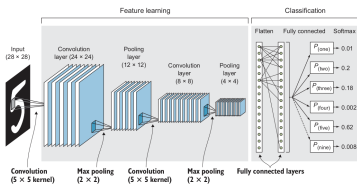# Sequence Modeling: Recurrent Neural Networks

Ju Sun

Computer Science & Engineering

University of Minnesota, Twin Cities

November 15, 2022

Recap: CNNs

(Credit: [Elgendy, 2020])

– (neuro-inspired) locality and weight sharing $\implies$ reduced complexity (than FCNN)

– conv + pooling $\implies$ (approx.) translation/deformation invariance (part of the learning can be avoided; see **scattering transform**

[Bruna and Mallat, 2013, Mallat, 2016, Zarka et al., 2019])

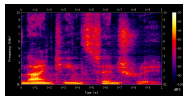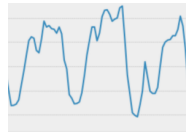CNNs are not only for images: ideal for tensors where **locality** matters

image

video

audio (spectrogram)

time series

what do you think of these cases?

Model sequences

... where directions matter

temporal sequences

disease prognosis   event analysis/video generation   speech to text

lexical sequences—most tasks in Natural Language Processing (NLP)

– machine translation, e.g., English ⇆ Chinese

– typing/writing prediction (smart compose)

– semantic classification

Outline

Basic setup
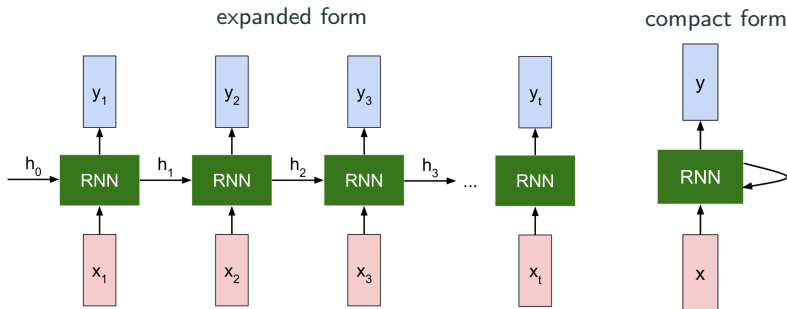
A sequence: $x_0 \to x_1 \to x_2 \to \ldots x_{n-1}$

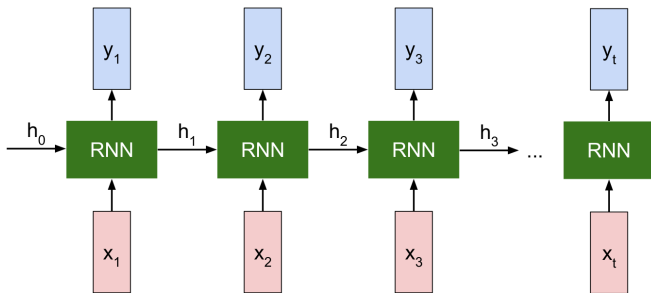A **state-space** model: h denotes the state, and state transition modeled by the recurrence formula

$$h_t = f_W(h_{t-1}, x_t)$$

with optional output

$$y_t = g_V(h_t)$$

expanded form   compact form

(Credit: Stanford CS231N)

A simple (vanilla) RNN

(Credit: Stanford CS231N)

$$h_t = \tanh\left(W_h h_{t-1} + W_x x_t\right)$$

$$y_t = V_y h_t$$

W_h, W_x and V_y are shared across the sequence

A first example: language modeling

– **language modeling** is the task of predicting future words

    ... The vaccine is effective, and COVID-19 will be _.

– applications: typing prediction (smart compose), machine translation, etc
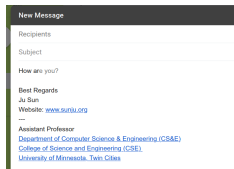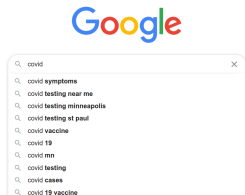
– (traditional) statistical formalism: given a sequence of words
$\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(t)}$, compute

$$\mathbb{P}\left[\boldsymbol{x}^{(t+1)} \mid \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)}\right]$$

where $\boldsymbol{x}^{(t+1)}$ can be any word from a vocabulary $\{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_N\}$, or
sometimes given some text $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}$

$$\mathbb{P}\left[\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}\right] = \prod_{t=1}^{T} \mathbb{P}\left[\boldsymbol{x}^{(t)} \mid \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(1)}\right]$$
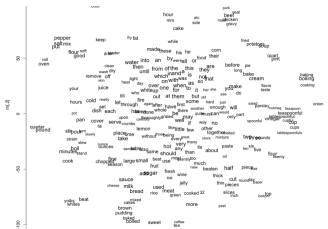
Modern neural language modeling—1

Representing words: word embedding

- one hot encoding

  $$I \mapsto [1, 0, 0, 0, \ldots], \text{you} \mapsto [0, 1, 0, 0, \ldots], \text{we} \mapsto [0, 0, 1, 0, \ldots], \ldots$$
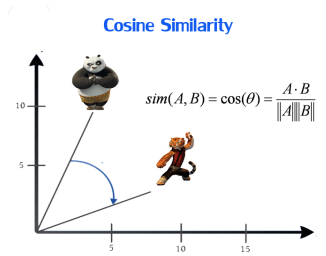
- word-to-vector embedding: map words into **dense** vectors so that certain arithmetic operations are consistent with semantics

(Credit: https://www.adityathakker.com/
introduction-to-word2vec-how-it-works/)

Cosine Similarity

$$sim(A, B) = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$

(Credit: https://towardsdatascience.com/
introduction-to-word-embedding-and-word2vec-652d0c206

e.g., word2vec, BloVe, ELMo

RNN modeling

output distribution

$$\hat{\boldsymbol{y}}^{(t)} = \mathrm{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$$

hidden states

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

$$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$$

words / one-hot vectors

$$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$$

(Credit: adapted from Stanford CS224N)

Training the RNN 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 → $\hat{\boldsymbol{y}}^{(1)}$ $\hat{\boldsymbol{y}}^{(2)}$ $\hat{\boldsymbol{y}}^{(3)}$ $\hat{\boldsymbol{y}}^{(4)}$

$\boldsymbol{h}^{(0)}$ $\boldsymbol{U}$ $\boldsymbol{h}^{(1)}$ $\boldsymbol{U}$ $\boldsymbol{h}^{(2)}$ $\boldsymbol{U}$ $\boldsymbol{h}^{(3)}$ $\boldsymbol{U}$ $\boldsymbol{h}^{(4)}$

$\boldsymbol{W}_h$ $\boldsymbol{W}_h$ $\boldsymbol{W}_h$ $\boldsymbol{W}_h$ $\boldsymbol{W}_h$ ...

$\boldsymbol{W}_e$ $\boldsymbol{W}_e$ $\boldsymbol{W}_e$ $\boldsymbol{W}_e$

$\boldsymbol{e}^{(1)}$ $\boldsymbol{e}^{(2)}$ $\boldsymbol{e}^{(3)}$ $\boldsymbol{e}^{(4)}$

$\boldsymbol{E}$ $\boldsymbol{E}$ $\boldsymbol{E}$ $\boldsymbol{E}$

Corpus → the $\boldsymbol{x}^{(1)}$ students $\boldsymbol{x}^{(2)}$ opened $\boldsymbol{x}^{(3)}$ their $\boldsymbol{x}^{(4)}$ exams ...
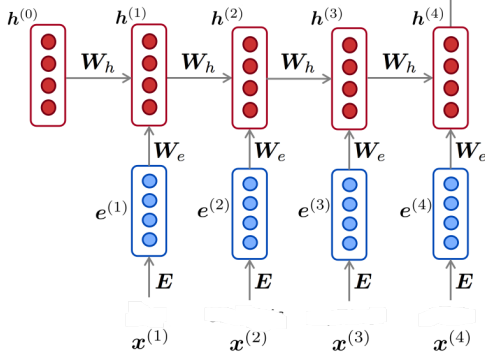
(Credit: Stanford CS224N)

Modern neural language modeling—4

The whole training pipeline

– Step 1: collect a large corpus of text, i.e., a long sequence
$\mathcal{T} = \boldsymbol{x}^{(1)} \to \cdots \to \boldsymbol{x}^{(T)}$ (e.g., a sentence, a document, etc)

– Step 2: feed $\mathcal{T}$ into the model, and compute output distribution $\widehat{\boldsymbol{y}}^{(t)}$ for each t

– Step 3: define loss, e.g., cross entropy between $\widehat{\boldsymbol{y}}^{(t)}$ and $\boldsymbol{y}^{(t)}$ (one-hot encoding of $\boldsymbol{x}^{(t+1)}$)

$$J^{(t)}\left(\boldsymbol{\theta}\right) = -\sum_{\boldsymbol{w} \in \mathcal{V}} \boldsymbol{y}_{\boldsymbol{w}}^{(t)} \log \widehat{\boldsymbol{y}}_{\boldsymbol{w}}^{(t)} = -\log \widehat{\boldsymbol{y}}_{\boldsymbol{x}^{(t+1)}}^{(t)}$$

– Step 4: gather and average all losses:

$$J\left(\boldsymbol{\theta}\right) = \frac{1}{T} \sum_{t=1}^{T} J^{(t)}\left(\boldsymbol{\theta}\right)$$

– Step 5: optimization: SGD (are the summation terms iid in the objective?), etc
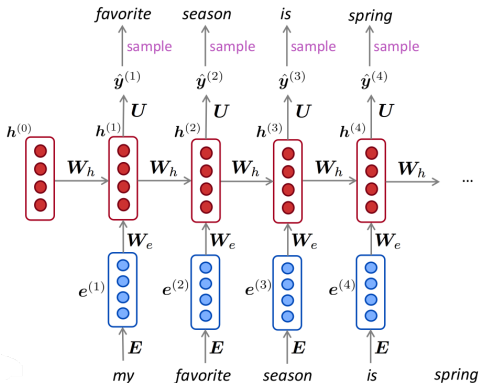
(Credit: Stanford CS224N)

starting from $h^{(0)}$ and `my`, repeat:

– compute $y^{(t)}$ and sample a word from the distribution

– feed the word as input to the next step

Outline

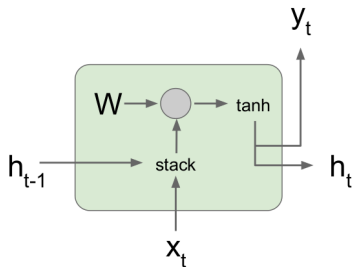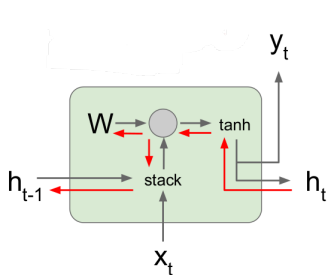# How to compute gradients?

computational graph
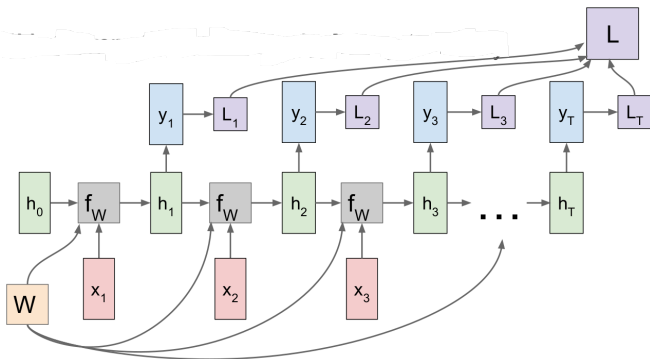
(Credit: Stanford CS231N)

– acyclic directed graph $\implies$ auto differentiation can be applied

– W is shared across all steps!

Look into the gradient

(Credit: Stanford CS231N)

(Credit: Stanford CS231N)

$$\boldsymbol{h}_t = \tanh\left(\boldsymbol{W_h}\boldsymbol{h}_{t-1} + \boldsymbol{W_x}\boldsymbol{x}_t\right)$$

$$= \tanh\left(\boldsymbol{W}\begin{bmatrix}\boldsymbol{h}_{t-1} \\ \boldsymbol{x}_t\end{bmatrix}\right)$$

where $\boldsymbol{W} = \begin{bmatrix}\boldsymbol{W_h} \ \boldsymbol{W_x}\end{bmatrix}$

$$\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}} = \mathrm{diag}\left(\tanh'\left(\boldsymbol{W_h}\boldsymbol{h}_{t-1} + \boldsymbol{W_x}\boldsymbol{x}_t\right)\right)\boldsymbol{W_h}$$
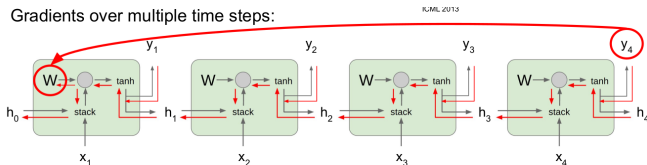
where

$$\tanh'(x) = 1 - \tanh^2(x)$$

Look into the gradient

$L = \sum_{t=1}^{T} L_t \implies$ total gradient: $\frac{\partial L}{\partial \boldsymbol{W}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \boldsymbol{W}}$

$$\boldsymbol{h}_t = \tanh\left(\boldsymbol{W_h h}_{t-1} + \boldsymbol{W_x x}_t\right) = \tanh\left(\boldsymbol{W} \begin{bmatrix} \boldsymbol{h}_{t-1} \\ \boldsymbol{x}_t \end{bmatrix}\right)$$

$$\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}} = \mathrm{diag}\left(\tanh'\left(\boldsymbol{W_h h}_{t-1} + \boldsymbol{W_x x}_t\right)\right) \boldsymbol{W}_h$$

Gradients over multiple time steps:

(Credit: Stanford CS231N)

$$\frac{\partial L_t}{\partial W} = \frac{\partial L_t}{\partial \boldsymbol{h}_t} \frac{\partial \boldsymbol{h}_t}{\partial h_{t-1}} \cdots \frac{\partial \boldsymbol{h}_1}{\boldsymbol{W}} = \frac{\partial L_t}{\partial \boldsymbol{h}_t} \left(\prod_{k=2}^{t} \frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{h}_{k-1}}\right) \frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{W}}$$

$$= \frac{\partial L_t}{\partial \boldsymbol{h}_t} \left(\prod_{k=2}^{t} \mathrm{diag}\left(\tanh'\left(\boldsymbol{W_h h}_{k-1} + \boldsymbol{W_x x}_k\right)\right) \boldsymbol{W}_h\right) \frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{W}}$$

What's wrong with the gradient?

consider $\prod_{k=2}^{t} \operatorname{diag}\left(\tanh'\left(\boldsymbol{W_h h}_{k-1} + \boldsymbol{W_x x}_k\right)\right) \boldsymbol{W_h}$

- for intuition, consider **identity** activation first, i.e., $\prod_{k=2}^{t} \boldsymbol{W_h} = \boldsymbol{W_h}^{t-1}$.
  $\|\boldsymbol{W_h}^{t-1}\|$, i.e., the largest singular value of $\boldsymbol{W_h}^{t-1}$, scales as $\|\boldsymbol{W_h}\|^{t-1}$
    * when $\|\boldsymbol{W_k}\| > 1$, gradient **explodes** if t large
    * when $\|\boldsymbol{W_k}\| < 1$, gradient **vanishes** if t large
- what happens with the $\tanh$ activation?
    * $\tanh'(x) = 1 - \tanh^2(x) \leq 1$—effectively always smaller
    * we have

$$\left\| \prod_{k=2}^{t} \operatorname{diag}\left(\tanh'\left(\boldsymbol{W_h h}_{k-1} + \boldsymbol{W_x x}_k\right)\right) \boldsymbol{W_h} \right\|$$

$$\leq \prod_{k=2}^{t} \left\| \operatorname{diag}\left(\tanh'\left(\boldsymbol{W_h h}_{k-1} + \boldsymbol{W_x x}_k\right)\right) \right\| \|\boldsymbol{W_h}\|$$

$$\leq \underbrace{\prod_{k=2}^{t} \left\| \operatorname{diag}\left(\tanh'\left(\boldsymbol{W_h h}_{k-1} + \boldsymbol{W_x x}_k\right)\right) \right\| \|\boldsymbol{W_h}\|^{t-1}}_{\text{product of many numbers} < 1 \text{ when } t \text{ large}}$$

gradient vanishing is more common

Gradient clipping

When the gradient is too large (exploding), rescale (i.e., clip) it. Let $\boldsymbol{g}$ be the gradient and $\xi > 0$ be a threshold
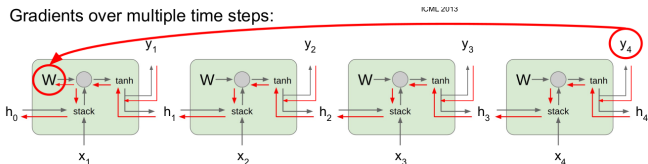
$$\widehat{\boldsymbol{g}} = \xi \frac{\boldsymbol{g}}{\|\boldsymbol{g}\|}$$

Without clipping

With clipping
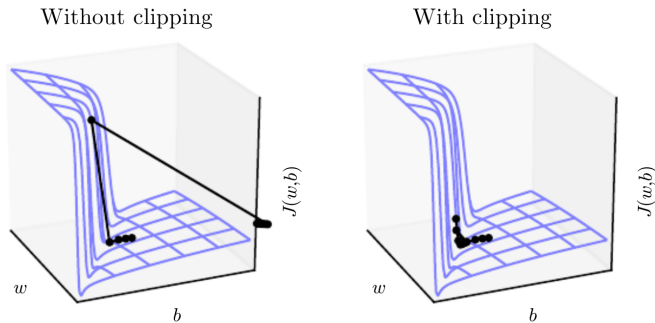
(Credit: [Goodfellow et al., 2017])

Gradients over multiple time steps:

(Credit: Stanford CS231N)

- **gradient vanishing**: $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_1}$ is (exponentially) small when t is large
  $\implies$ earlier states have little impact on latter states, i.e., memory is short

- but we hope to use RNN to encode reasonably long-term historical/contextual information

Solution? Modify the architecture

Outline

key idea: introduce a cell state c to explicitly store history, besides the hidden state h

Long Short Term Memory (LSTM)
[Hochreiter et al., 1997]
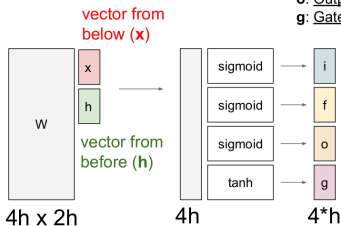
i: Input gate, whether to write to cell
f: Forget gate, Whether to erase cell
o: Output gate, How much to reveal cell
g: Gate gate (?), How much to write to cell

vector from below ($\mathbf{x}$)

vector from before ($\mathbf{h}$)

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$
$$h_t = o \odot \tanh(c_t)$$

4h x 2h   4h   4*h

(Credit: Stanford CS231N)

where σ denotes sigmoid

f: memory controller and i: writing controller and o: output controller learned *independently*

simplified version of LSTM ...

i: <u>Input gate</u>, whether to write to cell
f: <u>Forget gate</u>, Whether to erase cell
o: <u>Output gate</u>, How much to reveal cell
g: <u>Gate gate</u> (?), How much to write to cell

