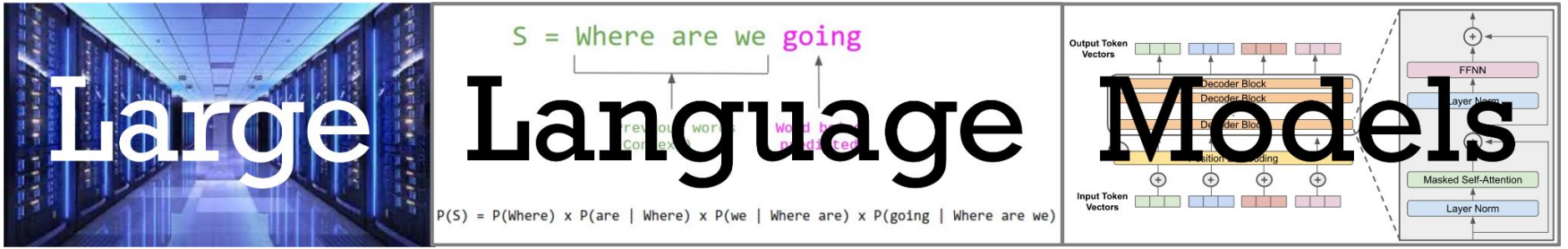


CIS 7000 - Fall 2024

The Pre-Transformer Era

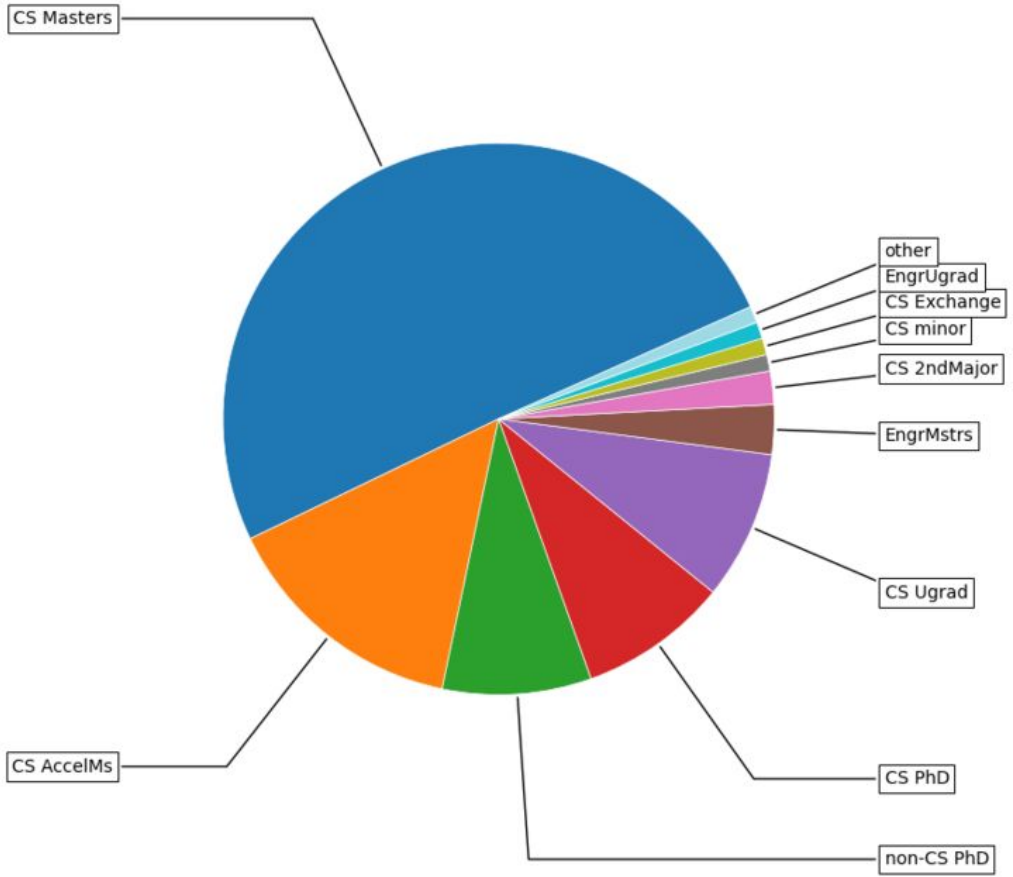
Professor Mayur Naik

Slides adapted in part from UC Berkeley's CS182: Designing, Visualizing and Understanding Deep Neural Networks (Spring'21), Stanford's CS224N: Natural Language Processing with Deep Learning (Spring'24), and Chapters 7 and 8 of Jurafsky/Martin's book "Speech and Language Processing" (3rd ed).

Recap of Last Lecture

- The Turing Test
- Overview of LLMs
  - How do LLMs work, What LLMs can do, Limitations of LLMs, What is the future
- Course Logistics

Who’s In CIS 7000?



Announcements

- **Homework 0** “Exploring LLMs” is due on **Sunday Sept 8 at 11:59 pm ET**. Available via Canvas and https://llm-class.github.io/homeworks.html. Late submissions will not be accepted!

- **Homework 1** “Transformer from Scratch” will be released on **Friday Sept 6**; much more work than Homework 0! Due in 3 parts: **Sun Sept 15** (Part 1), **Sun Sept 22** (Part 2), and **Sun Sept 29 at 11:59 pm ET** (Final).

Today’s Agenda

- Background
  - Language Modeling, Perplexity Evaluation, Feedforward Neural Networks
- Recurrent Neural Networks (RNNs) and their Limitations
- Attention Mechanisms

Background

1. Language Modeling
2. Perplexity Evaluation
3. Feedforward Neural Networks
    a. Forward pass
    b. Loss function
    c. Back-propagation

What is Language Modeling?

The task of computing $P(w \mid h)$, the probability distribution of possible words w from a vocabulary given some history (sequence of words) h.

… and thanks for all the _____

fish	0.70
memories	0.10
support	0.05
help	0.05
love	0.04
time	0.02
work	0.02
fun	0.01
…	…

What is Language Modeling?

The task of computing $P(w \mid h)$, the probability distribution of possible words w from a vocabulary given some history (sequence of words) h.

Enables to compute probabilities of entire sentences by applying **chain rule of probability**:

$P(w_1\ w_2 \ldots w_k) = P(w_1).P(w_2|w_1).P(w_3|w_{1:2})...P(w_k|w_{1:k-1}) = \prod_{i=1}^{k} P(w_i|w_{1:i-1})$

Question: Why is this task important?

Motivation for Language Modeling

- Generating more plausible sentences.
  - *P*(“I saw a van”) > *P*(“eyes awe of an”)
  - Many NLP applications: correcting grammar or spelling errors, machine translation, speech recognition, content summarization, conversational agents, etc.

- More importantly, Large Language Models are built by training them on this task!

An Approximation

- A problem: computing *P(w | h)* exactly is infeasible for arbitrary history *h* since language is **creative** and *h* might have never occurred before!

- Idea: the Markov assumption: approximate the history by just the last few words.

$$P(w_k|w_{1:k-1}) \approx P(w_k|w_{k-n+1:k-1})$$

- Example: n-gram models: look at n-1 words in the history. n=2 is bigrams, n=3 is trigrams, etc.

  LLMs use *much* larger n, in the thousands or even millions!

Estimating Probabilities

Goal: Train a language model P_θ with parameters (weights) θ such that $P_\theta(h)$ computes a probability distribution over the vocabulary of all possible words.

Start with a dataset $D_{train} = \{ (h_1, w_1), \ldots, (h_n, w_n) \}$.

Assumption: i.i.d. (independent and identically distributed)

Question: What is the objective of this model?

A good model is one that makes the data look probable. Therefore, choose θ such that

$$P(D_{train}) = \prod_{i=1}^{n} P(h_i).P_\theta(w_i \mid h_i) \text{ is maximized.}$$

Multiplying Probabilities

$$P(D_{train}) = \prod_{i=1}^{n} P(h_i).P_\theta(w_i \mid h_i)$$

Multiplying together many numbers <= 1

$$\log P(D_{train}) = \sum_{i=1}^{n} \log P(h_i) + \log P_\theta(w_i \mid h_i) = \sum_{i=1}^{n} \log P_\theta(w_i \mid h_i) + \cancel{\text{const}}$$

$$\theta^* \leftarrow \arg\max_\theta \sum_{i=1}^{n} \log P_\theta(w_i \mid h_i)$$ maximum likelihood estimation (MLE)

This is our **loss function**

$$\theta^* \leftarrow \arg\min_\theta -\frac{1}{n} \sum_{i=1}^{n} \log P_\theta(w_i \mid h_i)$$ negative log likelihood (NLL)

Also called *cross-entropy*

Evaluation: How Good is Our Model?

Suppose we train P_θ (more on how to do this later in today's lecture!)

How do we tell how good our LM is?

Does our LM prefer good sentences over bad ones?

- Assign higher probability to real or frequently observed sentences than ungrammatical or rarely observed ones?

Train the parameters of the LM on a **training set** (D_{train}).

Test the LM's performance on data we haven't seen.

- A **test set** is an unseen dataset different from D_{train}, totally unused.
- An **evaluation metric** tells us how well our model does on test set.

Extrinsic Evaluation of LMs

Best evaluation for comparing LMs A and B.

Put each model in a task (e.g. spelling corrector, speech recognizer, machine translation system).

Run the task and get an accuracy for A and B.

- (e.g. how many misspelled words corrected properly, how many words translated properly).

Whichever model has higher accuracy is better.

Problem: Time-consuming.

Intrinsic Evaluation of LMs

Common intrinsic evaluation metric for LMs: **perplexity**.

Bad approximation unless the test data looks *just* like the training data.

Generally only useful in pilot experiments.

But it is helpful to think about (as long as extrinsic evaluation is also done).

