RNNs: Other Applications, LSTMs

Pawan Goyal

CSE, IIT Kharagpur

CS60010

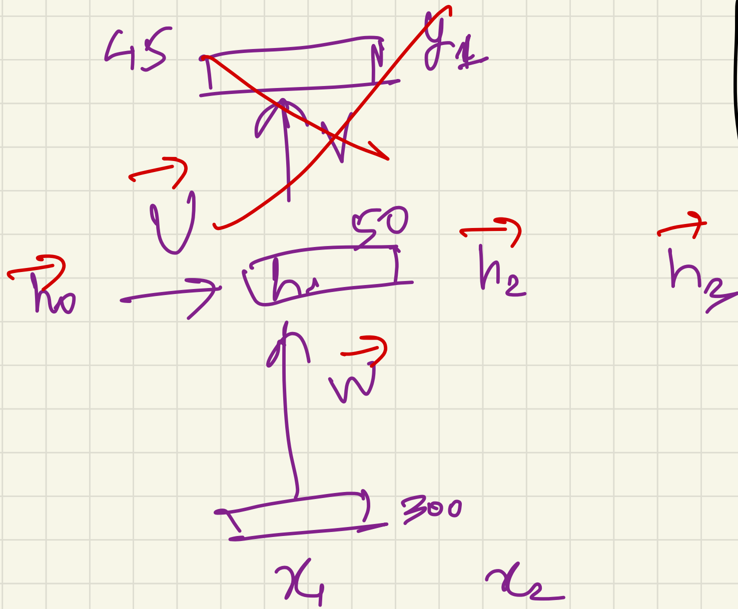
300-d word embeddings

hidden – 50 dim

# of parameters?

Using bi-RNN for POS task

45 POS tags



$\vec{U}$: 50×50

$\vec{W}$: 300×50

V : 50×45

$\overleftarrow{U}$: 50×50

$\overleftarrow{W}$: 300×50

$h_1 = [\vec{h}_1 ; \overleftarrow{h}_1]$

$\downarrow V$

V : 2×50×45

5-6 weeks of classes

25% - MidSem
40% - End-sem

GAN
VAE
Diffusion

Transformers
Pretraining (NWP)
LLM

Assignment +
1 Mini Project
Group - 3 members
1 month
+ 1 Quiz
+ 2 Quizzes
+ Attendance

→ 35%

10
20-25
5
5
5-10

Using Bidirectional RNNs for Sequence Classification

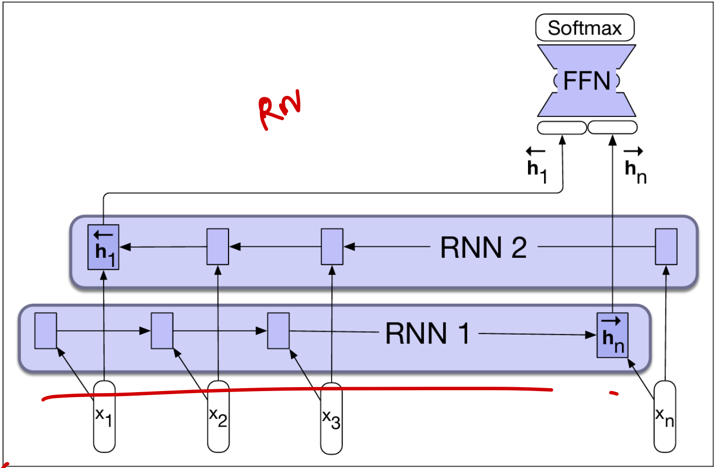
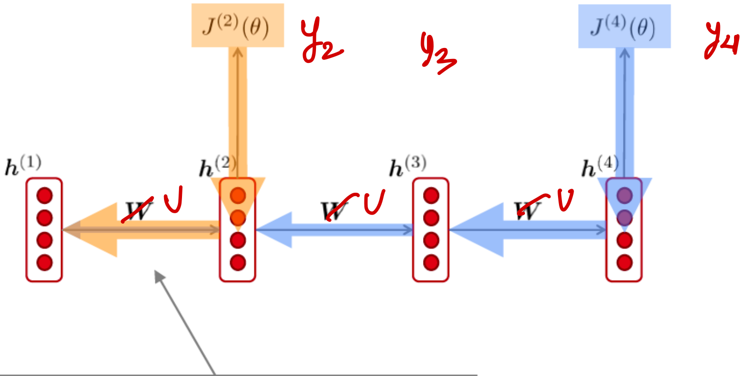


Figure 9.12 A bidirectional RNN for sequence classification. The final hidden units from the forward and backward passes are combined to represent the entire sequence. This combined representation serves as input to the subsequent classifier.