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$
$$h_t = o \odot \tanh(c_t)$$

f: memory controller and i: writing controller and o: output controller learned *independently* long-term memory when $f = 1$

GRU: no cell state

u: **update gate**, control state update

r: **reset gate**, control how previous state affects new content

g: new content

$$\begin{bmatrix} u \\ r \\ g \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \end{bmatrix} \left(W \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} \right)$$

$$g = \tanh \left(W_h \left(r \odot h_{t-1} \right) + W_x x_t + b_g \right)$$

$$h_t = u \odot h_{t-1} + (1 - u) \odot g$$

f, i, o are merged
long-term memory when $u = 1$ and $r = 1$

LSTM is more flexible and powerful but less efficient in speed

Do they save the vanishing gradient?
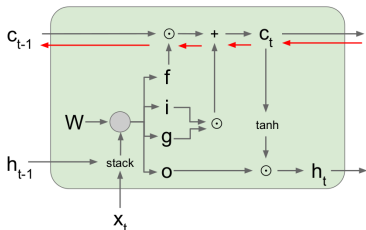
(Credit: Stanford CS231N)

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$
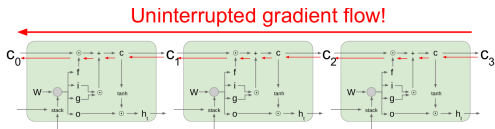$$h_t = o \odot \tanh(c_t)$$

(Credit: Stanford CS231N)

$$\frac{\partial \boldsymbol{c_t}}{\partial \boldsymbol{c_{t-1}}} = \operatorname{diag}\left(\boldsymbol{f}\right) \text{ — no multiplication by } \boldsymbol{W}$$

Uninterrupted gradient flow!
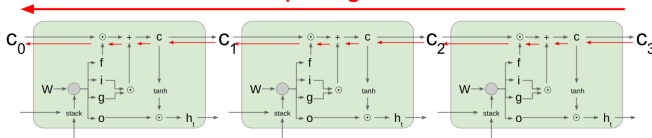
(Credit: Stanford CS231N)

Look familiar?

Uninterrupted gradient flow!

(Credit: Stanford CS231N)
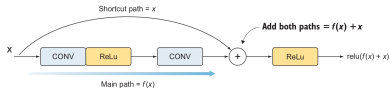
a residual block (Credit: [Elgendy, 2020])
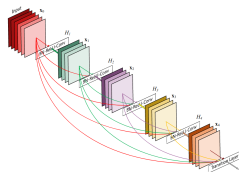
Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

(Credit: [Huang et al., 2016])

They are all skip-connections! Similarly for GRU.

- skip connections allow better modeling of long-distance dependency
- but no guarantee of solving the grad vanishing/explosion problem

Do we need to modify the architecture?

problem: $\boldsymbol{W}_h$ can have singular values other than 1

solution: ensure all singular values are $1 \Longrightarrow \boldsymbol{W}_h$ is orthogonal

$$\min_{\boldsymbol{W}_h, \boldsymbol{W}_x} L\left(\boldsymbol{W}\right), \, \text{s.\,t.} \, \boldsymbol{W}_h \, \textbf{orthogonal}, \text{i.e.}, \boldsymbol{W}_h^\intercal \boldsymbol{W}_h = \boldsymbol{I}$$

Good empirical performance, but cost is high for large-scale problems. See, e.g., [Arjovsky et al., 2016, Lezcano-Casado and Martínez-Rubio, 2019]
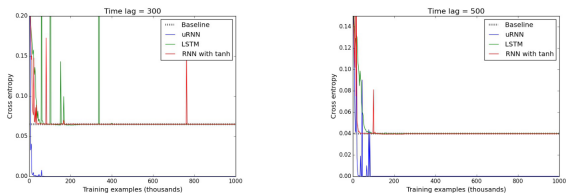
Figure 1. Results of the copying memory problem for time lags of $100, 200, 300, 500$. The LSTM is able to beat the baseline only for 100 times steps. Conversely the uRNN is able to completely solve each time length in very few training iterations, without getting stuck at the baseline.

(Credit: [Arjovsky et al., 2016])

(see demo based on PyGRANSO

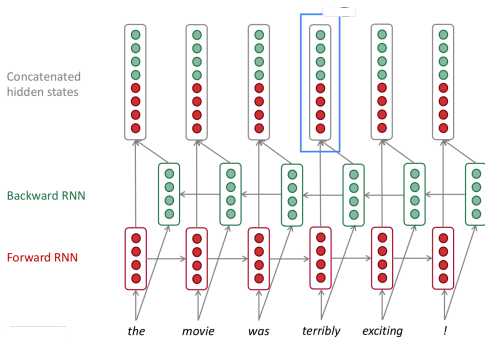https://ncvx.org/examples/D3_orthogonal_rnn.html)

Context is important!

