Long Short-Term Memory (LSTM) NNs for Natural Language Processing (NLP)

The first point to be aware of, in using NNs for NLP, is that the inputs to an NN are usually assumed to consist of "embedding vectors" (typically with 100 or so components) representing words, rather than the explicit words themselves. These embedding vectors are learned from large text corpora, in such a way that words that tend to occur in the vicinity of the same words will have similar vector embeddings; e.g., "dog" and "cat" will have similar embeddings because they tend to co-occur with the same words. Vector similarity is measured by their dot product (proportional to the cosine of the angle between them), transformed via the sigmoid function into a probability. A widely used method of obtaining embeddings is Word2Vec (with a "skip-gram" algorithm at its core); this is discussed in detail in the 3rd edition draft of Jurafsky & Martin's book, esp. sections 6.8-6.10 (see https://web.stanford.edu/~jurafsky/slp3/ed3book.pdf).

So how do we handle these vector inputs? (As reference, see Sec. 7.3 in Jurafsky & Martin.) The answer is that we simply concatenate the vectors, so that each component of each vector serves as a separate input at the NN input layer. Now consider the next layer, which is a hidden layer (i.e., not the output layer) in deep NNs. This hidden layer can have any number of units, where these units receive inputs from some or all of the units in the input layer. So we want to form a linear combination (dot product with a weight vector) of these inputs for each hidden unit, and then pass the result through a soft threshold function. We can think of the weight vectors as the rows of a matrix, and so we can do all the required dot products in one fell swoop by matrix multiplication. We then apply the same soft threshold (say, a sigmoid or relu — rectified linear unit) component-by-component to the vector resulting from the matrix-vector product. At the output layer, we usually desire a probability distribution over a set of alternatives (e.g., a POS for the middle word in the input). To achieve this, we apply a softmax transformation instead of a sigmoid function to the vector $(z_1, z_2, \ldots, z_d)$ resulting from the final matrix multiplication, i.e.,

$e^{z_i}/[e^{z_1}+\ldots+e^{z_d}]$.

So for example, this might represent the probability of the i^{th} possible POS ($i = 1, 2, \ldots, d$) of the middle input word.

Now, NLP typically involves processing arbitrarily long texts (or speech input). How do we use a NN of fixed size to do that? We can think of it as passing a "sliding window" of fixed size over the text – or more realistically, as pushing words of a text into an input array (*buffer*) at one end (by convention, the right end), and dropping them out at the other. So the NN "sees" successive input segments shifted by one word at each step. But a problem with this, in a strict feed-forward network, is that at each step the network completely forgets what it has already seen — each newly shifted buffer state is treated as unrelated to the previous state. This is at odds with the fact that dependencies in NLP can be rather long-distance. For example, in the question, *"Which chapter did you say I should plan to read tomorrow?"*, the words *"read tomorrow"* might be misclassified as VB NN (instead of VB RB), because it is the initial question phrase *"Which chapter"* that justifies the omission of an object following *"read"*. Consequently, we use *recurrent* NNs, i.e., ones containing cyclic connections. The simplest example is the use of units with a self-loop, i.e., feeding their output back into themselves as one of their inputs, as a way of "reminding themselves" of what they saw a moment earlier.

To understand how such cells can function as memory cells, we need to discuss how the signal propagation in an NN works, when the NN sees a shifted input at each step. We think of these steps as occurring at times t = 0, 1, 2, 3, etc. After each shift, the NN does its calculation (from the input layer all the way to the "top", the output layer). Now, this means that the calculation of

outputs of the units, if there's more that one layer, takes place in *less than* one time step. But then, wouldn't a recurrent unit feed into itself multiple times in a single time step? This puzzle comes from thinking about neural nets too realistically, in terms of signals transmitted very rapidly from neuron to neuron – much faster than it takes to read a word. The way to think about units in a neural net is to assume that they *maintain* their output value until the next time step. The outputs of all the units in the entire net are calculated "as if instantaneously" (but the calculation has to follow the graph structure), and the values thus obtained remain available till the next time step – and aren't recalculated till the next time step. Thus a recurrent unit doesn't get to "see" its own output till the next time step, when the input has been shifted. This process is often illustrated by "unrolling" the RNN in time, showing successive time steps – it's as if we created a new copy of the network where the recurrent connections carry outputs from the previous time step to the target unit (or subnetwork) in the new copy.