Let’s look at different intuitions of perplexity and define it!

Intuition of Perplexity

The Shannon Game: how well can we predict the next word in a given sentence?

I always order pizza with cheese and _____

The 33rd President of the US was _____

I saw a _____

mushrooms 0.1
pepperoni 0.1
anchovies 0.01
…
fried rice 0.0001
…
and 1e-100

A good LM is one that assigns a higher probability to the word that actually occurs.

Perplexity

The best LM is one that best predicts an unseen test set D_{test}.

Perplexity is 2^J where J is cross entropy loss on test set: $J = -\frac{1}{n}\sum_{i=1}^{n} \log P_\theta(w_i \mid h_i)$

Perplexity >= 1. Lower is better.

Equivalently, perplexity of a sentence $w_1\ w_2 \ldots w_n$ is $P_\theta(w_1\ w_2 \ldots w_n)^{-\frac{1}{n}}$ which is the same as $\sqrt[n]{\frac{1}{\prod_{i=1}^{n} P_\theta(w_i|w_{1:i-1})}}$

That is, it is the probability of the sentence normalized by the number of words.

Another Intuition of Perplexity

Perplexity is the **average branching factor** at any point in a sentence.

Example 1: task is to recognize sentence of random digits. Average branching factor at each step is 10, so perplexity is 10.

Example 2: task is to recognize Operator (1 in 4), Sales (1 in 4), Tech Support (1 in 4), and 30,000 names (1 in 120,000 each). Perplexity is ~ 53.

Lower Perplexity = Better Model

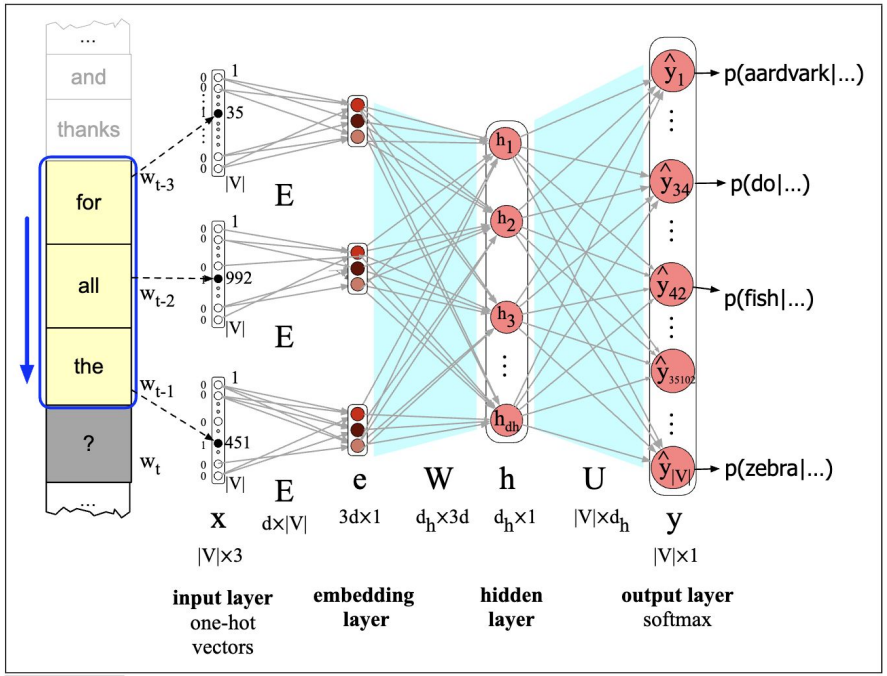
Training on 38 million words and testing on 1.5 million words from the WSJ.

N-gram Order	Unigram	Bigram	Trigram
Perplexity	962	170	109

Further Reading: Hugging Face doc article on Perplexity of fixed-length models.

Feedforward Neural Network for Language Modeling

Architecture and Illustration of Forward Pass



Sketch of feedforward neural language model with N=3.

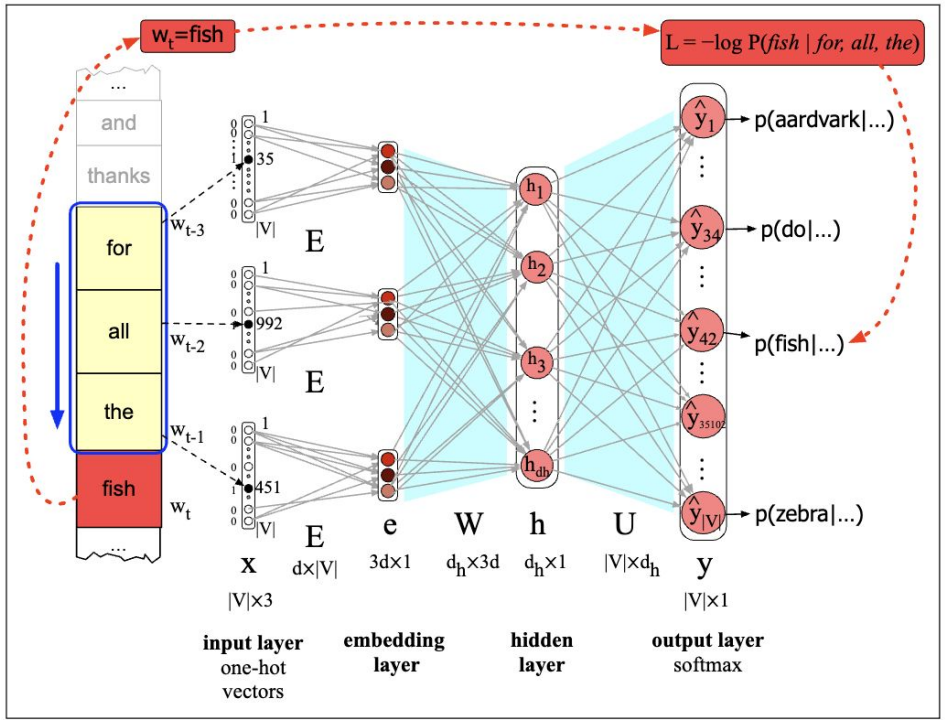
At each timestep *t* the network computes a *d*-dimensional embedding for each context word (by multiplying a one-hot vector by embedding matrix **E**), and concatenates the 3 resulting embeddings to get the embedding layer **e**.

The embedding vector **e** is multiplied by a weight matrix **W** and then an activation function is applied element-wise to produce the hidden layer **h**, which is then multiplied by another weight matrix **U**.

Finally, a softmax output layer predicts at each node *i* the probability that the next word w_t will be vocabulary word V_i.

Y. Benjio et al. A Neural Probabilistic Language Model. JMLR 2003.

Loss Function: Negative Log Likelihood (NLL)

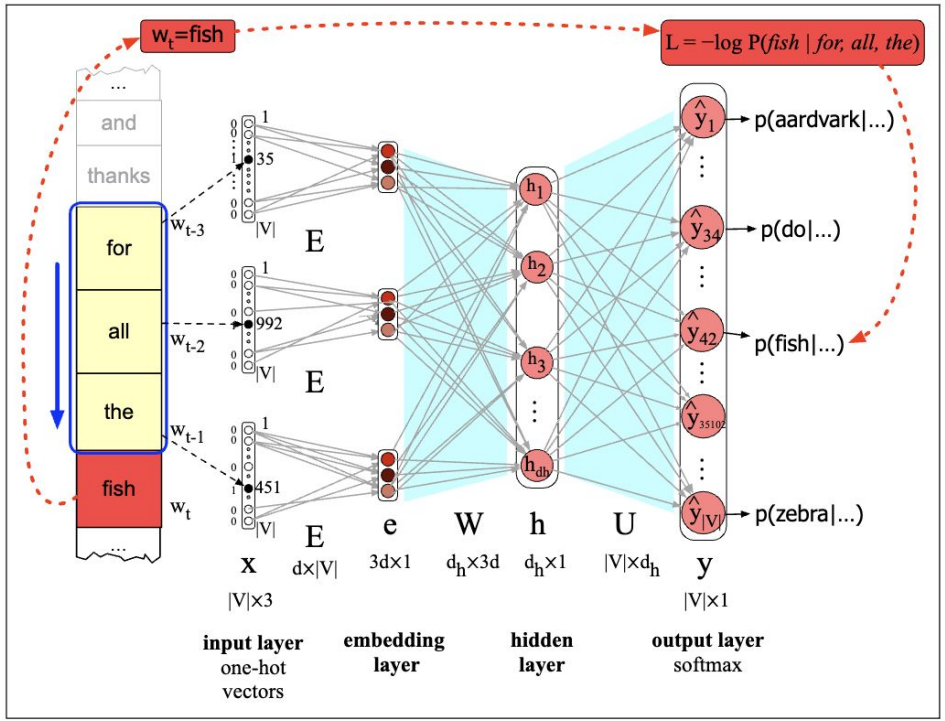


The parameter update for stochastic gradient descent for cross-entropy loss L from step s to s+1 is:

$$\theta^{s+1} = \theta^s - \eta \frac{\partial \left[-\log p(w_t|w_{t-1},\ldots,w_{t-n+1})\right]}{\partial \theta}$$

This gradient can be computed in any standard neural network framework (e.g. Pytorch) which will then backpropagate through θ = **E**, **W**, **U**, **b**.