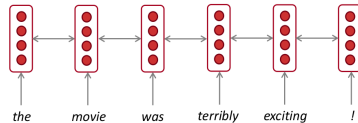sentiment classification
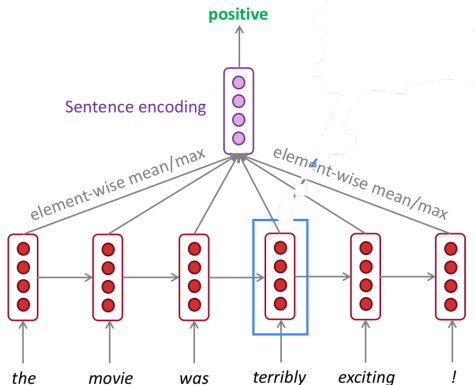
(Credit: adapted from Stanford CS224N)

– the state vectors are **contextual representation** of the input word
– but to tell sentiment, "exciting", which is to the right of "terribly" is crucial

Concatenated hidden states

Backward RNN

Forward RNN

the   movie   was   terribly   exciting   !

(Credit: Stanford CS224N)

simplified schematic

the   movie   was   terribly   exciting   !

(Credit: Stanford CS224N)

– both left and right contexts are now encoded!

– applicable when the full sequence is available

Deep RNNs

hidden state h can be thought of representation, and so far we only have one layer
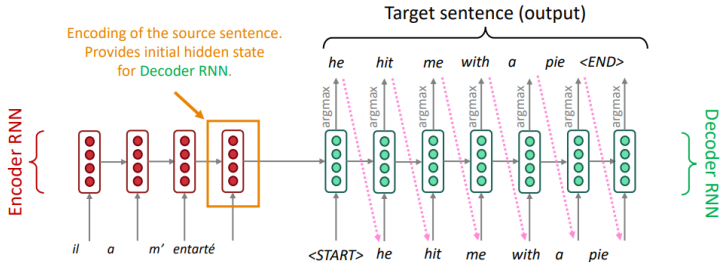
Go deeper for more powerful representation learning!
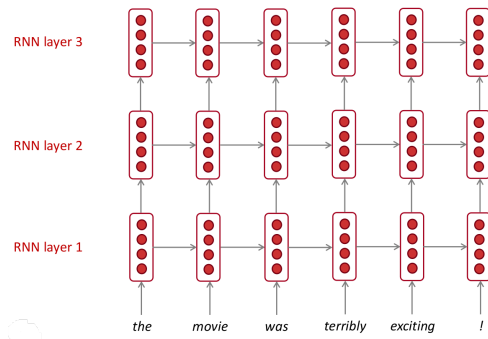
(Credit: Stanford CS231N)

multi-layer RNNs or stacked RNNs. Typically only few layers (much less than that of CNNs)
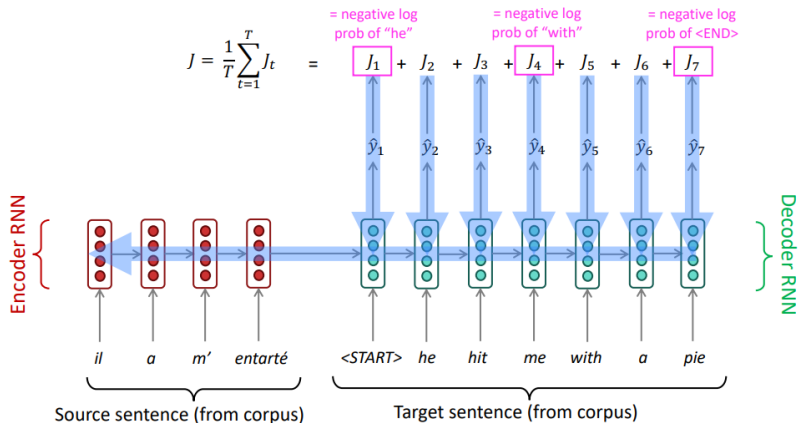
Sequence to sequence models (Seq2Seq)

machine translation, image-to-text, speech-to-text, etc
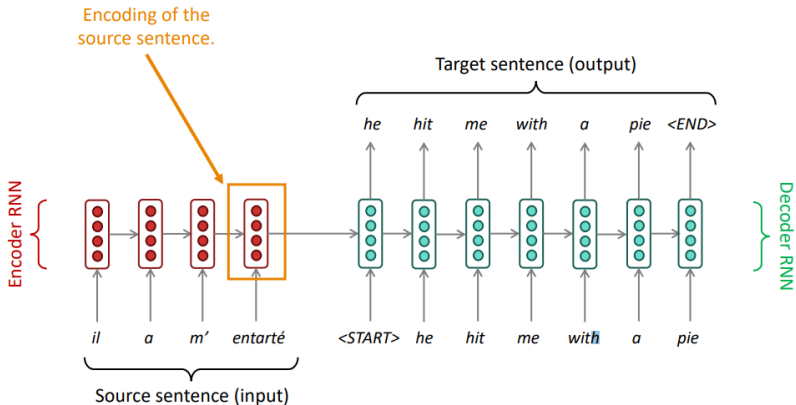

