batch, hidden_dim), and in this case that will be (1, 1, 3) for a 1 layer model with one batch/sequence of words to process (this one sentence) and 3 genereated, hidden state values.

1.1.2 Example Code

Next, let's see an example of one LSTM that is designed to look at a sequence of 4 values (numerical values since those are easiest to create and track) and generate 3 values as output. This is what the sentence processing network from above will look like, and you are encouraged to change these input/hidden-state sizes to see the effect on the structure of the LSTM!

```
In [1]: import torch
        import torch.nn as nn
        import matplotlib.pyplot as plt

        %matplotlib inline

        torch.manual_seed(2) # so that random variables will be consistent and repeatable for te
```

```
Out[1]: <torch._C.Generator at 0x7f215429df10>
```

1.1.3 Define a simple LSTM

A note on hidden and output dimensions

The `hidden_dim` and size of the output will be the same unless you define your own LSTM and change the number of outputs by adding a linear layer at the end of the network, ex. fc = nn.Linear(hidden_dim, output_dim).

```
In [2]: from torch.autograd import Variable

        # define an LSTM with an input dim of 4 and hidden dim of 3
        # this expects to see 4 values as input and generates 3 values as output
        input_dim = 4
        hidden_dim = 3
        lstm = nn.LSTM(input_size=input_dim, hidden_size=hidden_dim)
```

```
        # make 5 input sequences of 4 random values each
        inputs_list = [torch.randn(1, input_dim) for _ in range(5)]
        print('inputs: \n', inputs_list)
        print('\n')

        # initialize the hidden state
        # (1 layer, 1 batch_size, 3 outputs)
        # first tensor is the hidden state, h0
        # second tensor initializes the cell memory, c0
        h0 = torch.randn(1, 1, hidden_dim)
        c0 = torch.randn(1, 1, hidden_dim)


        h0 = Variable(h0)
        c0 = Variable(c0)
        # step through the sequence one element at a time.
        for i in inputs_list:
            # wrap in Variable
            i = Variable(i)

            # after each step, hidden contains the hidden state
            out, hidden = lstm(i.view(1, 1, -1), (h0, c0))
            print('out: \n', out)
            print('hidden: \n', hidden)
```

```
inputs:
 [tensor([[ 1.4934,  0.4987,  0.2319,  1.1746]]), tensor([[-1.3967,  0.8998,  1.0956, -0.5231]])


out:
 tensor([[[-0.4372,  0.2583,  0.2947]]])
hidden:
 (tensor([[[-0.4372,  0.2583,  0.2947]]]), tensor([[[-0.7344,  0.6209,  0.4191]]]))
out:
 tensor([[[-0.2836,  0.1314,  0.4133]]])
hidden:
 (tensor([[[-0.2836,  0.1314,  0.4133]]]), tensor([[[-0.5041,  0.2672,  0.6370]]]))
out:
 tensor([[[-0.3404,  0.4880,  0.1949]]])
hidden:
 (tensor([[[-0.3404,  0.4880,  0.1949]]]), tensor([[[-0.5552,  0.7909,  0.3300]]]))
out:
 tensor([[[-0.3544,  0.2405,  0.3150]]])
hidden:
 (tensor([[[-0.3544,  0.2405,  0.3150]]]), tensor([[[-0.5645,  1.0073,  0.6101]]]))
out:
 tensor([[[-0.3328,  0.0437,  0.3817]]])
```

```
hidden:
 (tensor([[[-0.3328,  0.0437,  0.3817]]]), tensor([[[-0.5311,  0.1181,  0.5304]]]))
```

You should see that the output and hidden Tensors are always of length 3, which we specified when we defined the LSTM with `hidden_dim`.

1.1.4 All at once

A for loop is not very efficient for large sequences of data, so we can also, **process all of these inputs at once.**

1. concatenate all our input sequences into one big tensor, with a defined batch_size
2. define the shape of our hidden state
3. get the outputs and the *most recent* hidden state (created after the last word in the sequence has been seen)

The outputs may look slightly different due to our differently initialized hidden state.

```
In [3]: # turn inputs into a tensor with 5 rows of data
        # add the extra 2nd dimension (1) for batch_size
        inputs = torch.cat(inputs_list).view(len(inputs_list), 1, -1)

        # print out our inputs and their shape
        # you should see (number of sequences, batch size, input_dim)
        print('inputs size: \n', inputs.size())
        print('\n')

        print('inputs: \n', inputs)
        print('\n')

        # initialize the hidden state
        h0 = torch.randn(1, 1, hidden_dim)
        c0 = torch.randn(1, 1, hidden_dim)

        # wrap everything in Variable
        inputs = Variable(inputs)
        h0 = Variable(h0)
        c0 = Variable(c0)
        # get the outputs and hidden state
        out, hidden = lstm(inputs, (h0, c0))

        print('out: \n', out)
        print('hidden: \n', hidden)
```

```
inputs size:
 torch.Size([5, 1, 4])
```

```
inputs:
 tensor([[[ 1.4934,  0.4987,  0.2319,  1.1746]],

        [[-1.3967,  0.8998,  1.0956, -0.5231]],

        [[-0.8462, -0.9946,  0.6311,  0.5327]],

        [[-0.8454,  0.9406, -2.1224,  0.0233]],

        [[ 0.4836,  1.2895,  0.8957, -0.2465]]])


out:
 tensor([[[ 0.1611,  0.2200,  0.2213]],

        [[ 0.0364, -0.0390,  0.2638]],

        [[-0.1425, -0.0174,  0.1504]],

        [[-0.1583,  0.1264,  0.1709]],

        [[-0.2007, -0.1559,  0.2489]]])
hidden:
 (tensor([[[-0.2007, -0.1559,  0.2489]]]), tensor([[[-0.4429, -0.2975,  0.3252]]]))
```

1.1.5 Next: Hidden State and Gates

This notebooks shows you the structure of the input and output of an LSTM in PyTorch. Next, you'll learn more about how exactly an LSTM represents long-term and short-term memory in it's hidden state, and you'll reach the next notebook exercise.

Part of Speech In the notebook that comes later in this lesson, you'll see how to define a model to tag parts of speech (nouns, verbs, determinants), include an LSTM and a Linear layer to define a desired output size, *and* finally train our model to create a distribution of class scores that associates each input word with a part of speech.

In [ ]: