Deep learning

13.1. Attention for Memory and Sequence Translation

François Fleuret
https://fleuret.org/ee559/
Nov 2, 2020
In all the operations we have seen such as fully connected layers, convolutions, or poolings, the contribution of a value in the input tensor to a value in the output tensor is entirely driven by their [relative] locations [in the tensor].

Attention mechanisms aggregate features with an importance score that

- depends on the feature themselves, not on their positions in the tensor,
- relax locality constraints.
Attention mechanisms modulate dynamically the weighting of different parts of a signal and allow the representation and allocation of information channels to be dependent on the activations themselves.

While they were developed to equip deep-learning models with memory-like modules (Graves et al., 2014), their main use now is to provide long-term dependency for sequence-to-sequence translation (Vaswani et al., 2017).
Neural Turing Machine
Graves et al. (2014) proposed to equip a deep model with an explicit memory to allow for long-term storage and retrieval.
The said module has an hidden internal state that takes the form of a tensor

\[ M_t \in \mathbb{R}^{N \times M} \]

where \( t \) is the time step, \( N \) is the number of entries in the memory and \( M \) is their dimension.

A “controller” is implemented as a standard feed-forward or recurrent model and at every iteration \( t \) it computes activations that modulate the reading / writing operations.
More formally, the memory module implements

- **Reading**, where given attention weights \( w_t \in \mathbb{R}_+^N, \sum_n w_t(n) = 1 \), it gets
  \[
  r_t = \sum_{n=1}^N w_t(n) M_t(n).
  \]

- **Writing**, where given attention weights \( w_t \), an *erase vector* \( e_t \in [0, 1]^M \) and an *add vector* \( a_t \in \mathbb{R}^M \) the memory is updated with
  \[
  \forall n, M_t(n) = M_{t-1}(n)(1 - w_t(n)e_t) + w_t(n)a_t.
  \]

The controller has multiple “heads”, and computes at each \( t \), for each writing head \( w_t, e_t, a_t \), and for each reading head \( w_t \), and gets back a read value \( r_t \).
The vectors $w_t$ are themselves recurrent, and the controller can strengthen them on certain key values, and/or shift them.

**Figure 2: Flow Diagram of the Addressing Mechanism.** The key vector, $k_t$, and key strength, $\beta_t$, are used to perform content-based addressing of the memory matrix, $M_t$. The resulting content-based weighting is interpolated with the weighting from the previous time step based on the value of the interpolation gate, $g_t$. The shift weighting, $s_t$, determines whether and by how much the weighting is rotated. Finally, depending on $\gamma_t$, the weighting is sharpened and used for memory access.

(Graves et al., 2014)
Results on the copy task

Figure 5: LSTM Generalisation on the Copy Task. The plots show inputs and outputs for the same sequence lengths as Figure 4. Like NTM, LSTM learns to reproduce sequences of up to length 20 almost perfectly. However, it clearly fails to generalise to longer sequences. Also note that the length of the accurate prefix decreases as the sequence length increases, suggesting that the network has trouble retaining information for long periods.

Figure 6: NTM Memory Use During the Copy Task. The plots in the left column depict the inputs to the network (top), the vectors added to memory (middle) and the corresponding write weightings (bottom) during a single test sequence for the copy task. The plots on the right show the outputs from the network (top), the vectors read from memory (middle) and the read weightings (bottom). Only a subset of memory locations are shown. Notice the sharp focus of all the weightings on a single location in memory (black is weight zero, white is weight one). Also note the translation of the focal point over time, which reflects the network’s use of iterative shifts for location-based addressing, as described in Section 3.3.2. Lastly, observe that the read locations exactly match the write locations, and the read vectors match the add vectors. This suggests that the network writes each input vector in turn to a specific memory location during the input phase, then reads from the same location sequence during the output phase.

Figures 4 and 5 demonstrate that the behaviour of LSTM and NTM in this regime is radically different. NTM continues to copy as the length increases, while LSTM rapidly degrades beyond length 20. The preceding analysis suggests that NTM, unlike LSTM, has learned some form of copy algorithm. To determine what this algorithm is, we examined the interaction between the controller and the memory (Figure 6). We believe that the sequence of operations performed by the network can be summarised by the following pseudocode:

initialise:
move head to start location

while input delimiter not seen
do
receive input vector
write input to head location
increment head location by 1
end while
return head to start location

while true
do
read output vector from head location
emit output
increment head location by 1
end while

This is essentially how a human programmer would perform the same task in a low-level script.

The limiting factor was the size of the memory (128 locations), after which the cyclical shifts wrapped around and previous writes were overwritten.

(Graves et al., 2014)
Results on the N-gram task

Figure 14: Dynamic N-Gram Inference. The top row shows a test sequence from the N-Gram task, and the rows below show the corresponding predictive distributions emitted by the optimal estimator, NTM, and LSTM. In most places the NTM predictions are almost indistinguishable from the optimal ones. However at the points indicated by the two arrows it makes clear mistakes, one of which is explained in Figure 15. LSTM follows the optimal predictions closely in some places but appears to diverge further as the sequence progresses; we speculate that this is due to LSTM “forgetting” the observations at the start of the sequence.

