Natural Language Processing with Deep Learning

Language Models, RNN and LSTM

Myungjun Kim

Seoul National University

January 28, 2022

Contents

Language Modelings

Language Modeling is the task of predicting what word comes next.

the students opened their ______

Given a sequence of words $x^{(1)}, x^{(2)}, \ldots, x^{(t)}$, compute the probability distribution of the next word $x^{(t+1)}$:

$$P(x^{(t+1)} | x^{(1)}, \ldots, x^{(t)})$$

where $x^{(t+1)}$ can be any word in the vocabulary $V = \{w_1, \ldots, w_{|V|}\}$. A system that does this is called a **Language Model**.

Language Modelings

- We can think of a language model as a system that assigns probability to a piece of text.
- If we have some text $x^{(1)}, \ldots, x^{(t)}$, then the probability of this text is

$$
\begin{aligned}
P(x^{(1)}, \ldots, x^{(t)}) &= P(x^{(1)}) \times P(x^{(2)} | x^{(1)}) \times \cdots \times P(x^{(t)} | x^{(1)}, \ldots, x^{(t-1)}) \\
&= \prod_{t=1}^{T} P(x^{(t)} | x^{(1)}, \ldots, x^{(t-1)}).
\end{aligned}
$$

- The standard evaluation metric for LM is **perplexity**.

$$
\text{perplexity} = \prod_{t=1}^{T} \left(\frac{1}{P_{\text{LM}}(x^{(t+1)} | x^{(1)}, \ldots, x^{(t)})} \right)^{1/T} = \exp\left(J(\theta)\right)
$$

Why should we care about LM?

- Language modeling is a *benchmark task* that helps us to measure our progress on understanding language.
- Language modeling is a *subcomponent* of many NLP tasks, especially those involving generating text or estimating the probability of text, e.g.,
  - Predictive typing
  - Speech recognition
  - Handwriting recognition
  - Spelling/grammar correction
  - Machine translation
  - Summarization
  - Dialogue

n-gram Language Models

- **n-gram Language Model** is a pre-Deep learning language model.
- A **n-gram** is a chunk of *n* consecutive words.
  - unigrams: "the", "students", "opened", "their"
  - bigrams: "the students", "students opened", "opened their"
  - trigrams: "the students opened", "students opened their"
  - 4-grams: "the students opened their"
- The idea is to collect statistics about how frequent different n-grams are and use these to predict next word.

n-gram Language Models

Under *Markov assumption*, we have the following approximation:

$$\begin{aligned} P(x^{(t+1)} \mid x^{(1)}, \dots, x^{(t)}) &= P(x^{(t+1)} \mid x^{(t-n+2)}, \dots, x^{(t)}) \qquad \text{(Markov)} \\ &= \frac{P(x^{(t-n+2)}, \dots, x^{(t)}, x^{(t+1)})}{P(x^{(t-n+2)}, \dots, x^{(t)})} \\ &\approx \frac{\text{count}(x^{(t-n+2)}, \dots, x^{(t)}, x^{(t+1)})}{\text{count}(x^{(t-n+2)}, \dots, x^{(t)})} \end{aligned}$$

n-gram Language Models: Example

Suppose we are learning a 4-gram Language Model.

~~as the proctor started the clock, the~~ students opened their ______

For example, suppose that in the corpus:

- "students opened their" occurred 1000 times
- "students opened their books" occurred 400 times

$$P(\textit{books} \mid \text{students opened their}) = 0.4$$

- "students opened their exams" occurred 100 times

$$P(\textit{exams} \mid \text{students opened their}) = 0.1$$

Problems with n-gram Language Models

- Sparsity Problems
  - What if "students opened their *w*" never occurred in data?
  - What if "students opened their" never occurred in data?
- Storage Problems: we need to store counts for all n-grams in the corpus.

$$P(w|\text{students opened their}) = \frac{\text{count}(\text{students opened their } w)}{\text{count}(\text{students opened their})}$$

Generating text with a n-gram Language Model

We can also use a language model to generate text.

today the price of gold per ton, while production of shoe lasts and shoe industry, the bank intervened just after it considered and rejected an imf demand to rebuild depleted european stocks, sept 30 end primary 76 cts a share.

It is surprisingly grammtical but **incoherent**. We need to consider more than three words at a time if we want to model language well. But increasing n worsens sparisty problem, and increases model size.