Need for better units: Vanishing Gradient



Gradient signal from faraway is lost because it's much smaller than gradient signal from close-by.

So model weights are only updated only with respect to near effects, not long-term effects.

Effect of vanishing gradient on RNN LM

- **LM task:** *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 can't learn this dependency
  - So the model is unable to predict similar long-distance dependencies at test time

Effect of vanishing gradient on RNN LM

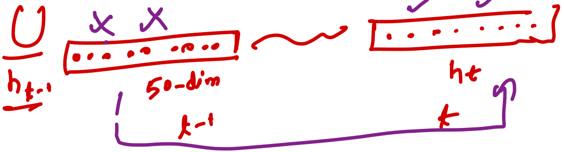
- **LM task:** *The writer of the books* ___ → *is* / *are*
- **Correct answer**: *The writer of the books* *is* *planning a sequel*
- **Syntactic recency:** *The writer of the books is* Syntax (correct)
- **Sequential recency:** *The writer of the books are* (incorrect)
- Due to vanishing gradient, RNN-LMs are better at learning from sequential recency than syntactic recency, so they make this type of error more often than we'd like [Linzen et al 2016]

How to fix vanishing gradient problem?

- The main problem is that it is too difficult for the RNN to learn to preserve information over many timesteps.
- In a vanilla RNN, the hidden state is constantly being rewritten

$$h^{(t)} = (1-\alpha)\tanh(U\underbrace{h^{(t-1)}}_{\text{prev.}} + W\underbrace{x^{(t)}}_{\text{new}}) + \alpha h^{(t-1)}$$

- How about better RNN units?

