Deep Natural Language Processing

A/Prof Richard Yi Da Xu, Ember Liang
http://richardxu.com

University of Technology Sydney (UTS)

June 13, 2018

Natural Language Processing Tasks (1)

Too many of them here we list a few of what is going on in our lab:

- **machine translation**: *encoder to decoder*
  automatically translate text from one human language to another, for example, English to Chinese. Since 2014, Neural Machine Translation (NMT) dominates!
- **text summerization**: *context to decoder*
  1. **Extraction-based summarization** extracts objects (part-sentences or words) form the long document without modification, i.e., pick the important bits
  2. **abstraction-based summarization**
     involves paraphrasing sections of the source document
- **Q and A**: *encoder to decoder given context*

  the above three (3) may share a design architecture/elements

Natural Language Processing Tasks (2)

Too many of them here we list a few of what is going on in our lab:

- **natural language generation** by learning document corpus
  generate natural language from a machine representation, or for machine to generate semantically-similar texts given a training corpus
- **chatbot**
  enable human and machine to communicate using natural language
- **natural language to cross-domain translation**
  1. NLP to image
  2. NLP to animation
- **topic modeling**
  this is **unsupervised learning**, tries to assign each document in the document corpus a latent distribution of topics
- Before we get into it, let's study the foundation of Deep NLP: Recurrent Neural Networks
- before 2018, it is the primary design element of any D-NLP!

Recurrent Neural Networks

- RNN equations are simple, it has three sets of parameters: (W, V, U)

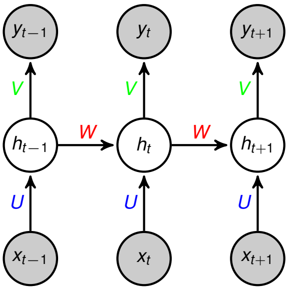
$$h_t = \tanh(\underbrace{Ux_t + Wh_{t-1}}_{z_t}) \qquad \hat{y}_t = \text{softmax}(Vh_t)$$

- The overall loss can be defined as sum of **cross entropy**:

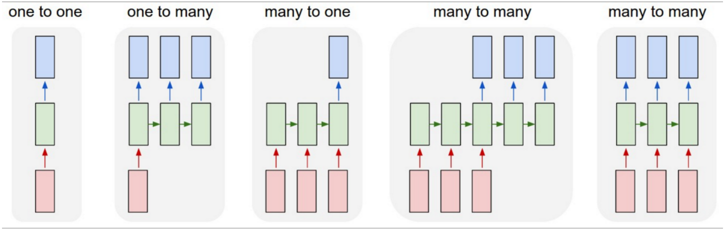
$$\mathcal{C}(y, \hat{y}) = \sum_t \mathcal{C}_t(y_t, \hat{y}_t) = -\sum_t \underbrace{\sum_{i \in \mathbb{S}} y_t \log \hat{y}_t}_{\text{-ve of cross entropy loss}}$$

- The overall loss can also be defined as sum of **square error**:

$$\mathcal{C}(y, \hat{y}) = \sum_t \mathcal{C}_t(y_t, \hat{y}_t) = \sum_t \sum_{\mathbb{S}} (y_{t,i} - \hat{y}_{t,i})^2$$



Various applications of RNNs



- each configuration serves a different applications
- let's discuss about the scenarios for their use

Back propagation of Vanilla RNN $\frac{\partial \mathcal{C}_t}{\partial V}$

$$h_t = \tanh(\underbrace{Ux_t + Wh_{t-1}}_{z_t}) \qquad \hat{y}_t = \mathrm{softmax}(\underbrace{Vh_t}_{b_t})$$

$$\mathcal{C}(y, \hat{y}) = \sum_t \mathcal{C}_t(y_t, \hat{y}_t) = -\sum_t \sum_{\mathbb{S}} y_t \log \hat{y}_t$$

where $\mathbb{S}$ is the output space, e.g., all the words we try to predict.

$$\begin{aligned}
\frac{\partial \mathcal{C}_t(y_t, \hat{y}_t)}{\partial V} &= \frac{\partial \mathcal{C}_t(y_t, \hat{y}_t)}{\partial b_t} \frac{\partial b_t}{\partial V} \\
&= \frac{\partial \left(-\sum_{\mathbb{S}} y_t \log \hat{y}_t\right)}{\partial b_t} \times \underbrace{\frac{\partial b_t}{\partial V}}_{\text{a vector}} \\
&= (\hat{y}_t - y_t) h_t^\top
\end{aligned}$$

Back propagation for $\frac{\partial \mathcal{C}_t}{\partial W}$

$$h_t = \tanh(\underbrace{Ux_t + Wh_{t-1}}_{z_t}) \qquad \hat{y}_t = \text{softmax}(\underbrace{Vh_t}_{b_t})$$

$$\mathcal{C}(y, \hat{y}) = \sum_t \mathcal{C}_t(y_t, \hat{y}_t) = -\sum_t \sum_{\mathbb{S}} y_t \log \hat{y}_t$$