Forward Pass and Loss Function in Equations



$$\mathbf{e} = [\mathbf{E}\,\mathbf{x}_{t-3}; \mathbf{E}\,\mathbf{x}_{t-2}; \mathbf{E}\,\mathbf{x}_{t-1}]$$

$$\mathbf{h} = \sigma(\mathbf{W}\,\mathbf{e})$$

$$\mathbf{z} = \mathbf{U}\,\mathbf{h}$$

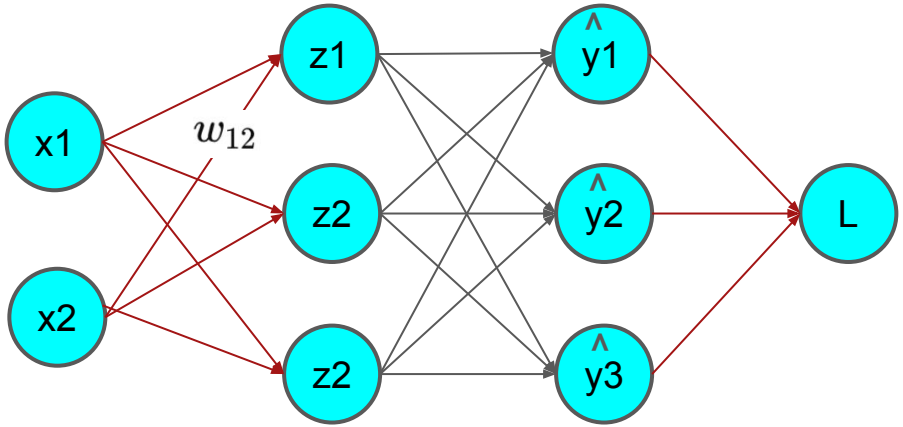
$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$$

$$\mathbf{L} = -\,\mathbf{y}\,(\log \hat{\mathbf{y}})^{T}$$

where **y** is one-hot vector representing ground truth.

Short Primer on Back-Propagation

Let's consider a simple feedforward network:



$$\mathbf{z} = \mathbf{W}\,\mathbf{x}$$
$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$$
$$L = -\,\mathbf{y}\log\hat{\mathbf{y}}$$

where **y** is one-hot vector representing ground truth.

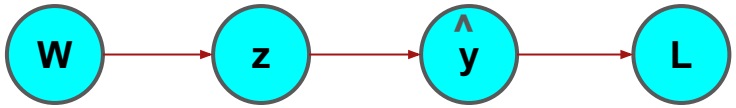
where $\hat{\mathbf{y}}_i = \frac{e^{z_i}}{\sum_{j=1}^{3} e^{z_j}}$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \mathbf{W} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \quad \mathbf{z} = \begin{bmatrix} w_{11}x_1 + w_{12}x_2 \\ w_{21}x_1 + w_{22}x_2 \\ w_{31}x_1 + w_{32}x_2 \end{bmatrix} \quad \hat{\mathbf{y}} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \end{bmatrix} \quad L = -\sum_{i=1}^{3} y_i \log(\hat{y}_i)$$

Short Primer on Back-Propagation

Computation Graph:

(linear in this example but a DAG in general)

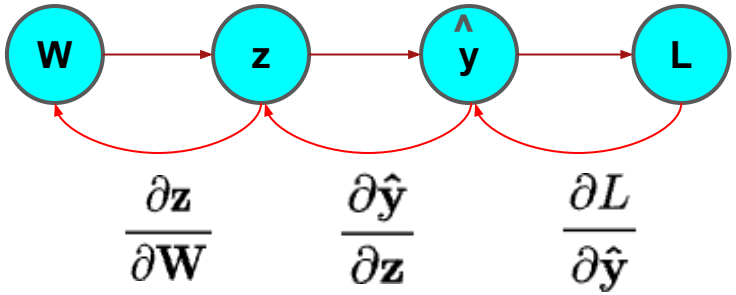


$$\mathbf{z} = \mathbf{W}\,\mathbf{x}$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$$

$$\mathbf{L} = -\,\mathbf{y}\log\hat{\mathbf{y}}$$

Backward Differentiation:

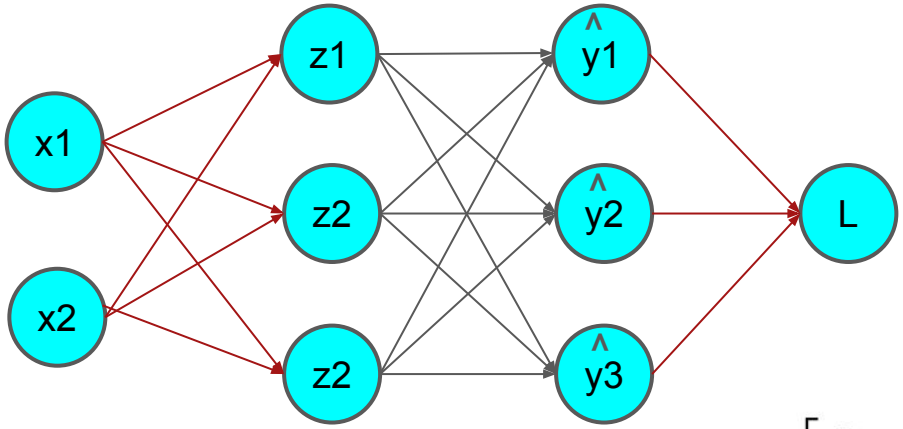


$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{W}}$$

(by chain rule of probability)

Short Primer on Back-Propagation

Gradient of Loss with Respect to Predicted Probabilities:



$$\mathbf{z} = \mathbf{W}\,\mathbf{x}$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$$

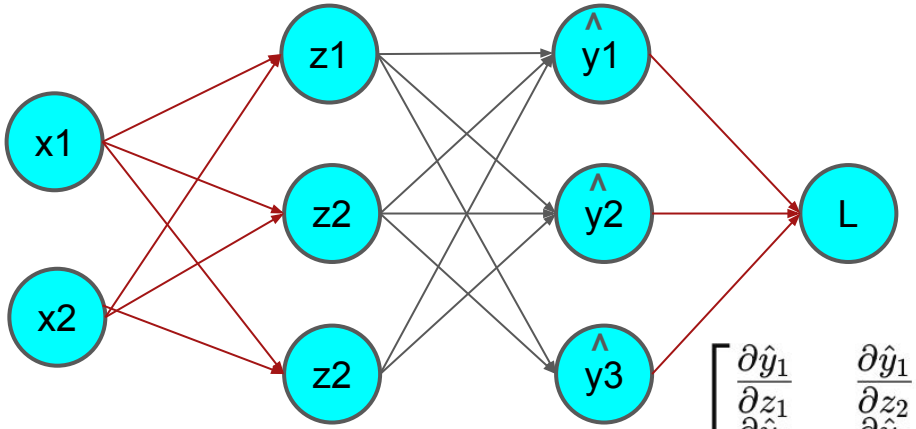
$$\mathbf{L} = -\,\mathbf{y}\log\hat{\mathbf{y}}$$

$$\frac{\partial L}{\partial \hat{\mathbf{y}}} \qquad \left[\frac{\partial L}{\partial \hat{y}_1} \quad \frac{\partial L}{\partial \hat{y}_2} \quad \frac{\partial L}{\partial \hat{y}_3}\right] = \left[-\frac{y_1}{\hat{y}_1} \quad -\frac{y_2}{\hat{y}_2} \quad -\frac{y_3}{\hat{y}_3}\right]$$

since $L = -\sum_{i=1}^{3} y_i \log(\hat{y}_i)$ and $\frac{\partial L}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i}$

Short Primer on Back-Propagation

Jacobian of Softmax with Respect to Logits:



$$\mathbf{z} = \mathbf{W}\,\mathbf{x}$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$$

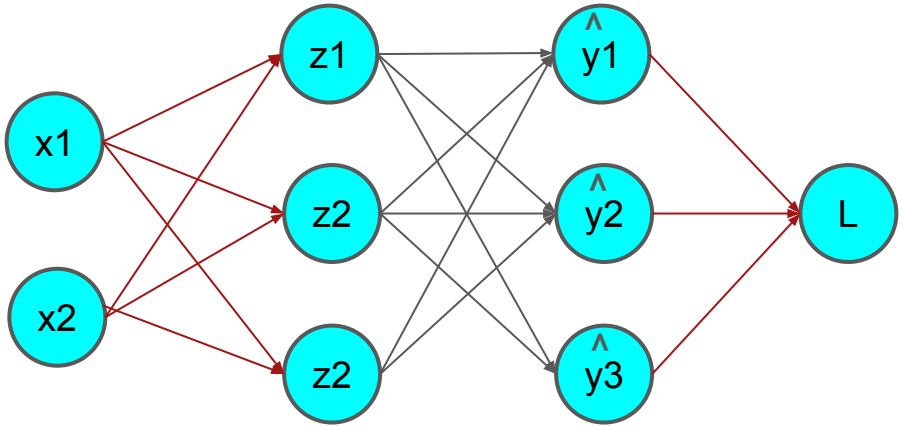
$$\mathbf{L} = -\,\mathbf{y}\log\hat{\mathbf{y}}$$

$$\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{z}} = \begin{bmatrix} \frac{\partial \hat{y}_1}{\partial z_1} & \frac{\partial \hat{y}_1}{\partial z_2} & \frac{\partial \hat{y}_1}{\partial z_3} \\ \frac{\partial \hat{y}_2}{\partial z_1} & \frac{\partial \hat{y}_2}{\partial z_2} & \frac{\partial \hat{y}_2}{\partial z_3} \\ \frac{\partial \hat{y}_3}{\partial z_1} & \frac{\partial \hat{y}_3}{\partial z_2} & \frac{\partial \hat{y}_3}{\partial z_3} \end{bmatrix} = \begin{bmatrix} \hat{y}_1(1-\hat{y}_1) & -\hat{y}_1\hat{y}_2 & -\hat{y}_1\hat{y}_3 \\ -\hat{y}_2\hat{y}_1 & \hat{y}_2(1-\hat{y}_2) & -\hat{y}_2\hat{y}_3 \\ -\hat{y}_3\hat{y}_1 & -\hat{y}_3\hat{y}_2 & \hat{y}_3(1-\hat{y}_3) \end{bmatrix}$$

since $\hat{y}_i = \dfrac{e^{z_i}}{\sum_{k=1}^{3} e^{z_k}}$ and $\dfrac{\partial \hat{y}_i}{\partial z_j} = \begin{cases} \hat{y}_i(1-\hat{y}_i) & \text{if } i = j \\ -\hat{y}_i\hat{y}_j & \text{if } i \neq j \end{cases}$