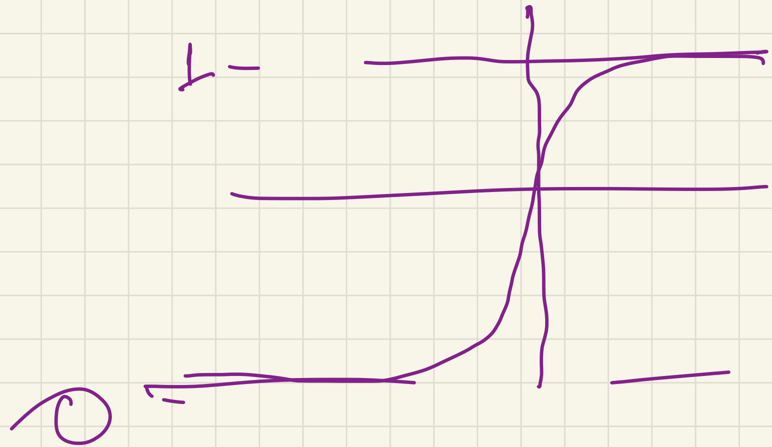


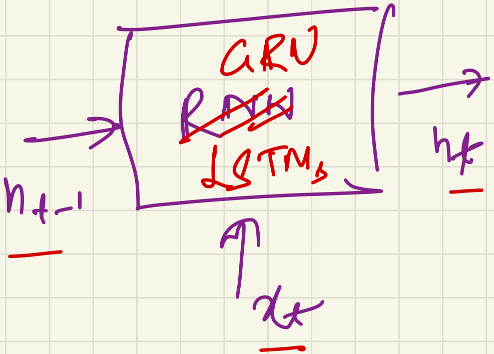
gate 1 0 0 1 0 1 1 1

50

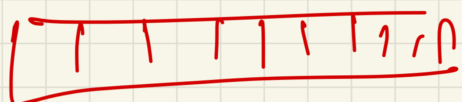
Neural Gate

Sigmoid





$$i_t = \sigma(U_i h_{t-1} + W_i x_t)$$



makes learnable params

Using Gates for better RNN units

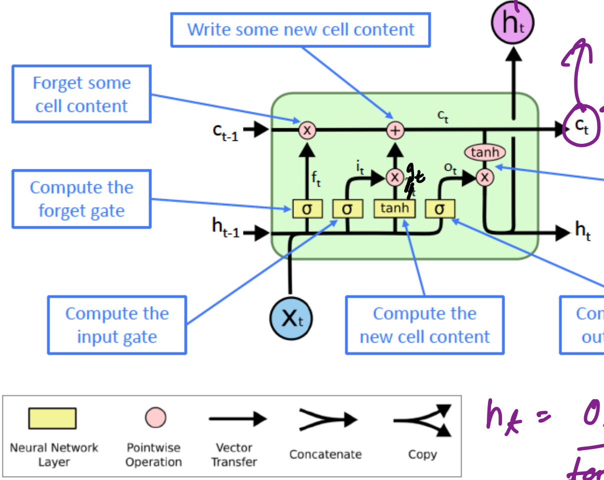
- The gates are also vectors
- On each timestep, each element of the gates can be open (1), close (0) or somewhere in-between.
- The gates are dynamic: their value is computed based on the current context.

Two famous architectures

GRUs, LSTMs

Long Short Term Memory (LSTM)

You can think of the LSTM equations visually like this:



y_t V

y_t

$h_t \rightarrow h_{t+1}$

context

x_t h_t c_t

context vector

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

cell-state

c_{t-1} $\qquad h_{t-1}$ $\qquad x_t$

$$g_t = \tanh(W_g x_t + U_g h_{t-1})$$

$$f_t \odot c_{t-1} + i_t \odot g_t$$

$g_t \rightarrow$ new ~~state~~ content (like RNN)

$$g_t = \boxed{0.25 \mid -0.2 \mid 0.3}$$

$$i_t = \boxed{0 \mid 0 \mid 1}$$

$$f_t = \sigma(W_f x_t + U_f h_{t-1})$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1})$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1})$$

LSTM: More Details

- For context management, an explicit context layer is added to the architecture
- It makes use of specialized neural units (gates) to control the flow of information
- The gates share a common design feature, and choice of sigmoid pushes its output to 0 or 1, thus it works as a binary mask.

LSTM: In Equations

Forget Gate

Controls what is kept vs forgotten from the context

$$\boxed{f_t = \sigma(U_f h_{t-1} + W_f x_t)}$$

Input Gate

Controls what parts of new cell content are written to the context

$$i_t = \sigma(U_i h_{t-1} + W_i x_t)$$

Output Gate

Controls what part of context are output to hidden state

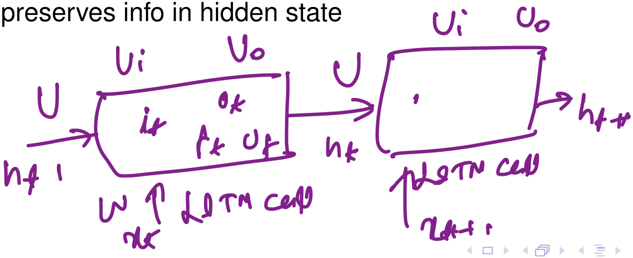
$$o_t = \sigma(U_o h_{t-1} + W_o x_t)$$

New Cell content: $g_t = tanh(U_g h_{t-1} + W_g x_t)$
New Context Vector: $c_t = i_t \odot g_t + f_t \odot c_{t-1}$
New Hidden State: $h_t = o_t \odot tanh(c_t)$

How does LSTM solve vanishing gradients?