- Looking at **individual** cost term $\mathcal{C}_t$:

$$\frac{\partial \mathcal{C}_t}{\partial W} = \left(\frac{\partial \mathcal{C}_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \right) \frac{\partial h_t}{\partial W} = \left(\frac{\partial \mathcal{C}_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \right) \sum_{k=0}^{t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W}$$

- when performing $\frac{\partial h_t}{\partial W}$, we need to **sum** over all intermediate latent nodes, i..e.,

$$\left(\frac{\partial h_t}{\partial h_1} \frac{\partial h_1}{\partial W} \right) + \left(\frac{\partial h_t}{\partial h_2} \frac{\partial h_2}{\partial W} \right) + \cdots + \left(\frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial W} \right)$$

- rewrite it to fill in the gap with chain rule:

$$\frac{\partial \mathcal{C}_t}{\partial W} = \sum_{k=0}^{t} \frac{\partial \mathcal{C}_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \left(\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_k}{\partial W}$$

- we need to sum over all $\mathcal{C}_t$

Back propagation for $\frac{\partial \mathcal{C}_t}{\partial W}$ (1)

$$h_t = \tanh(\underbrace{Ux_t + Wh_{t-1}}_{z_t}) \qquad \hat{y}_t = \text{softmax}(Vh_t)$$

$$\begin{aligned}
\frac{\partial \mathcal{C}_t}{\partial W} &= \sum_{k=0}^{t} \frac{\partial \mathcal{C}_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \left(\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_k}{\partial W} \\
&= \sum_{k=0}^{t} \frac{\partial \mathcal{C}_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \left(\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_k}{\partial z_k} \frac{\partial z_k}{\partial W}
\end{aligned}$$

- The following has $t + 1$ term, each with varying length due to the product term.
- Derivations can be understood better: $h_2 \left(\underbrace{c_2 + W(h_1(c_1 + W)}_{z_2} \right)$

$$\begin{aligned}
&\frac{\partial h_2 \left(c_2 + W(h_1(c_1 + W)\right)}{\partial W} \\
&= h_2'(c_2 + W(f(c_1 + W)) \frac{\partial (c_1 + W(h_1(c_1 + W))}{\partial W} && \text{using chain rule} \\
&= h_2'(c_2 + W(f(c_1 + W)) \left(h_1(c_1 + W) + Wh_1'(c_1 + W) \right) && \text{using product rule} \\
&= h_2'(c_2 + W(f(c_1 + W))h_1(c_1 + W) + h_2'(c_2 + W(h(c_1 + W))Wh_1'(c_1 + W) \\
&= \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial W} + \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial h_1} \frac{\partial h_1}{\partial W} = \frac{\partial h_2}{\partial W} + \frac{\partial h_2}{\partial h_1} \frac{\partial h_1}{\partial W}
\end{aligned}$$

Gradient Vanishing and/or Explosion

$$\frac{\partial \mathcal{C}_t}{\partial W} = \sum_{k=0}^{t} \frac{\partial \mathcal{C}_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \left(\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_k}{\partial z_k} \frac{\partial z_k}{\partial W}$$

before

$$h_t = \tanh(Ux_t + Wh_{t-1})$$
$$\hat{y}_t = \mathrm{softmax}(Vh_t)$$

hard to analyse $\frac{\partial h_t}{\partial h_k}$

alternative

$$h_t = Ux_t + Wf(h_{t-1})$$
$$\hat{y}_t = Vf(h_t)$$

easier to analyse $\frac{\partial h_t}{\partial h_k}$

In alternative represenation:

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=k+1}^{t} W \times \text{diag}[f'(h_{j-1})]$$

This is because:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} w_{1,1}x_1 + w_{1,2}x_2 \\ w_{2,1}x_1 + w_{2,2}x_2 \end{bmatrix} \Longrightarrow \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} \end{bmatrix} = W$$

Gradient vanishing and/or exploding: Matrix norm

Define matrix norm from vector norm:

$$\|A\| = \sup\{ \underbrace{\|Ax\|}_{\text{vector norm}} : x \in \mathbb{R}^n \text{ with } \underbrace{\|x\|}_{\text{vector norm}} = 1\}$$

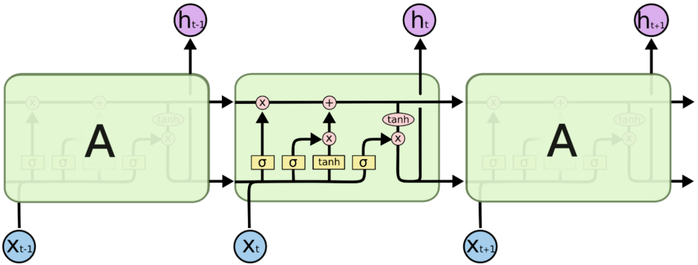
$$\left\| \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq \beta_W \beta_s$$

$$\left\| \frac{\partial h_t}{\partial h_k} \right\| = \left\| \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right\| = \left\| \prod_{j=k+1}^{t} W \times \text{diag}[f'(h_{j-1})] \right\| \leq (\beta_W \beta_s)^{t-k}$$

Possible solution:

- Let $f(x) = \max(0, x)$, i.e., another activation function, for example, ReLU helps with gradient.
- Initialise W to be the identity matrix.