Short Primer on Back-Propagation

Jacobian of Logits with Respect to Weights:



$$\mathbf{z} = \mathbf{W}\,\mathbf{x}$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$$

$$\mathbf{L} = -\,\mathbf{y}\log\hat{\mathbf{y}}$$

$$\frac{\partial \mathbf{z}}{\partial \mathbf{W}} = \begin{bmatrix} \frac{\partial z_1}{\partial W_{11}} & \frac{\partial z_1}{\partial W_{12}} & \frac{\partial z_1}{\partial W_{21}} & \frac{\partial z_1}{\partial W_{22}} & \frac{\partial z_1}{\partial W_{31}} & \frac{\partial z_1}{\partial W_{32}} \\ \frac{\partial z_2}{\partial W_{11}} & \frac{\partial z_2}{\partial W_{12}} & \frac{\partial z_2}{\partial W_{21}} & \frac{\partial z_2}{\partial W_{22}} & \frac{\partial z_2}{\partial W_{31}} & \frac{\partial z_2}{\partial W_{32}} \\ \frac{\partial z_3}{\partial W_{11}} & \frac{\partial z_3}{\partial W_{12}} & \frac{\partial z_3}{\partial W_{21}} & \frac{\partial z_3}{\partial W_{22}} & \frac{\partial z_3}{\partial W_{31}} & \frac{\partial z_3}{\partial W_{32}} \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & x_1 & x_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & x_1 & x_2 \end{bmatrix}$$

since $z_j = \sum_{n=1}^{2} W_{jn} x_n$ and $\frac{\partial z_j}{\partial W_{mn}} = \begin{cases} x_n & \text{if } j = m \\ 0 & \text{if } j \neq m \end{cases}$

Short Primer on Back-Propagation

Putting it all together:

$$\mathbf{z} = \mathbf{W}\,\mathbf{x}$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$$

$$L = -\,\mathbf{y}\log\hat{\mathbf{y}}$$

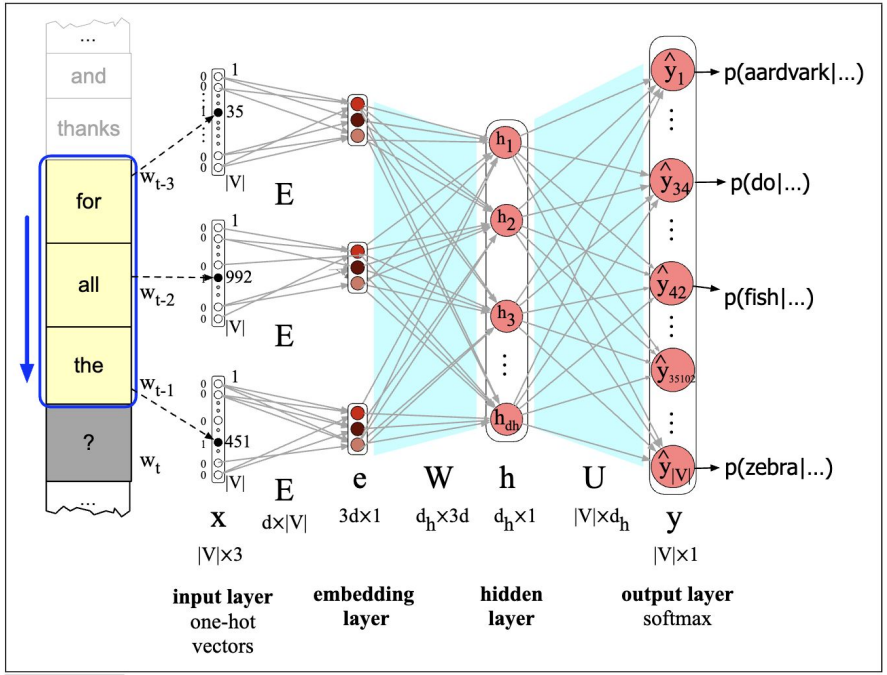
$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{W}}$$

$$= \begin{bmatrix} \frac{\partial L}{\partial \hat{y}_1} & \frac{\partial L}{\partial \hat{y}_2} & \frac{\partial L}{\partial \hat{y}_3} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial \hat{y}_1}{\partial z_1} & \frac{\partial \hat{y}_1}{\partial z_2} & \frac{\partial \hat{y}_1}{\partial z_3} \\ \frac{\partial \hat{y}_2}{\partial z_1} & \frac{\partial \hat{y}_2}{\partial z_2} & \frac{\partial \hat{y}_2}{\partial z_3} \\ \frac{\partial \hat{y}_3}{\partial z_1} & \frac{\partial \hat{y}_3}{\partial z_2} & \frac{\partial \hat{y}_3}{\partial z_3} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial z_1}{\partial W_{11}} & \frac{\partial z_1}{\partial W_{12}} & \frac{\partial z_1}{\partial W_{21}} & \frac{\partial z_1}{\partial W_{22}} & \frac{\partial z_1}{\partial W_{31}} & \frac{\partial z_1}{\partial W_{32}} \\ \frac{\partial z_2}{\partial W_{11}} & \frac{\partial z_2}{\partial W_{12}} & \frac{\partial z_2}{\partial W_{21}} & \frac{\partial z_2}{\partial W_{22}} & \frac{\partial z_2}{\partial W_{31}} & \frac{\partial z_2}{\partial W_{32}} \\ \frac{\partial z_3}{\partial W_{11}} & \frac{\partial z_3}{\partial W_{12}} & \frac{\partial z_3}{\partial W_{21}} & \frac{\partial z_3}{\partial W_{22}} & \frac{\partial z_3}{\partial W_{31}} & \frac{\partial z_3}{\partial W_{32}} \end{bmatrix}$$

$$= \begin{bmatrix} -\frac{y_1}{\hat{y}_1} & -\frac{y_2}{\hat{y}_2} & -\frac{y_3}{\hat{y}_3} \end{bmatrix} \cdot \begin{bmatrix} \hat{y}_1(1-\hat{y}_1) & -\hat{y}_1\hat{y}_2 & -\hat{y}_1\hat{y}_3 \\ -\hat{y}_2\hat{y}_1 & \hat{y}_2(1-\hat{y}_2) & -\hat{y}_2\hat{y}_3 \\ -\hat{y}_3\hat{y}_1 & -\hat{y}_3\hat{y}_2 & \hat{y}_3(1-\hat{y}_3) \end{bmatrix} \cdot \begin{bmatrix} x_1 & x_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & x_1 & x_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & x_1 & x_2 \end{bmatrix}$$

$$= \begin{bmatrix} (\hat{y}_1 - y_1)x_1 & (\hat{y}_1 - y_1)x_2 & (\hat{y}_2 - y_2)x_1 & (\hat{y}_2 - y_2)x_2 & (\hat{y}_3 - y_3)x_1 & (\hat{y}_3 - y_3)x_2 \end{bmatrix}$$

Pros/Cons of Feedforward Neural Network as Language Model



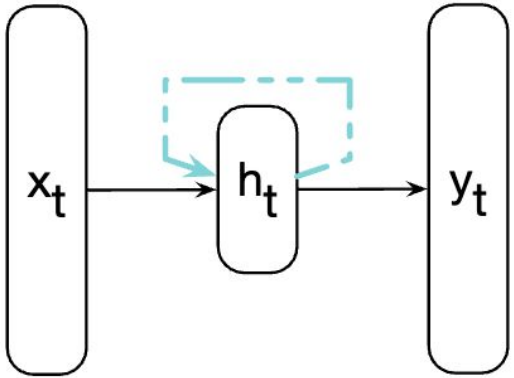
Improvements over n-gram LM:

- No sparsity problem.
- No need to store all observed n-grams.

Remaining problems:

- Fixed window is too small.
- Model size (W) increases for longer input context.
- Window can never be large enough!
- Each x_i is multiplied by completely different weights in W, so no symmetry in how the inputs are processed.