So let's see how we get from NNs with simple recurrent units to LSTMs (Long Short-Term Memory networks). The first thing to realize when looking at a schematic diagram of an LSTM is that all the nodes shown are vector-to-vector operations, and so the directed edges "carry" vectors forward. Don't think of the nodes as basic NN units that get a vector of n inputs and generate a single output (by forming a weighted sum & thresholding it). But note that if we have two successive NN layers, each with n units, where each unit of the first layer sends its output to all units of the second layer, and the second-layer units all use the same thresholding function, then the second layer is equivalent to applying an n by n matrix to the n-vector of first-layer output values, yielding another n-vector, and then applying the thresholding function component-by-component. So it's quite reasonable to stipulate matrix-vector multiplications, and component-wise operations on vectors in an NN – and that's exactly what we see in an LSTM diagram & its mathematical specification. We can think of the nodes as really representing layers of NN units.

The intuition behind the LSTM proposal (Hochreiter & Schmidhuber 1997, https://www.bioinf.jku.at/publications/older/2604.pdf) was probably this: A memory cell whose output is the weighted sum of (a) its current input vector from the preceding node and (b) its own output vector from the previous time step (i.e., fed back into itself), without thresholding, is potentially a memory of arbitrary duration. But if the weights of the (a)-inputs is large compared to the weight of the self-input (b), that self-input is apt to become quickly suppressed by subsequent (a)-inputs – so the "memory imprint" of an input from several time steps earlier will be very faint. As a result the derivatives of weights that contributed to (a)-inputs several time steps back become too small to be computed – the derivatives "vanish". On the other hand, if the weights of (a)-inputs are kept small compared to the weight of the self-input, the memory cell will have excellent, persistent "recall". However, it will barely take account of new inputs. (Apparently, this leads to wildly "oscillating" derivatives.)

So, one key idea behind LSTMs is to allow an ***input gate*** to control how heavily those (a)-inputs are weighted relative to the memory cell's self-input. The gate receives the same inputs as the node that supplies the memory cell's (a)-inputs, and it decides how strongly those (a)-inputs should affect the memory cell. It's as if the gate could say to the memory cell, "Hey, this (a)-input vector is important, remember it!" or conversely, "This is only of passing interest, don't pay it much attention". Technically, the way the input gate exerts its influence on what the memory cell sees is by applying its output component-by-component to the (a)-vector headed towards the memory cell, i.e., forming a Hadamard product. Thus if the gate keeper multiplies an input component by 0, it just won't get through. (If all gate-keeper components are 0, the

memory unit is totally protected from memory disruption.)

But now we may have the opposite problem that a memory unit that remembers an input indefinitely long may be disrupting its successors by sending information to them which is irrelevant at most or all later times. So before passing the memory cell output to the next layer, we apply another multiplicative gate, an ***output gate*** that effectively determines at any time what gets passed on to the next layer.

The LSTM memory unit design was subsequently improved by Gers, Schmidhuber & Cummins, 2000, by inserting a ***forget gate*** into the memory unit's self-loop, thus controlling what vector components and how much of them the cell remembers from moment to moment. Later, the idea was introduced that the current state of the memory might usefully contribute to the decisions of the gatekeeper units; in other words, the memory unit should "have a say" in what the 3 gates (input gate, output gate, and forget gate) do to its inputs, outputs, and memory. Thus, connections were added that allow the memory unit to send its output to the three gate-keepers (as well as to the next NN layer, via the output gate). The result is a "peephole memory cell" and thus a "peephole LSTM", as illustrated in the Wikipedia article on LSTMs at

https://en.wikipedia.org/wiki/Long_short-term_memory

(also reproduced at

http://www.cs.rochester.edu/u/schubert/530/in-order-parsing-excerpts/peephole-lstm-diagram.png).

Still later, the memory cells were made convolutional by replacing each matrix operation on the inputs (from the preceding, lower layer) with the convolution of a matrix with the input (again, see the Wikipedia article). Convolutional layers were inspired by the receptive fields (small regions of the retina, over which some regularity like a dot or edge is detected), observed in mammalian vision. A convolutional layer is special in that each of its units takes inputs from only a small segment (a few adjacent units) of the previous layer, where these segments slightly overlap; and further, the weights applied to these inputs are the same for all units in the convolutional layer. This greatly speeds up learning, since far fewer weights need to be learned than in a layer where each unit receives numerous inputs and applies its own unique weight vector to them.

Transition-Based Constituency Parsing Using LSTMs

The algorithm we assume for transition-based phrase-structure parsing is based on the article

Liu & Zhang, "In-order transition-based constituent parsing", TACL 2017, http://aclweb.org/anthology/Q17-1029;

see some relevant items in the *Supplementary Course Materials* section of the course web page. The essential algorithmic ideas are very much like those in bottom-up shift-reduce parsing, along with the chart-parsing idea of generating an active arc (or "dotted rule") whenever a constituent matching the initial right-hand element of a phrase-structure rule has been found. This is called a *project X* action. Instead of representing the dotted rule in a form like $X \rightarrow S_1 \circ S_2\ S_3$, the parser represents it on the stack as just $S_1\ X$, "waiting" for the remaining constituents $S_2\ S_3$. When $S_2\ S_3$ have appeared on the stack to the right of this (rightmost = top of stack), a reduce action replaces $S_1\ X\ S_2\ S_3$ by X.