The LSTM architecture makes it easier for the RNN to preserve information over many timesteps

- e.g., if the forget gate is set to remember everything on every timestep, then the info in the cell is preserved indefinitely
- By contrast, it is harder for vanilla RNN to learn a recurrent weight matrix U that preserves info in hidden state



Common RNN NLP Architectures

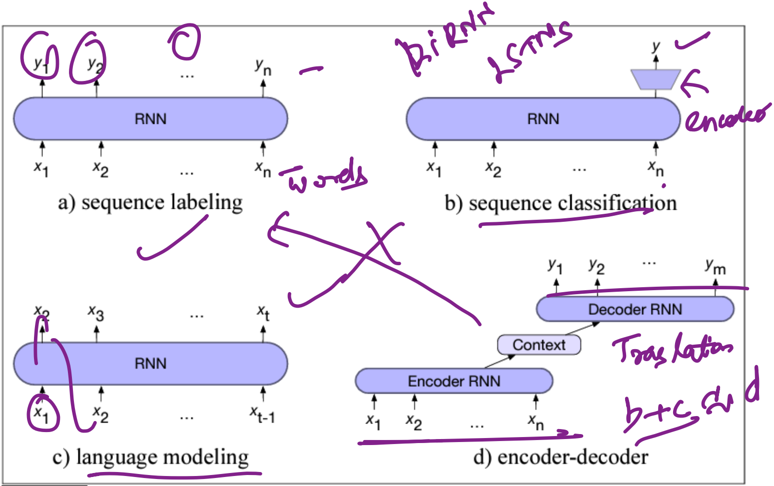


Figure 9.15 Four architectures for NLP tasks. In sequence labeling (POS or named entity tagging) we map each input token x_i to an output token y_i. In sequence classification we map the entire input sequence to a single class. In language modeling we output the next token conditioned on previous tokens. In the encoder model we have two separate RNN models, one of which maps from an input sequence $\mathbf{x}$ to an intermediate representation we call the **context**, and a second of which maps from the context to an output sequence $\mathbf{y}$.

- decoder

Encoder-decoder networks

Also known as sequence-to-sequence networks, and are capable of generating contextually appropriate, arbitrary length output sequences given the input sequence.

Three conceptual components

- An **encoder** that accepts an input sequence $x_{1:n}$ and generates a corresponding sequence of contextualized representations $h_{1:n}$
- A **context** vector, c, which is a function of $h_{1:n}$ and conveys the essence of the input to the decoder
- A **decoder** which accepts c as input and generates an arbitrary length sequence of hidden states $h_{1:m}$ from which the corresponding output states $y_{1:m}$ can be obtained.

Encoder-decoder networks

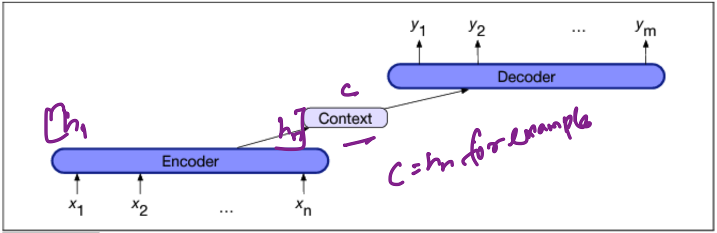


Figure 9.16 The encoder-decoder architecture. The context is a function of the hidden representations of the input, and may be used by the decoder in a variety of ways.

I/p: Seq of Text

o/p Seq of Text

Encoder-decoder networks for translation

$h_0 \to h_1 \to y_1$

<s>

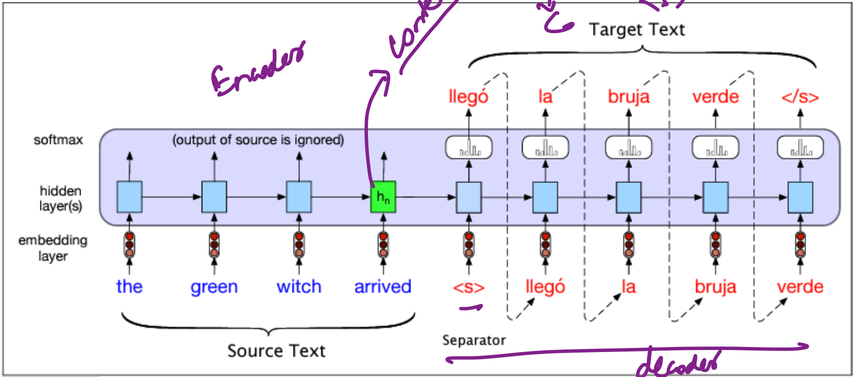


Figure 9.17 Translating a single sentence (inference time) in the basic RNN version of encoder-decoder approach to machine translation. Source and target sentences are concatenated with a separator token in between, and the decoder uses context information from the encoder's last hidden state.