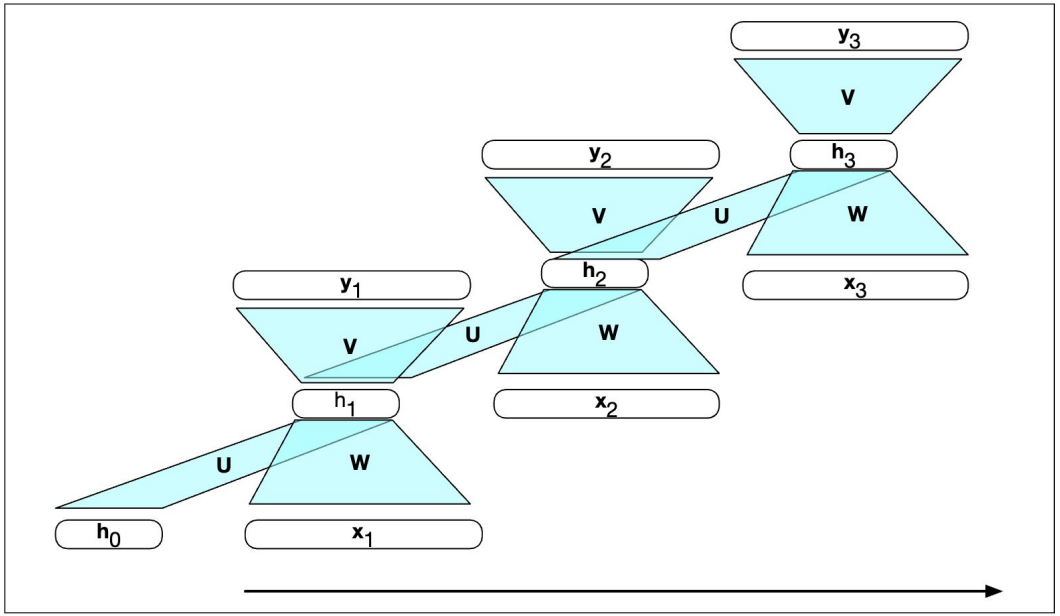
Recurrent Neural Networks



$$\mathbf{h}_t = \sigma\left(\mathbf{U}\,\mathbf{h}_{t-1} + \mathbf{W}\,\mathbf{x}_t\right)$$

$$\mathbf{y}_t = \text{softmax}(\mathbf{V}\,\mathbf{h}_t)$$

Recurrent Neural Networks: Forward Pass



$\mathbf{h}_0 = 0$

for **t** = 1 to length(**x**) do

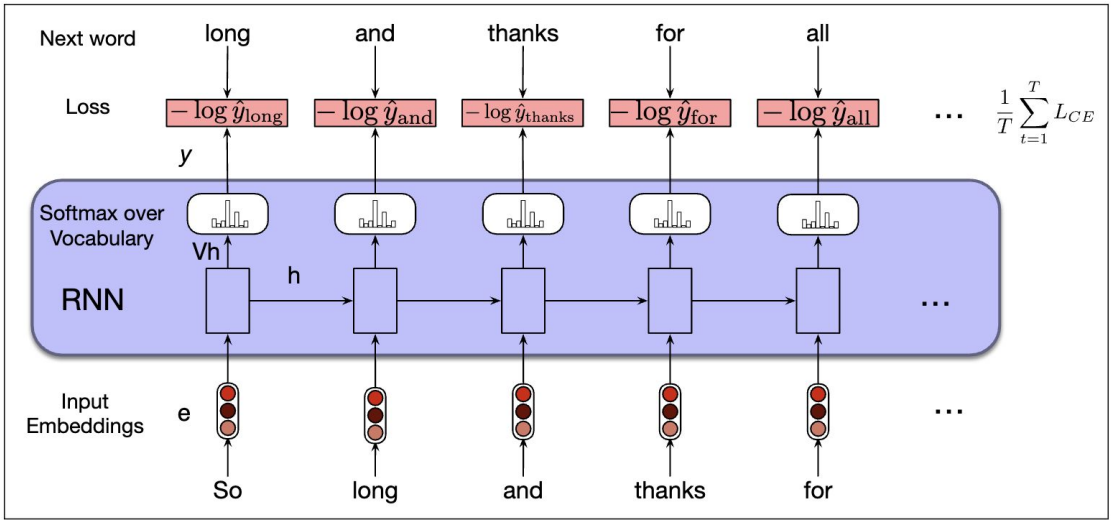
$\mathbf{h}_t = \sigma(\mathbf{U}\,\mathbf{h}_{t-1} + \mathbf{W}\,\mathbf{x}_t)$

$\mathbf{y}_t = \text{softmax}(\mathbf{V}\,\mathbf{h}_t)$

return **y**

The matrices **U**, **V** and **W** are shared across time, while new values for **h** and **y** are calculated with each time step.

Recurrent Neural Networks: Training

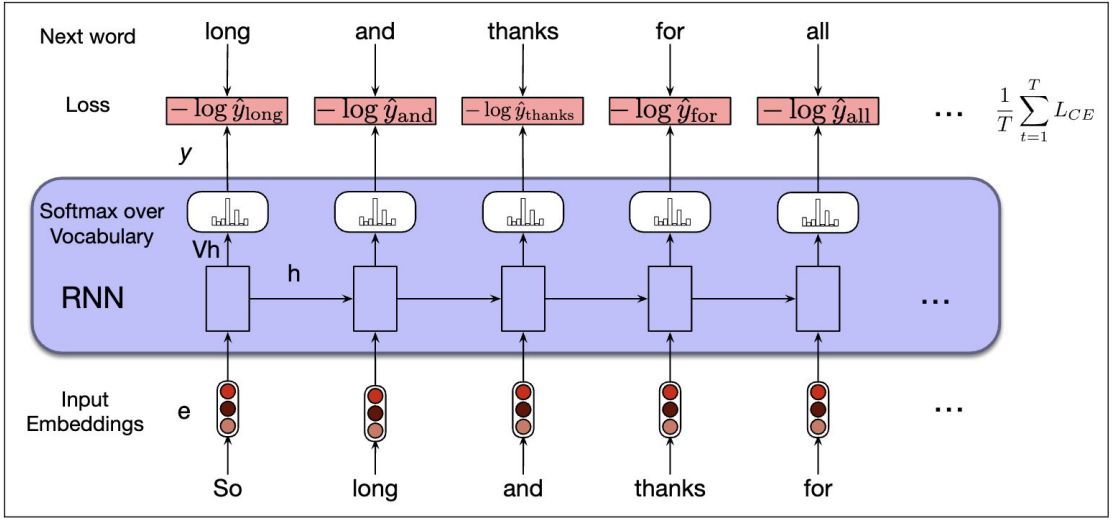


At each word position t of the input, the model takes word w_t together with $\mathbf{h_{t-1}}$, encoding information from the preceding $\mathbf{w_{1:t-1}}$, and uses them to compute a probability distribution over possible next words so as to compute the model's loss for the next word $\mathbf{w_{t+1}}$.

Then we move to the next word, ignore what the model predicted for the next word and instead use the correct word $\mathbf{w_{t+1}}$ along with the prior history encoded to estimate the probability of word $\mathbf{w_{t+2}}$.

This idea that we always provide the model the correct history sequence to predict the next word (rather than feeding the model its best case from the previous time step) is called **teacher forcing**.

Recurrent Neural Networks: Backpropagation

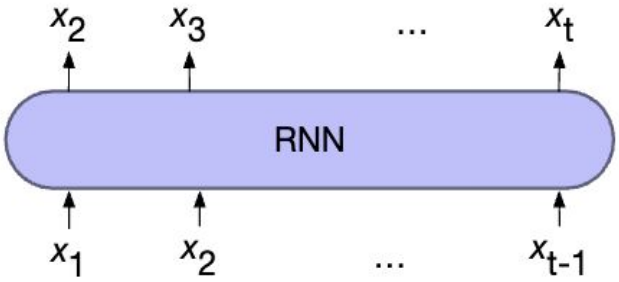


Backpropagate over timesteps t = T, … ,1, summing gradients as you go. This algorithm is called “backpropagation through time”.

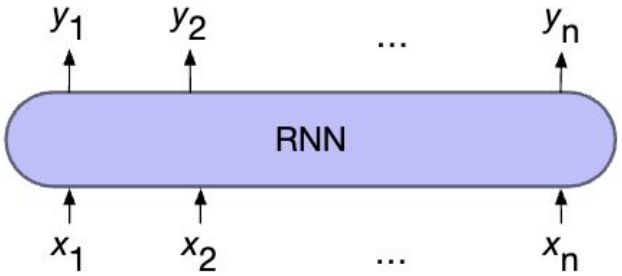
Computing loss and gradients over entire corpus at once is too expensive (memory-wise). In practice, consider (a batch of) sentences at a time.

In practice, often “truncated” after ~20 timesteps for training efficiency reasons.

RNN Architectures for Different NLP Tasks

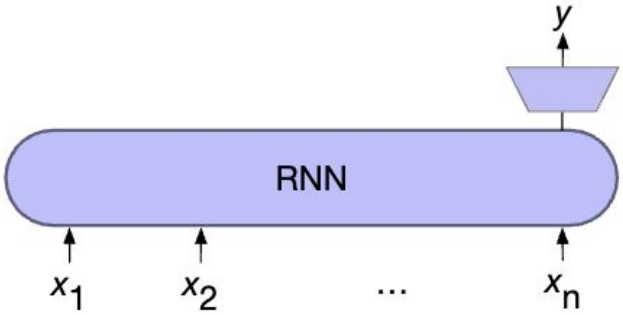


Language Modeling



Sequence Labeling

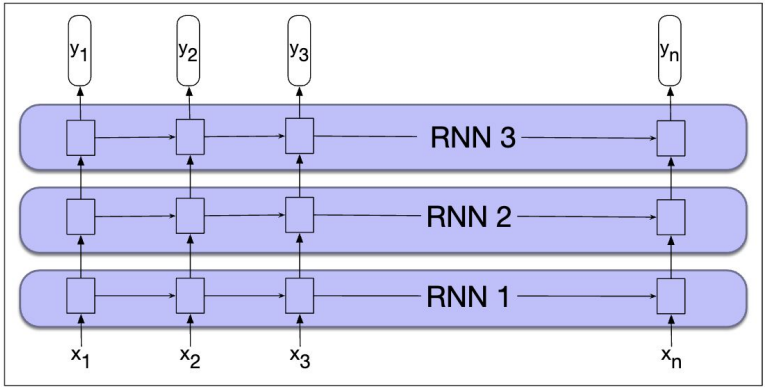
Applications:
Part-of-speech tagging,
Named entity recognition



Sequence Classification

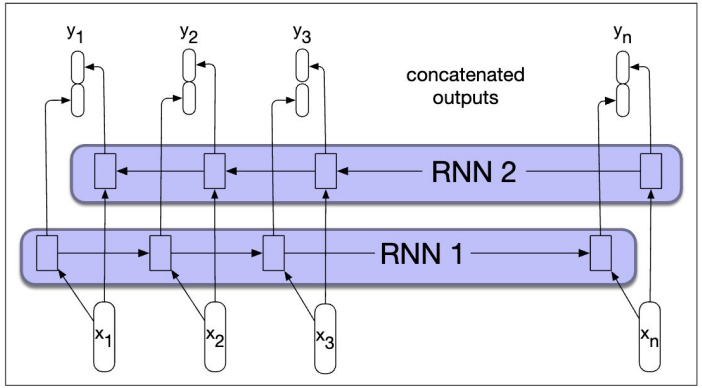
Applications:
sentiment analysis,
spam detection

Variations of RNNs



Stacking

The output of a lower level serves as the input to higher levels with the output of the last network serving as the final output. The resulting network induces representations at differing levels of abstraction across layers.