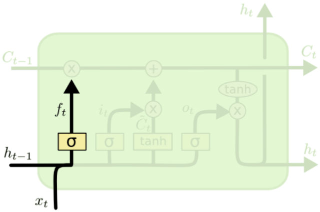
Looking at very complicated structure. But it works!



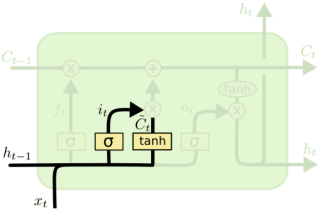
- There is a concept of Cell State $\{C_t\}$ in addition to state $\{h_t\}$.
- `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Long Short Term Memory (LSTM): forget and input gate

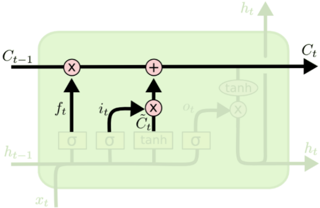
forget gate:
$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$



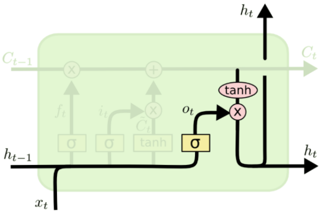
input gate:
$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$
$\tilde{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C)$



state update:
$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$



output gate:
$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$
$h_t = o_t \odot \tanh(C_t)$



more on LSTM

- a compact form of representation:

$$\begin{bmatrix} i \\ f \\ o \\ \tilde{C} \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} W \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} \qquad C_t = f_t \odot C_{t-1} + i \odot \tilde{C}_t$$

$$h_t = o_t \odot \tanh(C_t)$$

- in vanilla RNN, multiple by the same **W**, in LSTM, f_t changes each time step
- element-wise multiplication (LSTM) is nicer than full matrix multiplication RNN
- in LSTMs, cell state C_t. The derivative of consecutive States is of the form:

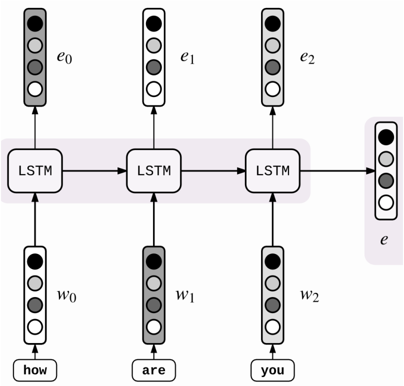
$$\begin{aligned} C_t &= f_t \times C_{t-1} + i_t \times \tilde{C}_t \\ &= f_t \times C_{t-1} + i_t \times \tanh(W_C[h_{t-1}, x_t] + b_C) \\ &= f_t(C_{t-1})C_{t-1} + \underbrace{i_t(h_{t-1}(C_{t-1})) \times \tanh(W_C[o_{t-1}(h_{t-1}(C_{t-1})) \times \tanh(C_{t-1}), x_t] + b_C)}_{\xi(C_{t-1})} \end{aligned}$$

$$\frac{\partial C_t}{\partial C_{t-1}} = \underbrace{f_t}_{\text{gradient-super highway}} + \underbrace{\frac{\partial f_t}{\partial C_{t-1}} C_{t-1} + \frac{\partial \xi(C_{t-1})}{\partial C_{t-1}} C_{t-1}}_{\text{contains exponentially fast decay function}}$$

- of course, f_t may still close to zero
- **trick is to** initialize bias to positive, e.g., $f_t = \sigma(W_f[h_{t-1}, x_t] + +ve)$ so to make f_t closer to 1 initially

Vanilla Seq2Seq: encoder

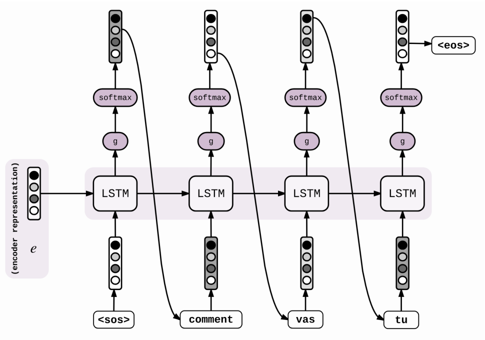
- *Sutskever et., al, 2014, Sequence to sequence learning with neural networks*
- at encoder: last neural representation e "**summerizes**" entire encoder sentence
- this neural representation is to be used at the decoder
- it uses RNN(LSTM) each time t



https://guillaumegenthial.github.io/sequence-to-sequence.html

Vanilla Seq2Seq: decoder

- *Sutskever et., al, 2014, Sequence to sequence learning with neural networks*
- at decoder: it uses last neural representation *e* from encoder
- it generates one word at the time
- during **training**, decoder sentence to minimize the cross entropy error



https://guillaumegenthial.github.io/sequence-to-sequence.html

Seq2Seq: beam-search (1)