S_E S_F

Training the Encoder-decoder model

zebra

h_1 $\quad$ h_n $\xrightarrow{V}$ h_1^d ... $|V|$ → output Vocab dim

x_1 $\quad$ x_n $\quad$ <S> $\xrightarrow{W}$ I $\quad$ $-\log p(\ )$

End-to-end training

- For MT, the training data typically consists of set of sentences and their translations
- The network is given a source sentence and then a separator token, it is trained auto-regressively to predict the next word
- Teacher forcing is used during training, i.e., the system is forced to use the gold target token from training as the next input x_{t+1}, rather than relying on the last decoder output $\hat{y}_t$

$t+1$

Never at inference time

Training the Encoder-decoder model

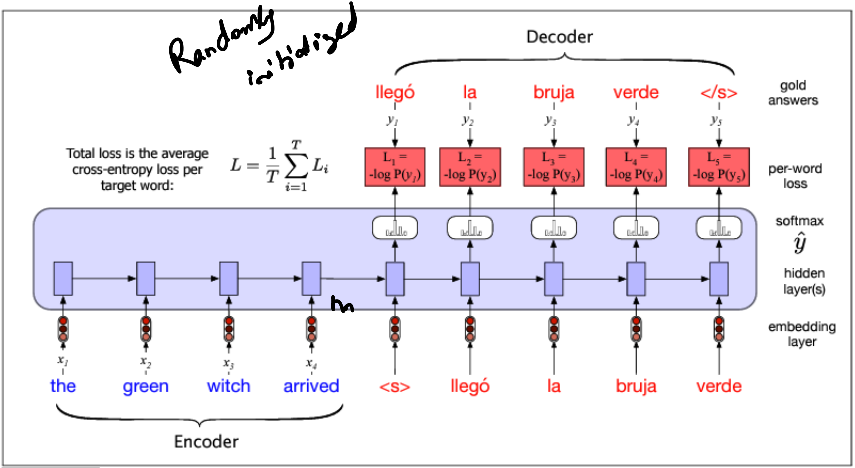


Figure 9.19 Training the basic RNN encoder-decoder approach to machine translation. Note that in the decoder we usually don't propagate the model's softmax outputs $\hat{y}_t$, but use **teacher forcing** to force each input to the correct gold value for training. We compute the softmax output distribution over $\hat{y}$ in the decoder in order to compute the loss at each token, which can then be averaged to compute a loss for the sentence.

Encoder-decoder: Bottleneck

- The context vector, h_n is the hidden state of the last time step of the source text
- It acts as a bottleneck, as it has to represent absolutely everything about the meaning of the source text, as this is the only thing decoder knows about the source text

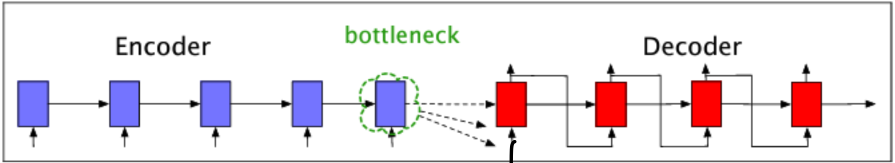


Figure 9.20 Requiring the context c to be only the encoder's final hidden state forces all the information from the entire source sentence to pass through this representational bottleneck.