Bidirectional

Separate models are trained in the forward and backward directions, with the output of each model at each time point concatenated to represent the bidirectional state at that time point.

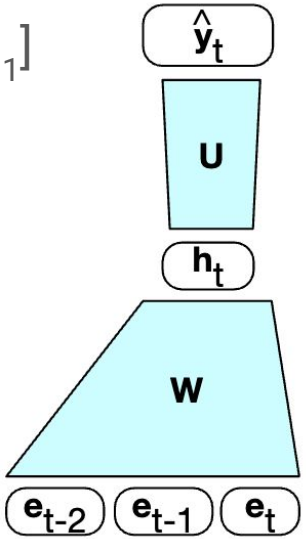
Feedforward vs. Recurrent Neural Network as Language Model

$\mathbf{e} = [\mathbf{E}\, x_{t-3}; \mathbf{E}\, x_{t-2}; \mathbf{E}\, x_{t-1}]$

$\mathbf{h} = \sigma(\mathbf{W}\, \mathbf{e})$

$\mathbf{z} = \mathbf{U}\, \mathbf{h}$

$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$

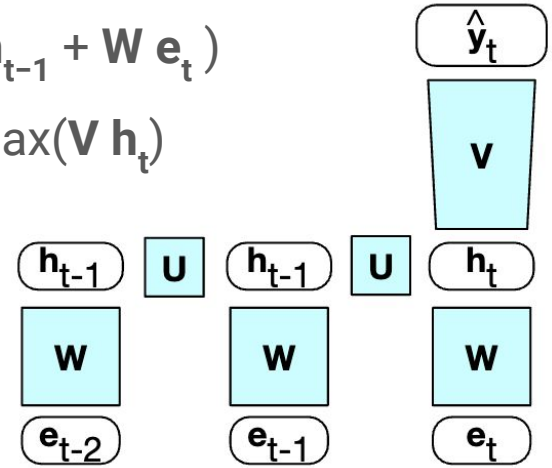


Feedforward Neural Network

$\mathbf{e}_t = \mathbf{E}\, x_t$

$\mathbf{h}_t = \sigma(\mathbf{U}\, \mathbf{h}_{t-1} + \mathbf{W}\, \mathbf{e}_t)$

$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{V}\, \mathbf{h}_t)$



Recurrent Neural Network

Pros/Cons of Recurrent Neural Network as Language Model

Pros:

- Can process any length input!
- Computation for step *t* can (in theory) use information from many steps back.
- Model size (W) doesn't increase for longer input context.
- Each x_i is multiplied by same weights in W, so there is symmetry in how inputs are processed.

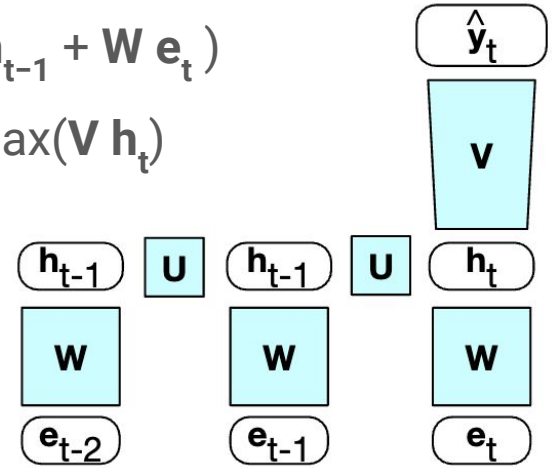
Cons:

- Recurrent computation is slow (cannot be parallelized).
- In practice, it is difficult to access information from many steps back.

$$e_t = E\, x_t$$

$$h_t = \sigma\,(U\, h_{t-1} + W\, e_t\,)$$

$$\hat{y}_t = \text{softmax}(V\, h_t)$$



Vanishing Gradient Problem!
RNN extensions such as LSTMs (or other varieties like GRUs) are commonly used.

Effect of Vanishing Gradient on RNN-Based Language Model

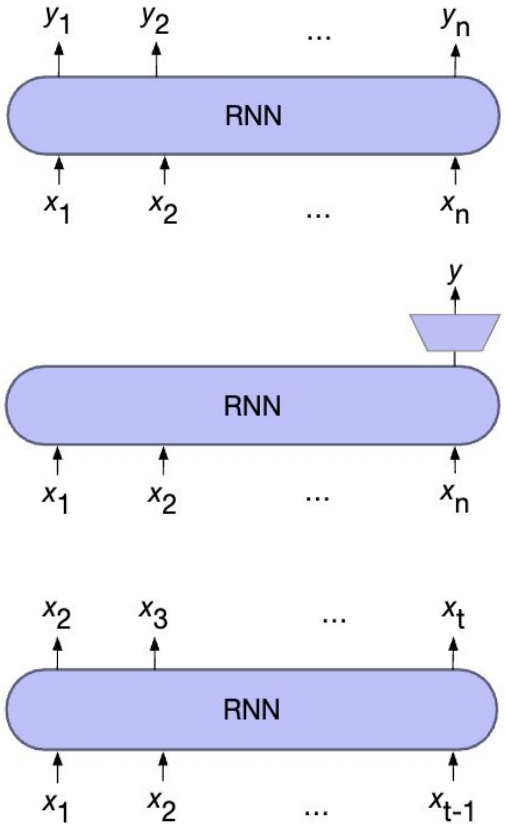
LM task: *When she tried to print her tickets, she found that the printer was out of toner. She went to the stationery store to buy more toner. It was very overpriced. After installing the toner into the printer, she finally printed her _____*

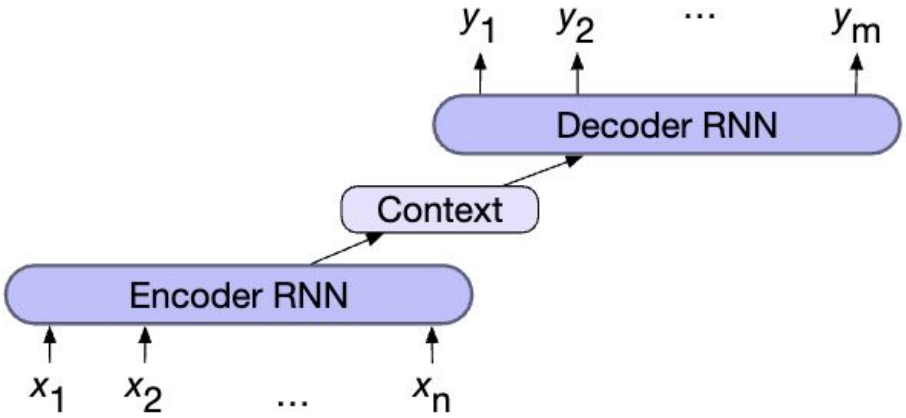
To learn from this training example, the RNN-LM needs to model the dependency between *"tickets"* on the 7^{th} step and the target word *"tickets"* at the end.

But if the gradient is small, the model can't learn this dependency. So, the model is unable to predict similar long-distance dependencies at test time.

More on vanishing/exploding gradient problems in training RNNs: R. Pascanu et al. On the difficulty of training recurrent neural networks. ICML 2013.