Figure 15: NTM Memory Use During the Dynamic N-Gram Task. The red and green arrows indicate points where the same context is repeatedly observed during the test sequence (“00010” for the green arrows, “01111” for the red arrows). At each such point the same location is accessed by the read head, and then, on the next time-step, accessed by the write head. We postulate that the network uses the writes to keep count of the fraction of ones and zeros following each context in the sequence so far. This is supported by the add vectors, which are clearly anti-correlated at places where the input is one or zero, suggesting a distributed “counter.” Note that the write weightings grow fainter as the same context is repeatedly seen; this may be because the memory records a ratio of ones to zeros, rather than absolute counts.

The red box in the prediction sequence corresponds to the mistake at the first red arrow in Figure 14; the controller appears to have accessed the wrong memory location, as the previous context was “01101” and not “01111.”

(Graves et al., 2014)
Figure 14: Dynamic N-Gram Inference. The top row shows a test sequence from the N-Gram task, and the rows below show the corresponding predictive distributions emitted by the optimal estimator, NTM, and LSTM. In most places the NTM predictions are almost indistinguishable from the optimal ones. However at the points indicated by the two arrows it makes clear mistakes, one of which is explained in Figure 15. LSTM follows the optimal predictions closely in some places but appears to diverge further as the sequence progresses; we speculate that this is due to LSTM “forgetting” the observations at the start of the sequence.

Figure 15: NTM Memory Use During the Dynamic N-Gram Task. The red and green arrows indicate points where the same context is repeatedly observed during the test sequence (“00010” for the green arrows, “01111” for the red arrows). At each such point the same location is accessed by the read head, and then, on the next time-step, accessed by the write head. We postulate that the network uses the writes to keep count of the fraction of ones and zeros following each context in the sequence so far. This is supported by the add vectors, which are clearly anti-correlated at places where the input is one or zero, suggesting a distributed “counter.” Note that the write weightings grow fainter as the same context is repeatedly seen; this may be because the memory records a ratio of ones to zeros, rather than absolute counts. The red box in the prediction sequence corresponds to the mistake at the first red arrow in Figure 14; the controller appears to have accessed the wrong memory location, as the previous context was “01101” and not “01111.”

(Graves et al., 2014)
Attention for seq2seq
Given an input sequence \( x_1, \ldots, x_T \), the standard approach for sequence-to-sequence translation (Sutskever et al., 2014) uses a recurrent model

\[
h_t = f(x_t, h_{t-1}),
\]

and considers that the final hidden state

\[
\nu = h_T
\]

carries enough information to drive an auto-regressive generative model

\[
y_t \sim p(y_1, \ldots, y_{t-1}, \nu),
\]

itself implemented with another RNN.
The main weakness of such an approach is that all the information has to flow through a single state $v$, whose capacity has to accommodate any situation.

There are no direct “channels” to transport local information from the input sequence to the place where it is useful in the resulting sequence.
Attention mechanisms (Bahdanau et al., 2014) can transport information from parts of the signal to other parts specified dynamically.
Bahdanau et al. (2014) proposed to extend a standard recurrent model with such a mechanism. They first run a bi-directional RNN to get a hidden state

\[ h_i = (h_i^{\rightarrow}, h_i^{\leftarrow}), \ i = 1, \ldots, T. \]

From this, they compute a new process \( s_i, \ i = 1, \ldots, T \) which looks at weighted averages of the \( h_j \), where the weight are functions of the signal.
Given $y_1, \ldots, y_{i-1}$ and $s_1, \ldots, s_{i-1}$ first compute an attention

$$\forall j, \alpha_{i,j} = \text{softmax}_j a(s_{i-1}, h_j)$$

where $a$ is a one hidden layer tanh MLP (this is “additive attention”, or “concatenation”).

Then compute the context vector from the $h$s

$$c_i = \sum_{j=1}^{T} \alpha_{i,j} h_j.$$
The model can now make the prediction

\[ s_i = f(s_{i-1}, y_{i-1}, c_i) \]
\[ y_i \sim g(y_{i-1}, s_i, c_i) \]

where \( f \) is a GRU (Cho et al., 2014).

This is context attention where \( s_{i-1} \) modulates what to look at in \( h_1, \ldots, h_T \) to compute \( s_i \) and sample \( y_i \).
Figure 2: The BLEU scores of the generated translations on the test set with respect to the lengths of the sentences. The results are on the full test set which includes sentences having unknown words to the models.

(Bahdanau et al., 2014)
The agreement on the European Economic Area was signed in August 1992.

It should be noted that the marine environment is the least known of environments.

Destruction of the equipment means that Syria can no longer produce new chemical weapons.

"This will change my future with my family," the man said.

One of the motivations behind the proposed approach was the use of a fixed-length context vector in the basic encoder–decoder approach. We conjectured that this limitation may make the basic encoder–decoder approach to underperform with long sentences. In Fig. 2, we see that the performance of RNNencdec dramatically drops as the length of the sentences increases. On the other hand, both RNNsearch-30 and RNNsearch-50 are more robust to the length of the sentences. RNNsearch-50, especially, shows no performance deterioration even with sentences of length 50 or more. This superiority of the proposed model over the basic encoder–decoder is further confirmed by the fact that the RNNsearch-30 even outperforms RNNencdec-50 (see Table 1).

(Bahdanau et al., 2014)
References