attention

Encoder-decoder with attention

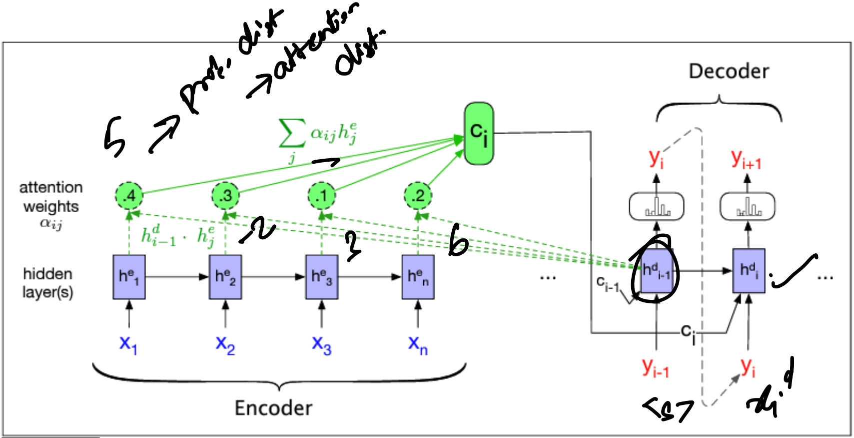


Figure 9.22 A sketch of the encoder-decoder network with attention, focusing on the computation of $\mathbf{c}_i$. The context value $\mathbf{c}_i$ is one of the inputs to the computation of $\mathbf{h}_i^d$. It is computed by taking the weighted sum of all the encoder hidden states, each weighted by their dot product with the prior decoder hidden state $\mathbf{h}_{i-1}^d$.

Attention: In Equations

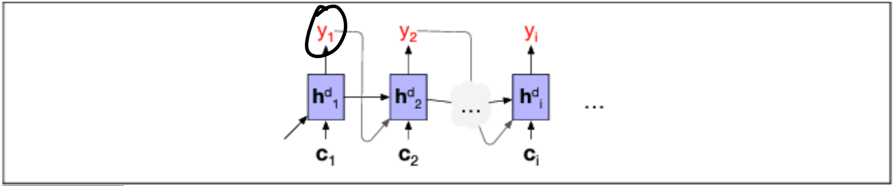


Figure 9.21 The attention mechanism allows each hidden state of the decoder to see a different, dynamic, context, which is a function of all the encoder hidden states.

The context vector c_i is generated anew with each decoding step i

$$h_i^d = g(\hat{y}_{i-1}, h_{i-1}^d, c_i)$$

weighted avg. of encoder states

$W_c c_i$

$h_{i-1}^d + c_i$

Attention: In Equations

Computing c_i

- Compute how much to focus on each encoder state, by seeing how relevant it is to the decoder state captured in h^d_{i-1} – give it a score
- Simplest scoring mechanism is dot-product attention

$$score(h^d_{i-1}, h^e_j) = h^d_{i-1} \cdot h^e_j \rightarrow h^d_{i-1} \, \underset{\uparrow \text{learnable}}{W} \, h^e_j$$

- Normalize these scores using softmax to create a vector of weights $\alpha_{ij} = softmax(score(h^d_{i-1}, h^e_j))$ e
- A fixed-length context vector is created for the current decoder state

$$c_i = \sum_j \alpha_{ij} h^e_j$$

$\tanh(a\, h^d_{i-1} + b\, h^e_j)$

Attention is quite helpful

Attention improves NMT performance

It is useful to allow decoder to focus on certain parts of the source

Attention helps with the long-term dependency problem

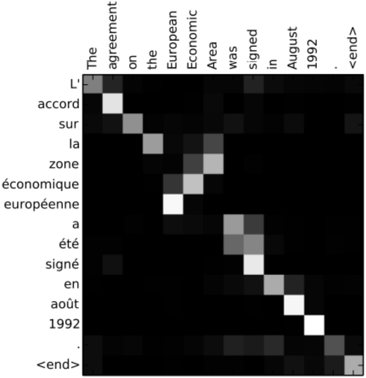
Provides shortcut to faraway states

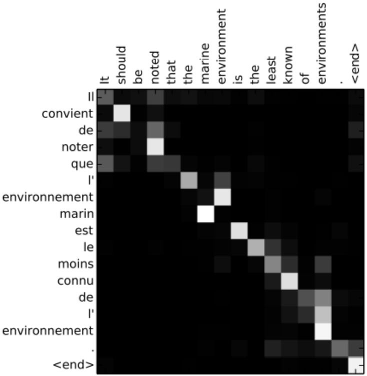
Attention provides some interpretability