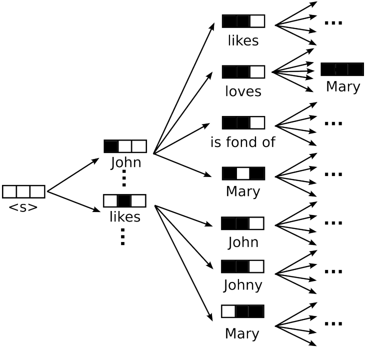
- in theory, decoder generates words **jointly**, so how we may compute:

$$\{\widehat{y}_1, \dots, \widehat{y}_T\} = \underset{y_1, \dots, y_T}{\arg\max} \Big[\Pr(y_1, \dots, y_T | \mathbf{x}) \equiv \Pr(y_1 | \mathbf{x}) \Pr(y_2 | y_1, \mathbf{x}), \dots, \Pr(y_T | y_1, \dots, y_{T-1}, \mathbf{x}) \Big]$$

1. **select all**: tree-width $= N$, so we get answer to be:

$$\{\widehat{y}_1, \dots, \widehat{y}_T\} = \underset{y_1, \dots, y_T}{\arg\max} \Big[\Pr(y_1 | \mathbf{x}) \Pr(y_2 | y_1, \mathbf{x}) \Pr(y_3 | y_1, y_2, \mathbf{x}) \\ \dots, \Pr(y_T | y_1, \dots, y_{T-1}, \mathbf{x}) \Big]$$

   in each depth, keep (select) **full width** N, until its full depth T, before select a best path
   (accurate, but computationally infeasible): N^T paths!



2. **select one**: tree-width $= 1$, **greedy algorithm** (*def. making locally optimal choice at each stage, to "approximately" a global optimum*)
   select best word at each depth t: choose one branch in a depth, and discard rest of sibling branches
   (fast & storage efficient, but accuracy-wise bad)

$$\{\widehat{y}_1, \dots, \widehat{y}_T\} \approx \{\tilde{y}_1, \dots, \tilde{y}_T\} \\ = \Big\{ \tilde{y}_1 \equiv \underset{y_1}{\arg\max} \Pr(y_1 | \mathbf{x}), \ \tilde{y}_2 = \underset{y_2}{\arg\max} \Pr(y_2 | \tilde{y}_1, \mathbf{x}), \dots, \tilde{y}_T \equiv \underset{y_T}{\arg\max} \Pr(y_T | \tilde{y}_1, \dots, \tilde{y}_{T-1} \mathbf{x}) \Big\}$$

beam-search (2)

- tree width of N and 1 are both **not ideal**, so we go for a comprise
- easy, select tree width of $1 < W < N$:
- loop from 1 to T, at each depth t:

1. use W **most probable** branches chosen at the previous depth
2. extend each W branch by depth $+1$, and to generate $W \times N$ candidate branches
3. choose the W **most probable** branches, and go for next iteration

beam-search: normalization

- **problem:** words are generated from

$$\Pr(y_1, \dots, y_T|\mathbf{x}) = \Pr(y_1|\mathbf{x}) \Pr(y_2|y_1, \mathbf{x}) \Pr(y_3|y_1, y_2, \mathbf{x}) \Pr(y_T|y_1, \dots, y_{T-1}, \mathbf{x})$$

therefore, shorter the word, higher the probability (less to multiply)

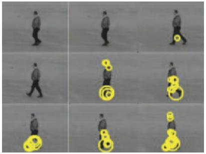
solution: beam-search normalization
one simple example: Andrew Ng's (2017) DL course:

$$\begin{aligned}\{\widehat{y}_1, \dots, \widehat{y}_T\} &= \arg\max_{y_1, \dots, y_T} \Pr(y_1, \dots, y_T|\mathbf{x}) = \Pr(y_1|\mathbf{x}) \Pr(y_2|y_1, \mathbf{x}), \dots, \Pr(y_T|y_1, \dots, y_{T-1}, \mathbf{x}) \\ &= \arg\max_{y_1, \dots, y_T} \left(\frac{1}{T^\alpha} \sum_{t=1}^{T} \log \Pr(y_t|y_1, \dots, y_{t-1}, \mathbf{x}) \right)\end{aligned}$$

- the method is not new, it's called **geometry mean**

$$\left(\prod_{i=1}^{T} p_i \right)^{\frac{1}{T}} = \exp \left[\frac{1}{T} \sum_{i=1}^{T} \ln p_i \right]$$

- I also used geometry mean to address problem of variable number of features at each image in a sequence

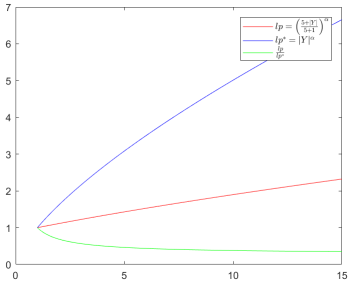


*Concha, **Xu**, Moghaddam, Piccardi (2011), HMM-MIO: An enhanced hidden Markov model for action recognition*

beam-search: more sophisticated normalization (1)