We now want to look at the role of LSTMs in this processing. Basically we create an "oracle" that chooses the right *shift, reduce, project,* or *finish* action at each step of the parser's operation, using LSTMs that "look at" the buffer, the stack, and the recent parser actions.

Liu & Zhang employ a bidirectional "stack-LSTM" taking the buffer contents at a given moment as input, another bidirectional stack-LSTM taking the stack as input, and a one-directional "vanilla" (ordinary) LSTM taking some portion of the action sequence so far carried out by the parser as input. The outputs of the LSTMs are fed into a softmax layer, which thus outputs a probability distribution over what the next action of the parser should be. In greater detail, the actions are *shift* the next word from the buffer to the stack; *project X* (for various possible nonterminal categories *X,* whose initial constituent is rightmost on the stack); reduce a sequence of stack elements $s_j\ X\ s_{j-1} \ldots s_0$ to a subtree *X* with children $s_j, s_{j-1}, \ldots, s_0$; or *finish* (just setting a boolean to true). To the extent that one action choice receives much higher probability than the others, and in fact corresponds to the correct parse, the NN serves as an effective oracle for the parser.

Let's try to figure out the purpose and operation of a *stack-LSTM,* borrowed from Dyer et al., 2016 (https://www.aclweb.org/anthology/P15-1033). The problem with applying a vanilla LSTM to the parser stack is that *reduce* actions make the reduced material (the child nodes) invisible to the LSTM by popping that material — but it may well be that this reduced material should influence the next choices of parser actions. For example, a sentence starting with *"it"* allows for certain continuations that are unlikely for other sentence subjects (compare *"it is possible that he's sick"* with the faulty **"Something is possible that he's sick"*). But if we've reduced *"it"* to NP, we won't be able to distinguish these cases. So the idea of a stack LSTM is to retain the popped material, and still add pushed elements on the right, but somehow identifying popped items, so that we can skip over them in moving a *Top* (of stack) marker backward to simulate pop actions. Now, in an algorithm using such a stack, we would have to individually mark each popped item as such — they needn't form a contiguous sequence. For example, suppose we repeatedly (a) push two items, and (b) pop one item; then you can verify that we'll get a sequence of alternating popped and non-popped items. Given this possibility, Dyer et al.'s explanation seems a bit unclear. They say,

> "Like a conventional LSTM, new inputs are always added in the right-most position, but in stack LSTMs, the current location of the stack pointer determines which cell in the LSTM provides c_{t-1} [the output of the central memory cell in an LSTM configuration] and h_{t-1} [the output after gating] when computing the new memory cell contents… In addition to adding elements to the end of the sequence, the stack LSTM provides a pop operation which moves the stack pointer to the previous element (i.e., the previous element that was extended, not necessarily the right-most element)."

Note that the stack-LSTM sees the stack as its input, and somehow performs those push and pop operations. What seems unclear is how the stack-LSTM "knows" where the previously added element is in the stack (or "stack summary", as they term the stack representation with all popped elements still visible) — as just noted, we can have alternating popped and unpopped elements in the stack (summary). Perhaps the point is for the stack-LSTM to *learn* how to move the *Top* marker?

Putting that aside, we should also note what is meant by a bidirectional LSTM. As the name suggests, it is really two LSTMs, one of which processes the sequence portion it is looking at from left to right, while the other processes it from right to left. The two outputs are then concatenated to produce the output of the bidirectional LSTM. Why does using such bi-LSTMs for the stack and the buffer provide better information about what action should be done next by the parser? This seems to call for further explanation; and why isn't it also advantageous to process the most recent set of actions forward and backward as well?

Anyway, you can see that once the parser has learned a good oracle, providing probabilities for possible next actions, we can not only produce a single likely parse, but also other plausible possibilities. A single likely parse might be produced by always picking the most probable action at each time step. More often multiple likely parses are produced, and then a *re-ranker* is applied to try to put the parses in an order where the correct parse is likely to be among the first 2 or 3. The re-ranker is separately trained on the same "gold" data that provided the original training set.

-=-=-=-=-=-=-=-=-=-=-=-=-

Your second assignment will be to code an "oracle" for training an in-order constituent parser. You won't need to actually train a parser, but just provide the oracle: This is given a phrase structure tree as input (in Treebank style, with words at the leaves), and outputs the sequence of actions that assign the correct parse tree to the words of the sentence. It should already be fairly clear what to do; you'll receive some more comments about the assignment.