(Credit: Stanford CS231N)

– Falls under encoder-decoder models

– Encoder RNN translate source into an encoding

– Decoder RNN is a language model generates output sentence based on the encoding

(Credit: Stanford CS231N)

(Credit: Stanford CS231N)

Problem: the encoding has to capture all info of the source to be effective

Solution: make each target state dependent on all source states

**Problem**: the encoding has to capture all info of the source to be effective
Solution: make each target state dependent on all source states

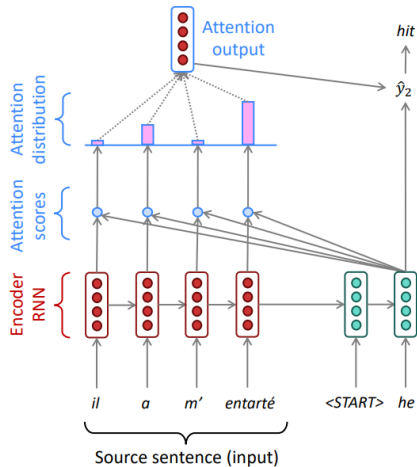Assume source state vectors $s_1, \ldots, s_N \in \mathbb{R}^h$, and current target state vector t_i

- Idea 1: concatenate, i.e., form $[s_1; \ldots; s_N; t_i]$ as the new state vector for the current target step. What's wrong?

- Idea 2: sum and concatenate, i.e., $[\frac{1}{N} \sum_{j=1}^{N} s_j; t_i]$. What's wrong?

- Idea 3: weighted sum and concatenate, i.e., $[\sum_{j=1}^{N} w_j s_j; t_i]$

  * What weights? Emphasize those most relevant to t_i
  * Set $w_j = \mathrm{similarity}(s_j, t_i)$: **attention mechanism**

  Attention is about measuring (nonlinear) correlation/similarity
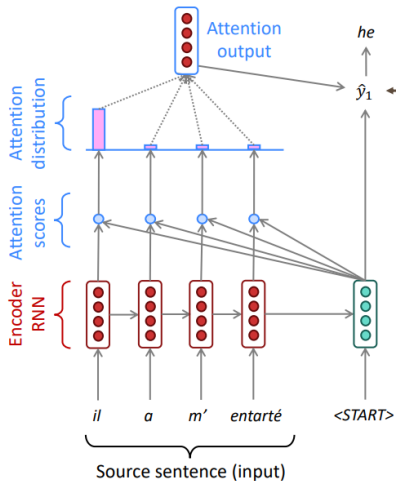
(Credit: Stanford CS231N)

(Credit: Stanford CS231N)

Attention in a nutshell

Assume source vectors $s_1, \ldots, s_N \in \mathbb{R}^h$, and target vector t, to obtain selective summary (e.g., weighted summation) of $s_1, \ldots, s_N \in \mathbb{R}^h$

$$\sum_{j=1}^{N} w_j s_j \quad \text{where } w_j = \text{similarity}(s_j, t)$$

Many possibilities:

- dot-product attention: $w_j = \langle s_j, t \rangle$ (Is is better to normalize this or rescale it by the dimension factor?)
- multiplicative attention: $w_j = \langle s_j, W t \rangle$
- "additive attention": $w_j = v^\intercal \sigma (W_1 s_j + W_2 t)$

Afterward, pass the whole weight vector $[w_1, \ldots, w_N]$ through softmax to turn it into a valid distribution

Attention is not only for Seq2Seq or RNNs, it is to: calculate a weighted sum of a bunch of (source) vectors, with the weights depedent on a target/query vector
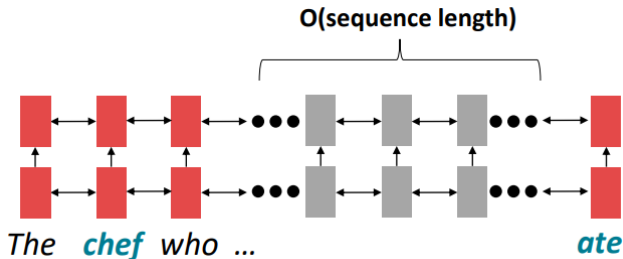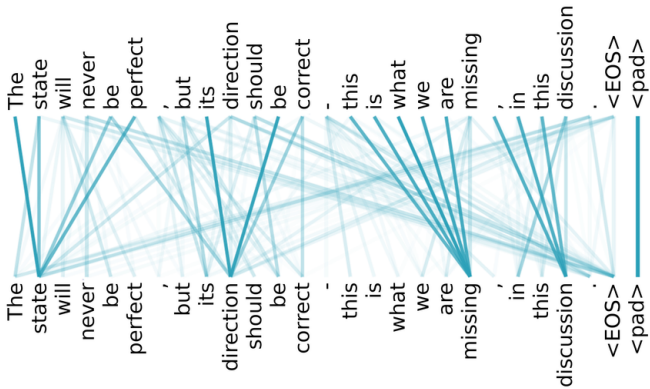
(Credit: Stanford CS231N)