- *Wu et. al., (2016), "Google's Neural Machine Translation System: Bridging the Gap"*
- to **maximize** scores generated by the model:

$$s(Y, X) = \frac{1}{\text{lp}(Y)} \log P(Y|X) + \text{cp}(X, Y)$$



- instead of normalize by the length $\text{lp}^* = |Y|^{\alpha}$, it uses a different normalization $\text{lp}(Y)$:

$$\text{lp}(Y) = \frac{(5 + |Y|)^{\alpha}}{(5 + 1)^{\alpha}}$$

- $\text{lp}(Y)$ penalizes as much as $\text{lp}^*(Y)$ when $|Y|$ is small, then it "gently" drops as $|Y|$ increases

beam-search: more sophisticated normalization (2)

$$s(Y, X) = \frac{1}{\text{lp}(Y)} \log P(Y|X) + \text{cp}(X, Y)$$

- it needs also to maximize **coverage penalty**:

$$\text{cp}(X, Y) = \beta \sum_{j=1}^{|X|} \log \left(\min \left(\sum_{i=1}^{|Y|} a_{i,j}, 1.0 \right) \right)$$

  where $a_{i,j}$ is attention probability of i-th target **decoder** word on j-th source **encoder** word
- we know that:

$$\sum_{j=1}^{|X|} a_{i,j} = 1 \qquad \text{and} \qquad \sum_{i=1}^{|Y|} a_{i,j} \neq 1 \text{ in general}$$

- think **a** as a matrix of size $|Y| \times |X|$, which we distribute a total mass of $|Y|$ in value among all of its elements, for example, $|X| = |Y| = 3$:

$$\underbrace{\mathbf{a} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}}_{\textbf{minimized}\ \text{cp}(X,Y)} \qquad \underbrace{\mathbf{a} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\textbf{maximized}\ \text{cp}(X,Y)}$$

- favor translations that fully cover source sentence according to the attention module
- lastly, one may encourage the decoder to be longer than the encoder: `opennmt.net/OpenNMT/translation/beam_search/`

$$\text{ep}(X, Y) = \gamma \frac{|Y|}{|X|}$$

Sequence to Sequence with Attention

- **encoders** have hidden states, $\{z_1, \dots, z_m\} \in \mathbb{R}^h$
- **decoders** have hidden states, $\{h_1, \dots, h_n\} \in R^h$
- compute **dot-product**: $e_{i,j} = h_i^\top z_j$
- **attentions** between i^{th} decoder state and j^{th} encoder state is:

$$a_{ij} = \frac{\exp\left(e_{i,j}\right)}{\sum_{t=1}^{m} \exp\left(e_{i,t}\right)} = \frac{\exp\left(h_i^\top z_j\right)}{\sum_{t=1}^{m} \exp\left(h_i^\top z_t\right)}$$

- each i^{th} decoder has $\mathbf{a}_i = \{a_{i,1}, \dots, a_{i,m}\}$ attention weights of the encoder
- **condition vector** c_i for each decode word i:

$$c_i = \sum_{j=1}^{m} a_{i,j}(z_j)$$

- new **augmented decoder state** $\tilde{h}_t$:

$$\tilde{h}_t = [c_i; h_t] \in \mathbb{R}^{2h}$$

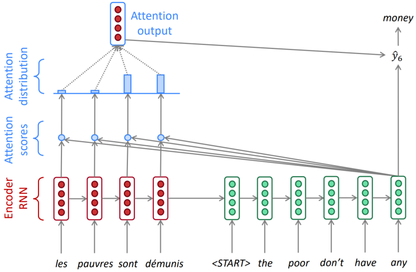


image from Stanford CS224n 2018

something about **dot-product**: $e_{i,j} = h_i^\top z_j$, many version exist:

1. $e_{i,j} = h_i^\top \mathbf{W} z_j$
   enable h and z have different dimensionality
2. $e_{i,j} = v_i^\top \tanh(\mathbf{W_1} h_i + \mathbf{W_2} z_j)$
   $(v_i, \mathbf{W_1}, \mathbf{W_2})$ are parameters of this dot-product
   Pointer Networks uses this!

Sequence to Sequence with Attention: issues (1)

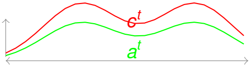
- **issue one**: decoder sometimes repeat themselves (e.g. "machine learning machine learning ...")
solution *(See, et., al, 2017), Get To The Point: Summarization with Pointer-Generator Networks*
  - **coverage vector** Sum of attention distributions so far:

$$c^t = \sum_{t'=0}^{t-1} a^{t'}$$

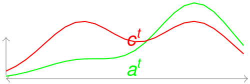
  - penalize overlap between **coverage vector** c^t and new attention distribution a^t:

$$\text{covloss}_t = \sum_i \min(a_i^t, c_i^t)$$

  - the above equation can be understood as follows:
imagine $c_i^t \geq a_i^t \forall i$, then $\text{covloss}_t = 1$, which is its **maximum**, this happens when covloss_t is a multilicative envelop of a^t:



$\text{covloss}_t = \sum_i a_i^t = 1$ at maximum



$\text{covloss}_t < 1$

  - in essence, covloss_t tries to make a^t distributed differently to c^t

Sequence to Sequence with Attention: issues (2)

- **issue two** decoder may not able to translate "out-of-vocabulary words" such as names of a company
- suppose to have the following text summerization task:
  - **original text**:
    "The QueenslandCo has made all reasonable efforts to ensure that this material has been reproduced with the consent of NSWCo"
  - **summerized text**:
    "NSWCo allowed QueenslandCo to reuse its content"
  - some of the word should appear **as is it**
- RNN-based summarization may replace "Mary" with "Jane" and "Sydney" with "Melbourne" since these word embedding tend to cluster (and hence their dot product are similar!)
- **solution** "Pointer Networks" may be handy to comes to help!

