CS224n: NLP with Deep Learning **Winter 2019**

Lecture 7

Course Coordinator: Prof. Chris Manning *Scribes: Akash Gupta*

Vanishing gradient intuition - Since the gradients in RNNs are computed using chain rule, the final gradient of loss function $J^{(t)}(\theta)$ w.r.t. some hidden state $h^{(t-l)}$ will get very small the indvidual gradients are small in a long sequence.

$->$ Formally, the gradient has a term with weights (W_h) in the product (Refer to the calculation) and if weights are small then the product is going to be exponentially small due to L2 norm.

$$||u.v|| \leq ||u||\,||v||$$

Why is that a problem? - Gradient signal from faraway while backpropagating is going to get minimized and we will be making updates mostly w.r.t. nearby gradients, thereby losing the long term effects.

$->$ In a long sentence in a LM task, RNN-LM is not able to learn long-term dependencies and gives the prediction based on a near dependencies.

Effects of vanishing gradients on RNN-LM -

$->$ LM-task: The writer of the books (blank) $->$ Correct answer: The writer of the books is planning a sequel. $->$ Syntactic recency: The writer of the books is $->$ due to writer $->$ Sequential recency: The writer of the books are $->$ 4 due to books

RNN-LMs are better at learning sequential recency than syntactic.

Why is exploding gradient a problem? - If the gradients are too big, update steo becomes too big. This can cause bad updates. You will Inf or NaN in the numbers.

Solution - Gradient clipping - if norm of the gradient is greater than some threshold, scale it down before applying SGD.

How to fix vanishing gradient problem? -

$->$ Long-Short Term Memory (LSTM) - A type of RNN solving the vanishing gradient problem.

- On step t, there is a hidden state $h^{(t)}$ and a cell state $c^{(t)}$. Both are n-length vectors. The cell stores long term info.
- LSTM can erase, read and write the cell state based on the forget, input and output gates respectively.
- Gates are also n-length vectors but have elements of 1(open) and 0(closed). They are dynamic at each timestep t.
- Full Architecture - Link
- How does is solve the problem - e.g. if we set the forget gate to remember everything then the info is preserved indefinitely (though this might not be a good idea).
- LSTMs have kinda like a route (cell state) for the faraway gradients to travel directly without going thru inside the cells.

Q) *Why in the equations do we have compute current cell state from previous hidden state and current input and why not from previous cell state itself?*
Ans) Might be learning a general algo where every step is designed to learn smth specific. Also, h^t is computed using c^t.

$->$ Gated Recurrent Units (GRUs) - Another faster type of RNN due to less computations and parameters

- It has 2 gates - update gate and reset gate.
- update gate controls what parts of hidden states are updated vs preserved
- reset gate controls what parts of previous hidden states are used to compute new hidden state.
- If update gate set to 0, retains info.

LSTM vs GRU - There are also many number of gated RNNs but these are pretty common. GRUs are quicker to compute. Rule of thumb - LSTM good default choice but switch to GRUs if performance not good or need faster computation.

Is vanishing/exploding gradient just an RNN problem? - No, it is also seen in deep feed-forward/conv networks. One solution is adding more direct connections. E.g. ResNet, Densenet - Skip connections, HighwayNet - Skip connections but instead of directly adding there is a gate that controls how much to add.

Bidirectional RNNs - E.g. in Sentiment Classification where starting from both ends is helpful.

$->$ We have a forward and a backward RNN and we concatenate the hidden states from both. They are calculated simultaneously but the weights are separate.

$->$ Only useful when we access to the entire input sequence. Not for a LM task. BERT is built on the idea of Bidirectionality.

Multi-layer RNNs - Also called Stacked RNNs since we are stacking RNN layers on top. While computing backprop, PyTorch completes it for a one layer and then moves to next layer.

$->$ They aren't as deep like CNNs or feed-forwards. In NMT, usually 2 to 4 layers. Transformers are deeper architectures (12 to 24 layers) as they have a lot of skip connections.