Encoder-Decoder or Seq2Seq Architecture

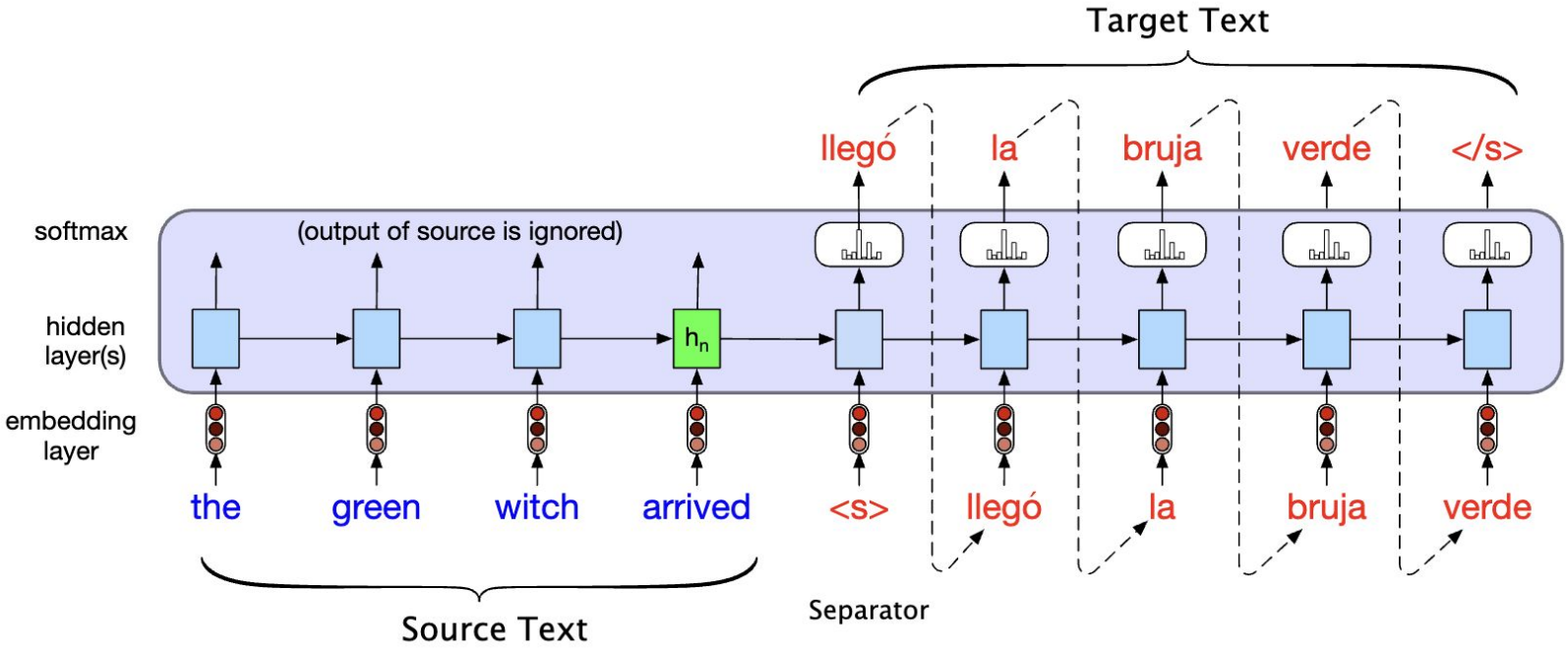




Taking an input sequence and translating it to an output sequence that is of a different length than the input, and doesn't align with it in a word-to-word way.

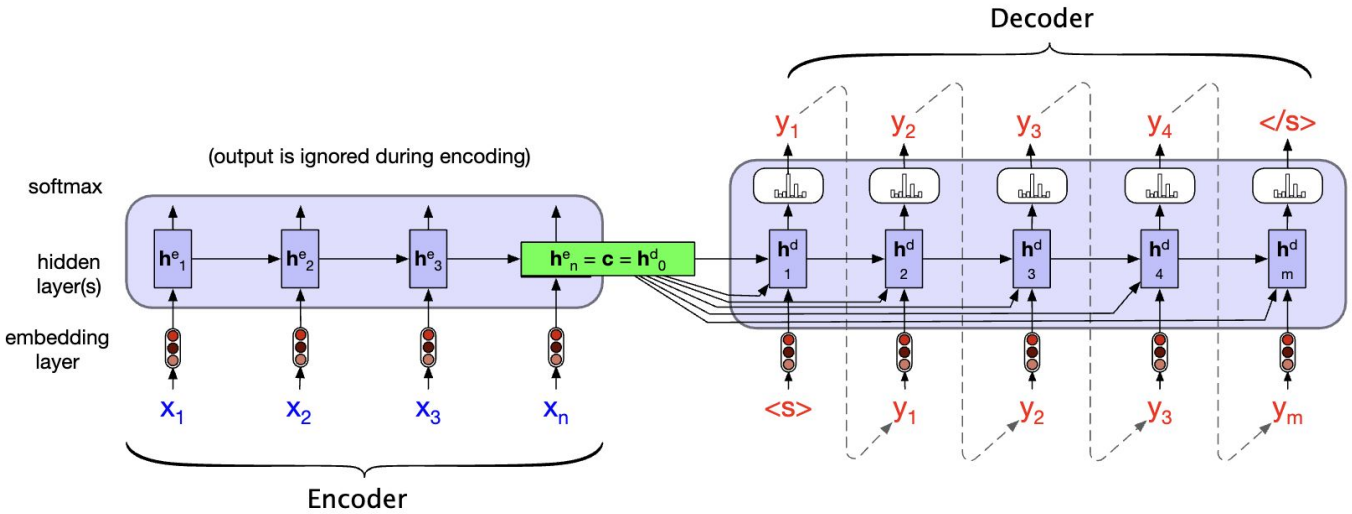
Many applications: machine translation, summarization, question-answering, dialogue, …

Illustrative Example: Machine Translation



Encoder-Decoder RNN, More Formally

The final hidden state of the encoder RNN, $\mathbf{h}_n^e$, serves as the context for the decoder in its role as $\mathbf{h}_0^d$ in the decoder RNN, and is also made available to each decoder hidden state.



$$\mathbf{c} = \mathbf{h}_n^e$$

$$\mathbf{h}_0^d = \mathbf{c}$$

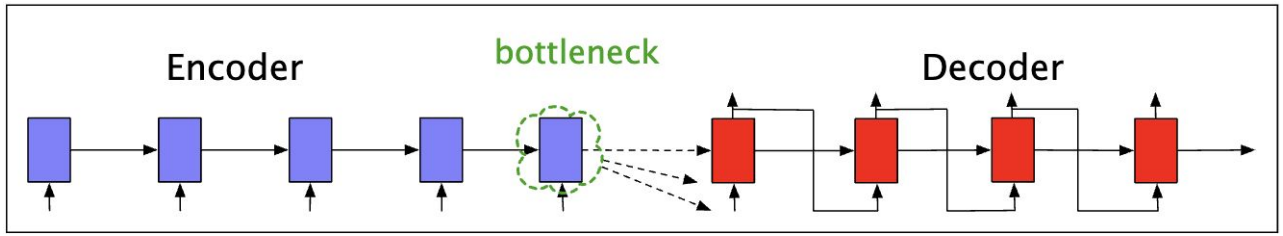
$$\mathbf{h}_i^d = g(\hat{\mathbf{y}}_{i-1}, \mathbf{h}_{i-1}^d, \mathbf{c})$$

$$\hat{\mathbf{y}}_i = \text{softmax}(\mathbf{h}_i^d)$$

where **g** is a stand-in for some flavor of RNN.

The Bottleneck Problem

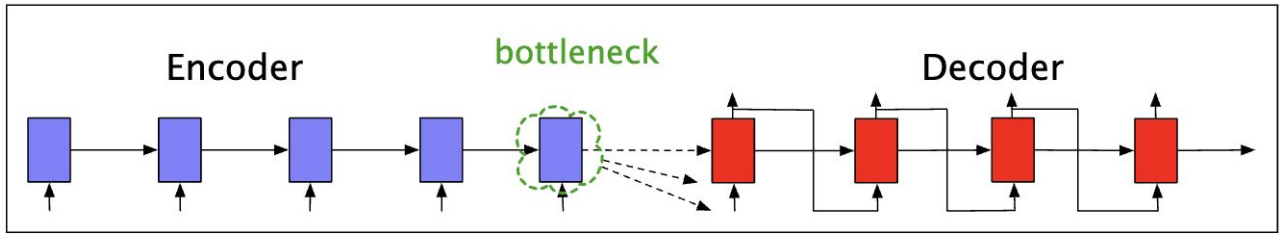
Requiring the context **c** to be only the encoder's final hidden state forces all the information from the entire source sentence to pass through this representational bottleneck!



D. Bahdanau et al. Neural Machine Translation by Jointly Learning to Align and Translate. ICLR 2015.

Attention Mechanism: A Solution to the Bottleneck Problem

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!



$$\mathbf{c} = \mathbf{h}_n^e$$

$$\mathbf{h}_0^d = \mathbf{c}$$

$$\mathbf{h}_i^d = g(\hat{\mathbf{y}}_{i-1}, \mathbf{h}_{i-1}^d, \mathbf{c}_i)$$

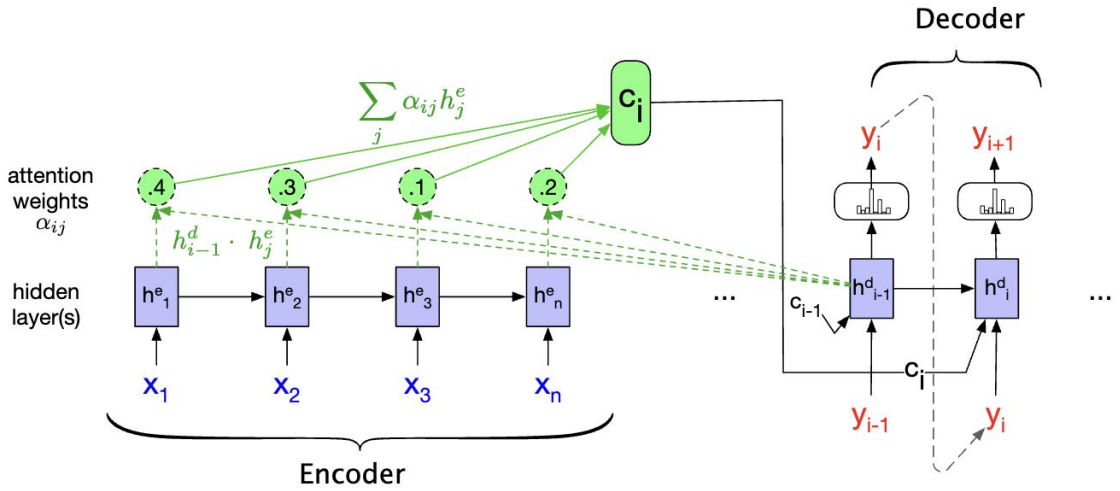
$$\hat{\mathbf{y}}_i = \text{softmax}(\mathbf{h}_i^d)$$

Instead of a static context vector **c**, dynamically generated a context vector $\mathbf{c}_i$ for each decoding step **i** that takes all encoder hidden states into account in its derivation.

$$\mathbf{c}_i = f(\mathbf{h}_1^e, \ldots, \mathbf{h}_n^e)$$

D. Bahdanau et al. Neural Machine Translation by Jointly Learning to Align and Translate. ICLR 2015.

Dot-Product Attention



Finally compute a fixed-length context vector for the current decoder state by taking a weighted average over all the encoder hidden states.