- By inspecting attention distribution, we can see what the decoder was focusing on
- We get alignment for free even if we never explicitly trained an alignment system

Example: Machine Translation

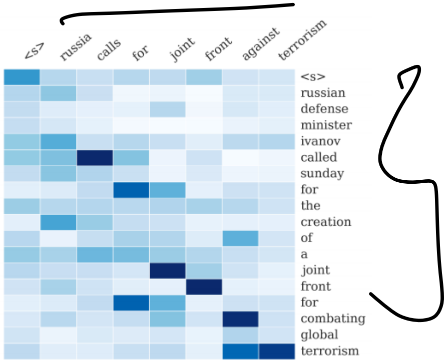
Neural Machine Translation by jointly learning to align and Translate, ICLR 2015





Example: Text Summarization

A Neural Attention Model for Sentence Summarization, EMNLP 2015



Summary

h_1^e h_2^e h_3^e $h_1^d \rightarrow h_2^d$?

softmax 2 -3 1

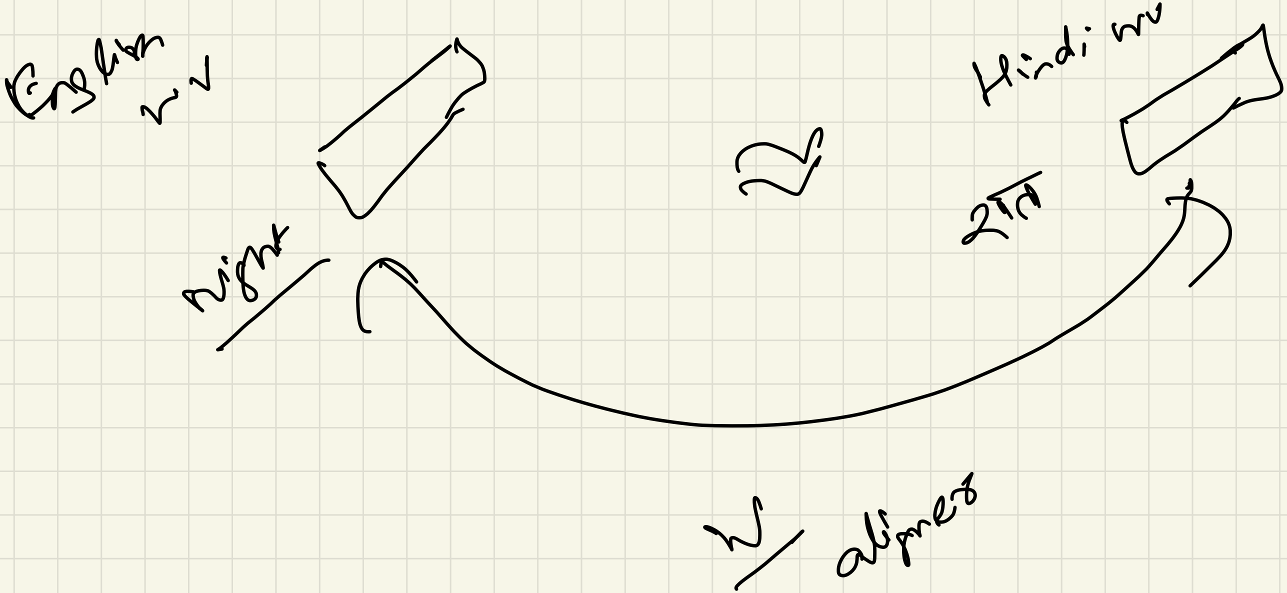
y_1

- Attention has proved to be a very impactful idea in NLP
- Lot of new models are based on self-attention, e.g., Transformer, BERT

e^2 / Σ e^{-3} / Σ e / Σ

$$\sum \alpha_{1j} h_j^e$$

$$\alpha_{1j} = \text{softmax}(h_1^d \cdot h_j^e)$$


English vec
right
≃
Hindi vec
2nd
W
alignmt