Contents

How to build a neural Language Model?

$$\begin{tikzcd}
\textbf{output distribution} : \hat{y} = \mathrm{softmax}(Uh + b_2) \in \mathbb{R}^{|V|} \\
\textbf{hidden layer} : h = f(We + b_1) \arrow[u] \\
\textbf{concatenated word embeddings} : e = [e^{(1)}; e^{(2)}; e^{(3)}; e^{(4)}] \arrow[u] \\
\textbf{words/one-hot vectors} : x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)} \arrow[u]
\end{tikzcd}$$

A fixed-window neural Language Model

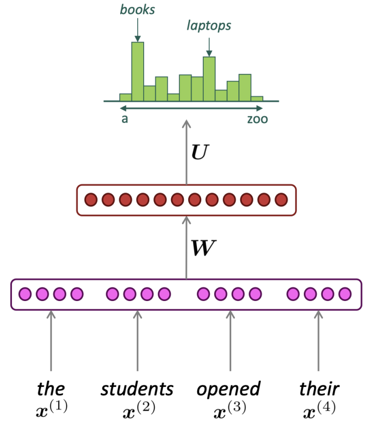


Figure 1: A fixed-window neural Language Model.

A fixed-window neural Language Model

- Improvements over n-grams LM:
  - No sparsity problem.
  - Don't need to store all observed n-grams.
- Remaining problems:
  - Fixed window is too small.
  - Enlarging window enlarges W and window can never be large enough.
  - $x^{(1)}$ and $x^{(2)}$ are multiplied by completely different weights in W.

Recurrent Neural Networks (RNN)

- **Recurrent Neural Networks** (RNN) are capable of conditioning the model on *all* previous words in the corpus.
- A key idea of RNN is to apply the same weights repeatedly.

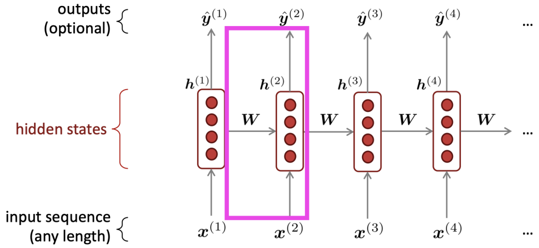


Figure 2: Recurrent Neural Network (RNN) is a family of neural architectures which apply the same weights repeatedly.

Recurrent Neural Networks (RNN)

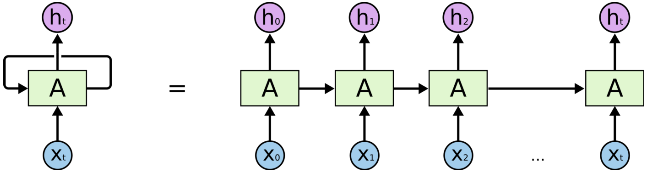


Figure 3: A recursive representation of recurrent neural networks.

A Simple RNN Language Model

- words/one-hot vectors: $x^{(t)} \in \mathbb{R}^{|V|}$
- word embeddings: $e^{(t)} = Ex^{(t)}$
- hidden state: $h^{(t)} = \sigma\left(W_h h^{(t-1)} + W_e e^{(t)} + b_1\right)$
- initial hidden state: $h^{(0)}$
- output distribution: $\hat{y}^{(t)} = \text{softmax}(Uh^{(t)} + b_2) \in \mathbb{R}^{|V|}$

A Simple RNN Language Model

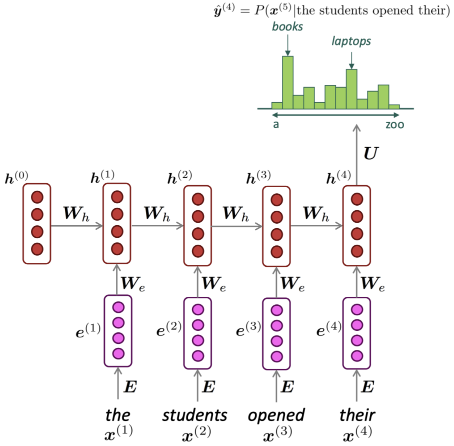


Figure 4: A RNN Lanaguage Model.

RNN Language Models

RNN is good because ...

- RNN can process *any length* input.
- Computation for step t can use information from many steps back. (in theory)
- Model size doesn't increase for longer input context.
- Same weights applied on every timestep, so there is *symmetry* in how inputs are processed.