– **linear interaction distance**: challenging to encode long-range dependencies, even within the same sequence

– **resistance to parallelization**: state generation is inherently sequential—problematic for very long sequences
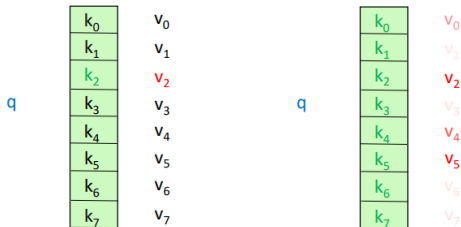
Solution—self-attention

build connections between each other: each state vector depends on all the rest

- $O(1)$ interaction distance
- two state vectors (query, key) to allow parallelization

Self-attention: a closer look

- Each word now encoded as (query, key, value) triple
- For an input $\boldsymbol{x}_i$, we have:

$$\boldsymbol{q}_i = (\boldsymbol{W}^Q)^\intercal \boldsymbol{x}_i, \quad \boldsymbol{k}_i = (\boldsymbol{W}^K)^\intercal \boldsymbol{x}_i, \quad \boldsymbol{v}_i = (\boldsymbol{W}^V)^\intercal \boldsymbol{x}_i$$

- Calculate attention scores between query and all keys: $e_{ij} = \langle \boldsymbol{q}_i, \boldsymbol{k}_j \rangle$
- softmax normalization $w_{ij} = \exp(e_{ij}) / \sum_k \exp(e_{ik})$
- output the weighted sum of values $\sum_j w_{ij} \boldsymbol{v}_j$

Self-attention in matrix notation

Assume X collects all input words, each one a row:

- Compute queries, keys, and values

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$
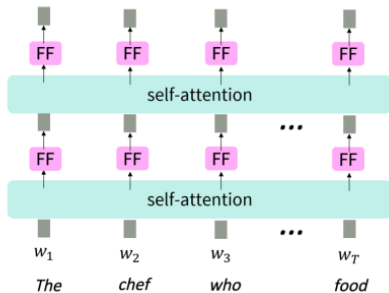
- Calculate attention scores between query and all keys: $E = QK^\mathsf{T}$
- softmax normalization $W = \text{softmax}(E)$
- output the weighted sum of values AV

$$\text{output} = \text{softmax}(QK^\mathsf{T})V$$

Equation for Feed Forward Layer

$$m_i = MLP(\text{output}_i)$$
$$= W_2 * \text{ReLU}(W_1 \times \text{output}_i + b_1) + b_2$$
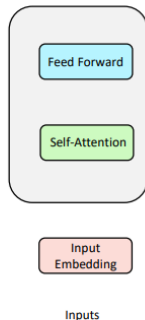
First step toward the Transformer!

Suggested reading

- Stanford CS224N
  http://web.stanford.edu/class/cs224n/index.html#schedule

- Understanding LSTM Networks
  http://colah.github.io/posts/2015-08-Understanding-LSTMs/

- A Guide to the Encoder-Decoder Model and the Attention Mechanism
  https://medium.com/better-programming/
  a-guide-on-the-encoder-decoder-model-and-the-attention-mechanism-

- Attention is all you need: Discovering the Transformer paper
  https://towardsdatascience.com/
  attention-is-all-you-need-discovering-the-transformer-paper-73e5ff

[Arjovsky et al., 2016] Arjovsky, M., Shah, A., and Bengio, Y. (2016). **Unitary evolution recurrent neural networks.** In *International Conference on Machine Learning*, pages 1120–1128.

[Bruna and Mallat, 2013] Bruna, J. and Mallat, S. (2013). **Invariant scattering convolution networks.** *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1872–1886.

[Elgendy, 2020] Elgendy, M. (2020). **Deep Learning for Vision Systems.** MANNING PUBN.

[Goodfellow et al., 2017] Goodfellow, I., Bengio, Y., and Courville, A. (2017). **Deep Learning.** The MIT Press.

[Huang et al., 2016] Huang, G., Liu, Z., van der Maaten, L., and Weinberger, K. Q. (2016). **Densely connected convolutional networks.** *arXiv:1608.06993*.

[Lezcano-Casado and Martínez-Rubio, 2019] Lezcano-Casado, M. and Martínez-Rubio, D. (2019). **Cheap orthogonal constraints in neural networks: A simple parametrization of the orthogonal and unitary group.** *arXiv1901.08428*.

[Mallat, 2016] Mallat, S. (2016). **Understanding deep convolutional networks.**
*Philosophical Transactions of the Royal Society A: Mathematical, Physical and
Engineering Sciences*, 374(2065):20150203.

[Zarka et al., 2019] Zarka, J., Thiry, L., Angles, T., and Mallat, S. (2019). **Deep
network classification by scattering and homotopy dictionary learning.**
arXiv:1910.03561.