What is Pointer Networks anyway?

- *(Vinyals, 2016), Pointer Networks*
- "Seq2Seq with attention" is to predict **content** of next word
- "Pointer Networks" is to predict next **position** of encoding sequence
- $e_{i,j} = v_i^\top \tanh(\mathbf{W_1} h_i + \mathbf{W_2} z_j)$

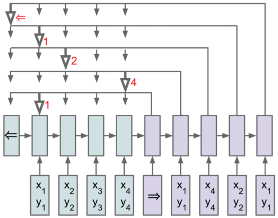
$$a_{ij} = \frac{\exp\left(e_{i,j}\right)}{\sum_{t=1}^{m} \exp\left(e_{i,t}\right)} = \frac{\exp\left(h_i^\top z_j\right)}{\sum_{t=1}^{m} \exp\left(h_i^\top z_t\right)}$$

- instead of compute **conditional vector** c_i and concatenate with h_i as the case of "Seq2Seq with attention", it performs:

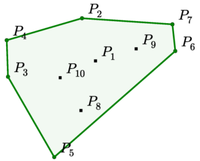
$$\Pr(C_i | C_1, \ldots, C_{i-1}, \mathcal{P}) = \text{softmax}(e_{i,1}, \ldots, e_{i,n})$$

- now that we apply Pointer Network to "copy" rare words from encoder to decoder, what about generating words that don't appear in the encoder?
- the answer is a **mixture model** that does both **copy (extraction)** and **generation (abstraction)**

- pointer network structure:

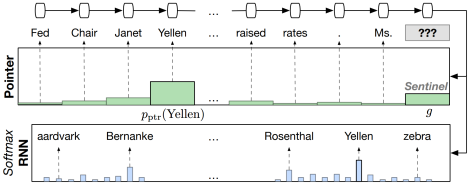


- it could solve combinatorial geometry problems:



Pointer Sentinel Mixture Models (1)

- *(Merity, 2016), Pointer sentinel mixture models*
- combines **abstraction** and **extraction** together



$$p(\text{“}Yellen\text{''}) = g \times p_{\text{vocab}}(\text{“}Yellen\text{''}) + (1 - g) \times p_{\text{ptr}}(\text{“}Yellen\text{''})$$

- g is mixture gate, uses sentinel to dictate how much probability mass to give to vocabulary
- note that PSMM paper doesn't discuss seq2seq, instead it is about generate $\Pr(y_N | w_1, \dots, w_{N-1})$

Pointer Sentinel Mixture Models (2): its design

- simplest way to compute an attention score with all past hidden states $\{h\}_i = 1^{N-1}$, with each hidden state $h_i \in \mathbb{R}^H$
- However, when computing score for most recent word with hidden state h_N, if it's a **repeating word** with previous hidden state h_{N-1}, then $h_N^\top h^{N-1} = \|h_N\|_{L2}^2$, i.e., big, and hence it is more likely to generate itself again!
- the paper hence project the previous hidden state h_{N-1} to a query vector q:

$$q = \tanh(W h_{N-1} + b)$$

- so, that dot-product between "candidate state" and "previous state" pair is no longer $e_{N,N-1} = h_N^\top h_{N-1}$, instead it's $e_{N,N-1} = h_N^\top q$
- rest is standard: $a = \text{softmax}(e)$

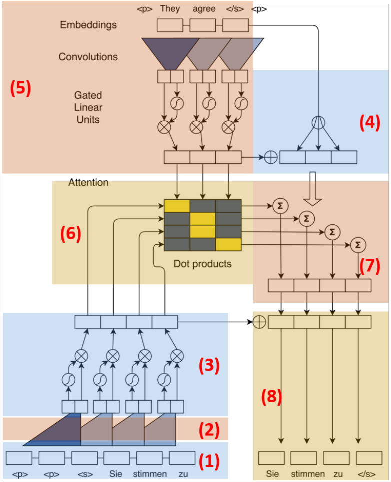
Beyond Seq2Seq with Attention using LSTM!