However ...

- Recurrent computation is slow.
- It is difficult to access information from many steps back in practice.

Training a RNN Language Model

1. Get a big corpus of text which is a sequence of words $x^{(1)}, \ldots, x^{(T)}$.
2. Feed into RNN-LM to compute output distribution $\hat{y}^{(t)}$ for every step t.
   i.e., predict probability distribution of every word, given words so far.
3. Loss function on step t is

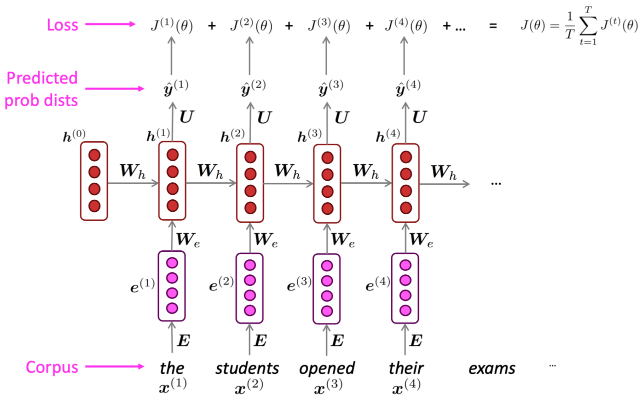
$$J^{(t)}(\theta) = H(y^{(t)}, \hat{y}^{(t)}) = -\sum_{w \in V} y_w^{(t)} \log \hat{y}_w^{(t)} = -\log \hat{y}_{x_{t+1}}^{(t)}$$

   which is the cross entropy between $\hat{y}^{(t)}$ and $y^{(t)}$.

4. Average this to get overall loss for entire training set:

$$J(\theta) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \log \hat{y}_{x_{t+1}}^{(t)}$$

Training a RNN Language Model



Loss
$J^{(1)}(\theta)$ + $J^{(2)}(\theta)$ + $J^{(3)}(\theta)$ + $J^{(4)}(\theta)$ + ... = $J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta)$
Predicted prob dists
$\hat{\boldsymbol{y}}^{(1)}$ $\hat{\boldsymbol{y}}^{(2)}$ $\hat{\boldsymbol{y}}^{(3)}$ $\hat{\boldsymbol{y}}^{(4)}$
$\boldsymbol{U}$
$\boldsymbol{h}^{(0)}$ $\boldsymbol{h}^{(1)}$ $\boldsymbol{h}^{(2)}$ $\boldsymbol{h}^{(3)}$ $\boldsymbol{h}^{(4)}$
$\boldsymbol{W}_h$
...
$\boldsymbol{W}_e$
$\boldsymbol{e}^{(1)}$ $\boldsymbol{e}^{(2)}$ $\boldsymbol{e}^{(3)}$ $\boldsymbol{e}^{(4)}$
$\boldsymbol{E}$
Corpus
the students opened their exams ...
$\boldsymbol{x}^{(1)}$ $\boldsymbol{x}^{(2)}$ $\boldsymbol{x}^{(3)}$ $\boldsymbol{x}^{(4)}$

Figure 5: The loss function of RNN language model with teacher forcing.

Backpropagation for RNNs

We need to compute the gradient with respect to W_e, W_h, U, b_1 and b_2. Note that

$$h^{(t)} = \sigma\left(W_h h^{(t-1)} + W_e e^{(t)} + b_1\right) \xrightarrow{\text{softmax}} \hat{y}^{(t)} \longrightarrow J(\theta) = \frac{1}{T}\sum J^{(t)}(\theta)$$

Then we have

$$\frac{\partial J^{(t)}}{\partial U} = \frac{\partial J^{(t)}}{\partial \hat{y}^{(t)}} \frac{\partial \hat{y}^{(t)}}{\partial U}$$

$$\frac{\partial J^{(t)}}{\partial W_h} = \frac{\partial J^{(t)}}{\partial \hat{y}^{(t)}} \frac{\partial \hat{y}^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial W_h} = \sum_{k=1}^{t} \frac{\partial J^{(t)}}{\partial \hat{y}^{(t)}} \frac{\partial \hat{y}^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial h^{(k)}} \frac{\partial h^{(k)}}{\partial W_h}$$

$$\frac{\partial J^{(t)}}{\partial W_e} = \frac{\partial J^{(t)}}{\partial \hat{y}^{(t)}} \frac{\partial \hat{y}^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial W_e} = \sum_{k=1}^{t} \frac{\partial J^{(t)}}{\partial \hat{y}^{(t)}} \frac{\partial \hat{y}^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial h^{(k)}} \frac{\partial h^{(k)}}{\partial W_e}$$

Exploding Gradient Problem

If the gradient becomes too big, then the update step becomes too big.

$$\theta^{\text{new}} = \theta^{\text{old}} - \alpha \nabla_\theta J(\theta)$$

This can cause bad updates: we take too large a step and reach a weird and bad solution. In the worst case, this will result in Inf or NaN.

Gradient Clipping

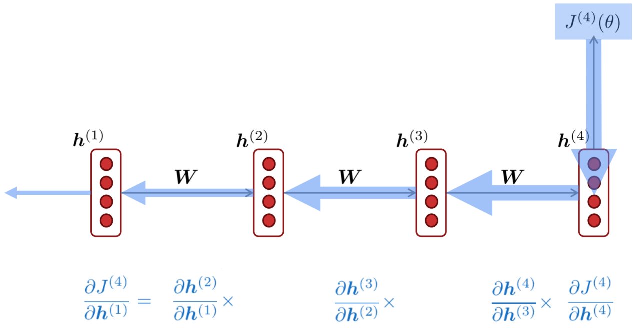
A typical solution for exploding gradient problem is **Gradient Clipping**: if the norm of the gradient is greater than some threshold, scale it down before applying update.

Algorithm 1 Pseudo-code for norm clipping

```
$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
**if** $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
    $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$
**end if**
```

Intuition: take a step in the same direction, but a smaller step.

Vanishing Gradient Problem



$$\frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(1)}} = \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{h}^{(1)}} \times \frac{\partial \boldsymbol{h}^{(3)}}{\partial \boldsymbol{h}^{(2)}} \times \frac{\partial \boldsymbol{h}^{(4)}}{\partial \boldsymbol{h}^{(3)}} \times \frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(4)}}$$

Figure 6: When the gradients are small, the gradient signal gets smaller and smaller as it backpropagates further.

Vanishing Gradient Problem

When she tried to print her tickets, she found that the printer was out of toner. She went to the stationery store to buy more toner. It was very overpriced. After installing the toner into the printer, she finally printed her ______

- To learn from this training example, the RNN-LM needs to *model the dependency* between "tickets" on the 7th step and the target word "tickets" at the end.
- But if gradient is small, the model cannot learn this dependency.
- So, the model is unable to predict similar long-distance dependencies at test time.

Contents

Long Short-Term Memory (LSTM)

- The main problem of RNN is vanishing gradient: it is too difficult for the RNN to learn to preserve information over many timesteps.
- The **Long Short-Term Memory** (LSTM) architecture makes it easier for the RNN to preserve information over many timesteps.
- The LSTM can *read*, *erase*, and *write* information from the cell.
- In 2013-2015, LSTMs started achieving state-of-the-art results.
- Now (2022), other approaches (e.g, Transformers) have become dominant for many tasks.

Long Short-Term Memory (LSTM)

We have a sequence of inputs $x^{(t)}$ and we will compute a sequence of hidden states $h^{(t)}$ and **cell states** $c^{(t)}$. On timestep t:

Forget gate: controls what is kept vs forgotten, from previous cell state

Input gate: controls what parts of the new cell content are written to cell

Output gate: controls what parts of cell are output to hidden state

Sigmoid function: all gate values are between 0 and 1

$$\left.\begin{aligned}
\boldsymbol{f}^{(t)} &= \boxed{\sigma}\left(\boldsymbol{W}_f \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_f \boldsymbol{x}^{(t)} + \boldsymbol{b}_f\right) \\
\boldsymbol{i}^{(t)} &= \boxed{\sigma}\left(\boldsymbol{W}_i \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_i \boldsymbol{x}^{(t)} + \boldsymbol{b}_i\right) \\
\boldsymbol{o}^{(t)} &= \boxed{\sigma}\left(\boldsymbol{W}_o \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_o \boldsymbol{x}^{(t)} + \boldsymbol{b}_o\right) \\
\tilde{\boldsymbol{c}}^{(t)} &= \tanh\left(\boldsymbol{W}_c \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_c \boldsymbol{x}^{(t)} + \boldsymbol{b}_c\right) \\
\boldsymbol{c}^{(t)} &= \boldsymbol{f}^{(t)} \circ \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \circ \tilde{\boldsymbol{c}}^{(t)} \\
\boldsymbol{h}^{(t)} &= \boldsymbol{o}^{(t)} \circ \tanh \boldsymbol{c}^{(t)}
\end{aligned}\right\} \text{All these are vectors of same length } n$$