- Sequence-to-sequence (Seq2Seq) using LSTM building block has been **the method** since 2015!
- however, LSTM cannot be parallelized, so what then? two replacement methods stood out in second half of 2017:
- “attention is all you need” (Google Research)
- “Convolution Sequence to Sequence” (Facebook AI Research)

- *(Gehring, 2017), Convolutional sequence to sequence learning*
- **(1)** these are decoder raw inputs $\{g_i \equiv h_i^{(0)}\}$ representing input sentence $\{x_1, \ldots, x_n\}$
- **(2)** concatenate features within window size k:
  1. for each decoder position i, take a k element set $\{g_i \equiv h_i^{(0)}\}$: $\{h^{(0)}_{(i-k)/2}, \ldots, h^{(0)}_{(i+k)/2}\}$
  2. concatenate to form a vector $\hat{h}_i^{(0)}$, which has size $k \times d$
- **(3)** repeat the above two steps for several layers, relationship between current l and previous $l-1$ layers are:

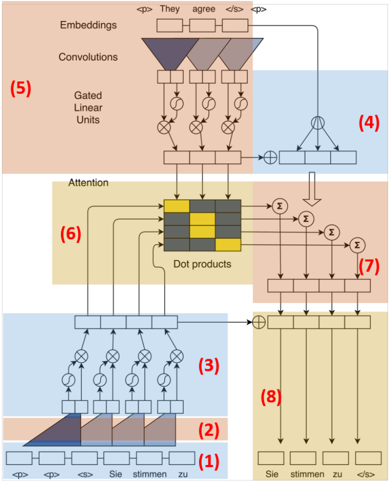
$$h_i^{(l)} = v\big(W^l \hat{h}_i^{(l-1)}\big) \qquad \text{where } v(.) \text{ is } \textbf{gated convolution}$$

- during **training**, each **decoding word** g_i can be embedded to $h_i^{(l)}$ in parallel



Convolution Sequence to Sequence (2): Encoder

- **(4)** for **encoder** produce sequence $\{e_1, \dots e_m\}$ where $e_j = w_i + p_i$: word embedding + position embedding
- **(5)** the process to embed encoder sequence into the last layer: $\{z_1^u, \dots, z_m^u\}$ using same process as decoder



Convolution Sequence to Sequence (3): Put together

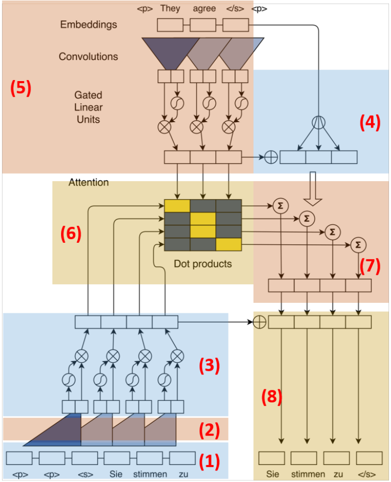
- **(6)** to compute attention $a_{ij}^{(l)}$:

$$d_i^{(l)} = W_d^{(l)} h_i^{(l)} + b_d^{(l)} + g_i \qquad a_{ij}^{(l)} = \frac{\exp\left({d_i^{(l)}}^\top z_j^{(u)}\right)}{\sum_{t=1}^m \exp\left({d_i^{(l)}}^\top z_t^{(u)}\right)}$$

- note that this is slightly different to the diagram: the paper has only dot product term $(d_i^{(l)} \odot z_j^{(u)})$
- **(7)** to compute condition vector c_i^l for each decoding word i:

$$c_i^{(l)} = \sum_{j=1}^m a_{ij}^{(l)} (z_j^{(u)} + e_j)$$

- **(8)** to generate the sequence using predict y_{i+1} from $\{c_i^l + h_i^l\}_{l=1}^L$
- **during testing**:
  - y_i is generated one at the time
  - the "dot product" table in **(6)** is increase one row at the time



Gated Convolutional Network

- so what is **gated convolution** used in convolutional seq2seq?
- *Yann N. Dauphin, Language Modeling with Gated Convolutional Networks*
- look at last box **gating**, reduce vanishing gradient problem:

1. gradient of LSTM-style gating:

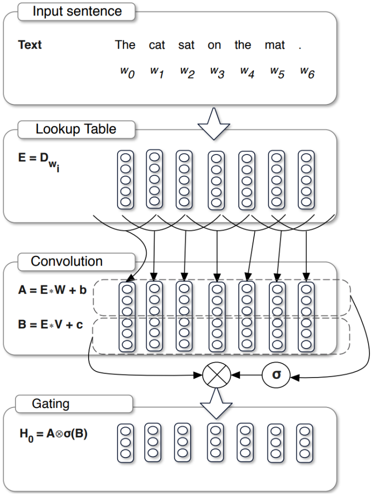
$$\begin{aligned} h_t &= o_t \odot \tanh(C_t) \\ &= \sigma(W_o[h_{t-1}, x_t] + b_o) \odot \tanh(C_t) \end{aligned}$$

writing it more generically:

$$\nabla[\tanh(X) \odot \sigma(X)]$$
$$= \underbrace{\tanh'(X)}_{\text{down scaling}} \nabla X \odot \sigma(X) + \underbrace{\sigma'(X)}_{\text{down scaling}} \nabla(X) \odot \tanh(X)$$

2. Gated Convolution Networks

$$\nabla[X \odot \sigma(X)]$$
$$= \underbrace{\nabla X \odot \sigma(X)}_{\text{no down scaling path}} + \underbrace{\sigma'(X)}_{\text{down scaling}} \nabla(X) \odot X$$



Transformer Networks: Dot-Product Attention

- let (q, K, V) be tuples: the Dot-Product Attention (DPA) is defined as:

$$A(q, K, V) = \sum_i \underbrace{\frac{\exp[q^\top k_i]}{\sum_j \exp[q^\top k_j]}}_{a_i} v_i$$