New cell content: this is the new content to be written to the cell

Cell state: erase ("forget") some content from last cell state, and write ("input") some new cell content

Hidden state: read ("output") some content from the cell

Long Short-Term Memory (LSTM)

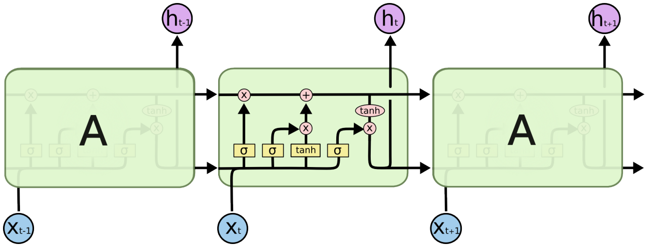
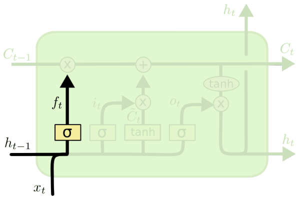


Figure 7: The repeating module in an LSTM contains four interacting layers.

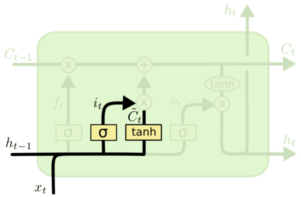
Forget Gate $f^{(t)}$



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

- The first step in LSTM is to decide what information we are going to throw away from the cell state.
- This decision is made by a sigmoid layer (forget gate): it looks at $h^{(t-1)}$ and $x^{(t)}$, and outputs a number between 0 and 1 for each number in the cell state $c^{(t-1)}$.

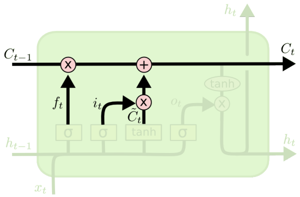
Input Gate $i^{(t)}$



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

- The next step is to decide what new information we are going to store in the cell state.
- First, a sigmoid layer (input gate) decides which values we will update.
- Next, a tanh layer creates a vector of new candidate values $\tilde{c}^{(t)}$.

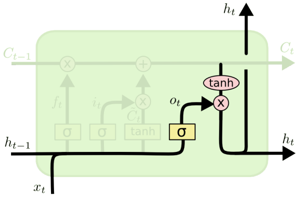
Update Cell State $c^{(t)}$



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

- Multiply the old state by $f^{(t)}$, forgetting the things we decided to forget earlier.
- Add $i^{(t)} \odot \tilde{c}^{(t)}$. This is the new candidate values, scaled by how much we decided to update each state value.

Output Gate $o^{(t)}$

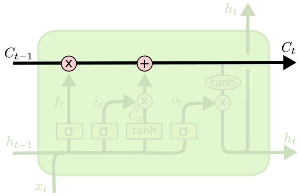


$$o_t = \sigma\left(W_o \left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

- Run a sigmoid layer (output gate) which decides what parts of the cell state we are going to output.
- Put the cell state $c^{(t)}$ through tanh and multiply it by $o^{(t)}$, so that we only output the parts we decided to do.

Remarks



- The key to LSTMs is the cell state. It runs straight down the entire chain, with only some minor linear interactions.
- The sigmoid layer outputs numbers between 0 and 1, describing how much of each component should be let through.
- LSTM does not guarantee that there is no vanishing/exploding gradient, but it does provide an easier way to learn long-distance dependencies.

References

[1] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult, 1994.

[2] G. Chen. A gentle tutorial of recurrent neural network with error backpropagation, 2018.

[3] F. A. Gers, J. A. Schmidhuber, and F. A. Cummins. Learning to forget: Continual prediction with lstm, oct 2000.

[4] D. Jurafsky and J. H. Martin. Speech and language processing (2nd edition), 2009.

[5] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks, 2013.