- in the case of **seq2seq with attention**:
  - $q = h$
  - $k_i = v_i = z_i$
  - $A(q, K, V) = A(h_i, z, z) = c_i$, where is our **conditional** or **context** vector:

$$a_{ij} = \frac{\exp\left(e_{i,j}\right)}{\sum_{t=1}^m \exp\left(e_{i,t}\right)} = \frac{\exp\left(h_i^\top z_j\right)}{\sum_{t=1}^m \exp\left(h_i^\top z_t\right)}$$

- now we have many $Q = \{q_i\}$, e.g., N words in the decoder, we can rewrite it as:

$$A(Q, K, V) = \mathrm{softmax}(QK^\top)V$$

$$\underbrace{\underbrace{\begin{bmatrix} - & q & - \end{bmatrix}}_{d_k} \left.\begin{bmatrix} | & \vdots & | \\ k_1 & \vdots & k_m \\ | & \vdots & | \end{bmatrix}\right\} d_k \begin{bmatrix} - & v_1 & - \\ \vdots & \cdots & \vdots \\ - & v_m & - \end{bmatrix}}_{A(q,K,V)} \Longrightarrow \underbrace{\underbrace{\begin{bmatrix} - & q_1 & - \\ \vdots & \cdots & \vdots \\ - & q_n & - \end{bmatrix}}_{d_k} \left.\begin{bmatrix} | & \vdots & | \\ k_1 & \vdots & k_m \\ | & \vdots & | \end{bmatrix}\right\} d_k \begin{bmatrix} - & v_1 & - \\ \vdots & \cdots & \vdots \\ - & v_m & - \end{bmatrix}}_{A(Q,K,V)}$$

Vaswani, e.t, al, (2017), "Attention Is All You Need" (Google)

problem:

- as d_k is larger variance of $q^\top k$ increases:
- as a result, some dot product values gets very large, with $\exp(.)$, softmax $\mathbf{p}$ gets peaky!
- remember derivative of cross-entropy between softmax $\mathbf{p}$ and $\mathbf{y}$ is:

$$\mathcal{C}(\mathbf{z}) = -\sum_{k=1}^{K} y_k \left[\log\left(p_k\right)\right] = -\sum_{k=1}^{K} y_k \left[\log\left(\frac{\exp^{z_k}}{\sum_l \exp^{z_l}}\right)\right]$$

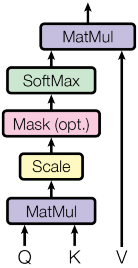
$$\Longrightarrow \frac{\mathcal{C}(\mathbf{z})}{\partial \mathbf{z}} = (\mathbf{p} - \mathbf{y})$$

- with a peaky softmax, lots of element in gradient vector $\frac{\mathcal{C}(\mathbf{z})}{\partial \mathbf{z}}$ are zero!

solution:

- scale by length of d_k:

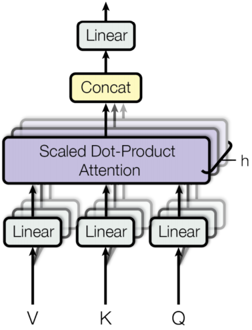
$$A(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$$



- "mask (opt.)" is only used at decoder during training

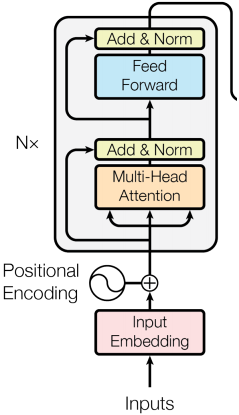
Self-attention and Multi-head attention

- **generic** input vectors could be (Q, K, V)
- when allow some equal signs, for example, $Q = K = V$, it achieves **self-attention**!
- **even better** can we let words to have multiple ways of interactions with each other?
- **Multi-head attention!**
  1. **loop** through $i \in \{1 \dots h\}$, for each i-th iteration:
     - linear transform Q, K, V into several lower dimensional spaces, to obtain
       $$\text{head}_i = (QW_i^q, KW_i^k, VW_i^v)$$
     - each iteration i correspond to one "surface" of the ["linear", "Scaled Dot-Product Attention"] on the diagram
  2. then concatenate to produce output matrix $\mathbf{H}$
     $$\mathbf{H} = [\text{head}_1, \dots, \text{head}_h]$$
  3. finally,
     $$\text{MultiHead}(Q, K, V) = \mathbf{H}W^o$$



Attention is all you need: encoder

- for **encoder** each block, use same ($Q = K = V$) from previous layer
- all the goodies: ReLU + ResNet+ NN + LayerNorm
- blocks repeated $N\times$ times
- unlike RNN, using attention loose the ordering of the words in encoder, therefore, **explicit position encoding** is required



Attention is all you need: decoder

- again, all the goodies: ReLU + ResNet+ NN + LayerNorm
- during training: masked decoder self-attention on previously generated outputs
- Encoder-Decoder Attention: Q come from previous decoder layer and K and V come from output of encoder
- the above is similar to **seq2seq with attention** $Q = h$ from decoder, $K = V = z$ from encoder
- blocks repeated N times