First step in computing $\mathbf{c}_i$: How much to focus on each encoder state $\mathbf{h}_j^e$, or how relevant it is to the decoder state captured in $\mathbf{h}_{i-1}^d$.

$$\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e) = \mathbf{h}_{i-1}^d \cdot \mathbf{h}_j^e$$

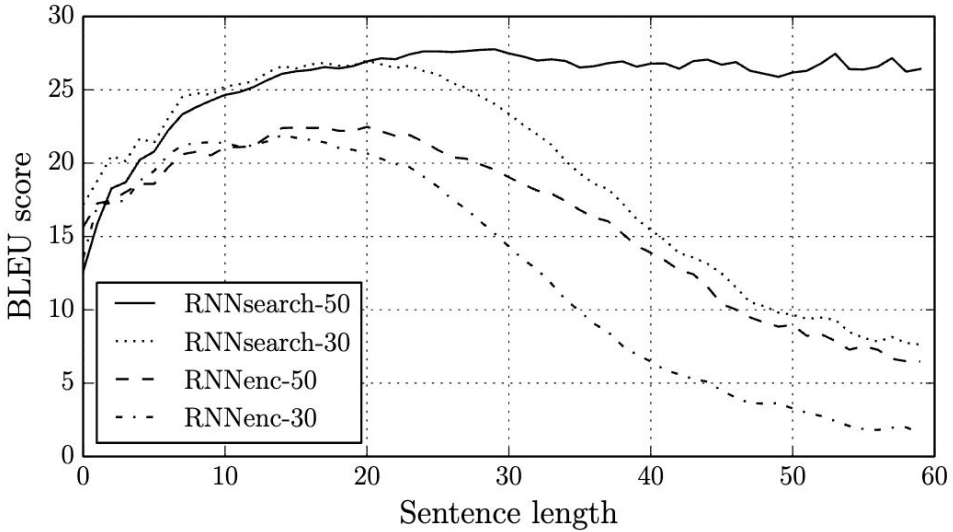
Then normalize the weights.

$$\boldsymbol{\alpha}_{ij} = \text{softmax}(\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e))$$

$$\mathbf{c}_i = \sum_j \boldsymbol{\alpha}_{ij} \mathbf{h}_j^e$$

More sophisticated scoring function: $\mathbf{h}_{i-1}^d \mathbf{W} \mathbf{h}_j^e$ using learned weights $\mathbf{W}$.

Experimental Results With vs. Without Attention Mechanism



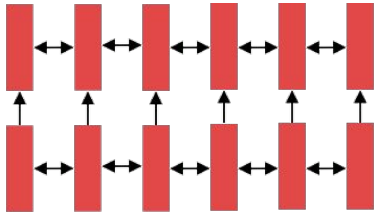
Performance of RNN encoder-decoder models trained on sentences of length upto 30 or 50:

RNNsearch - with attention
RNNenc - without attention

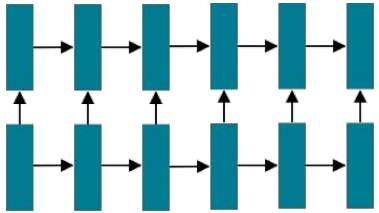
D. Bahdanau et al. Neural Machine Translation by Jointly Learning to Align and Translate. ICLR 2015.

The Story So Far: RNNs for (Most) NLP

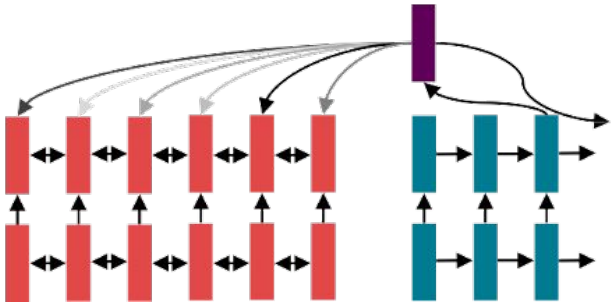
- Circa 2016, the de facto strategy in NLP is to encode sentences with a bidirectional LSTM (e.g., the source sentence in a translation).



- Define your output (parse, sentence, summary) as a sequence, and use an LSTM to generate it.



- Use attention to allow flexible access to memory.



Motivation for Transformer

- Minimize (or at least not increase) computational complexity per layer.
- Minimize path length between any pair of words to facilitate learning of long-range dependencies.
- Maximize the amount of computation that can be parallelized.

1. Transformer Motivation: Computational Complexity Per Layer

When sequence length (n) << representation dimension (d), the complexity per layer is lower for a Transformer model compared to RNN models.

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

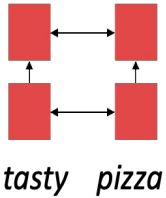
Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Table 1 of paper by A. Vaswani et al. Attention Is All You Need. NeurIPS 2017.

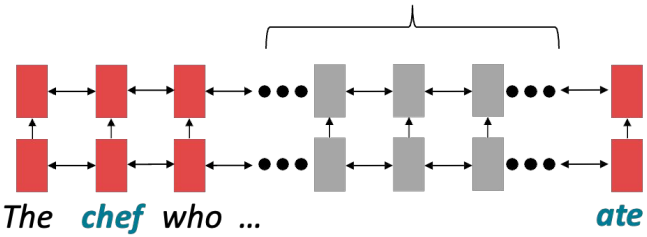
2. Transformer Motivation: Minimize Linear Interaction Distance

- RNN is unrolled “left-to-right”.
- It encodes linear locality: a useful heuristic!
- **Problem:** RNN takes **O(sequence length)** steps for distant word pairs to interact.
  - Hard to learn long-distance dependencies due to gradient problems.
  - Linear order of words is “baked in”; we already know sequential structure doesn't tell the whole story...

Nearby words often affect each other’s meanings



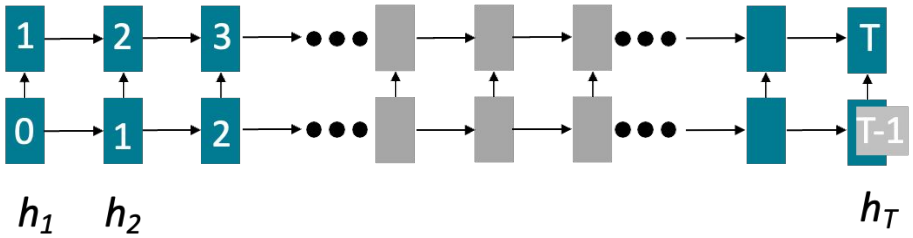
Info about ***chef*** has gone through O(sequence length) many layers!



3. Transformer Motivation: Maximize Parallelizability

Forward and backward passes have **O(sequence length)** unparallelizable operations.

- GPUs (and TPUs) can perform many independent computations at once!
- But future RNN hidden states can't be computed in full before past RNN hidden states have been computed.
- Inhibits training on very large datasets!
- Particularly problematic as sequence length increases, as we can no longer batch many examples together due to memory limitations.

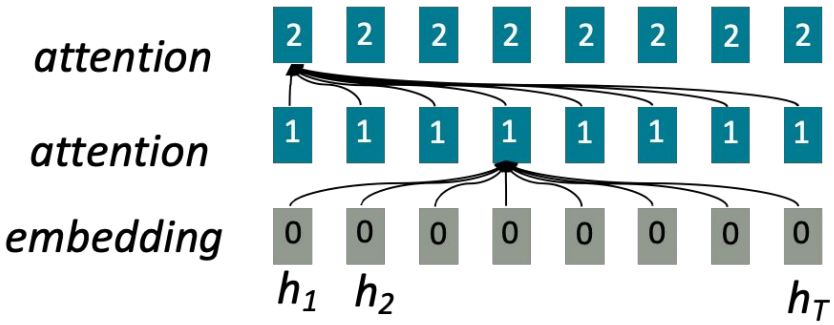


Numbers indicate min # of steps before a state can be computed

High-Level Architecture: Transformer is all about (Self) Attention

Earlier, we saw attention from the decoder to the encoder in a recurrent sequence-to-sequence model.

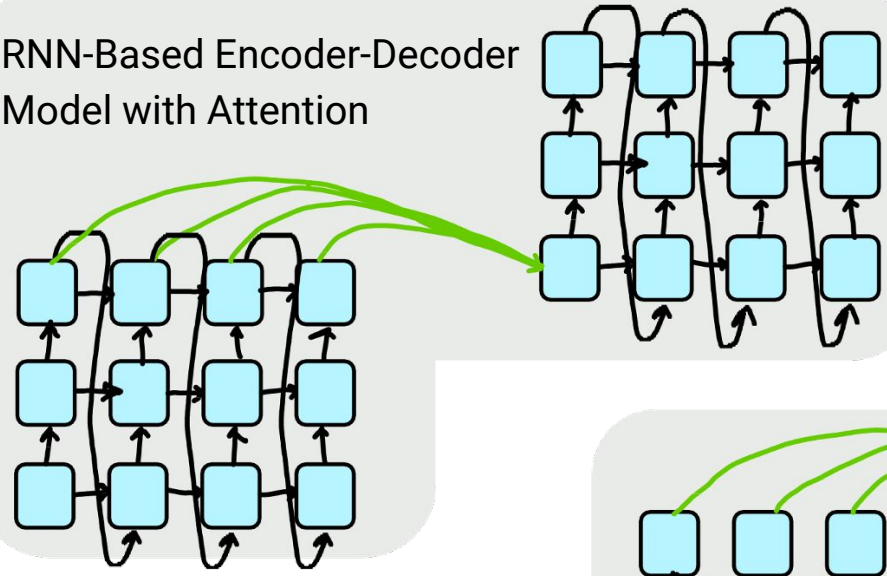
Self-attention is encoder-encoder (or decoder-decoder) attention where each word attends to each other word **within the input (or output)**.



All words attend to all words in previous layer; most arrows here are omitted.

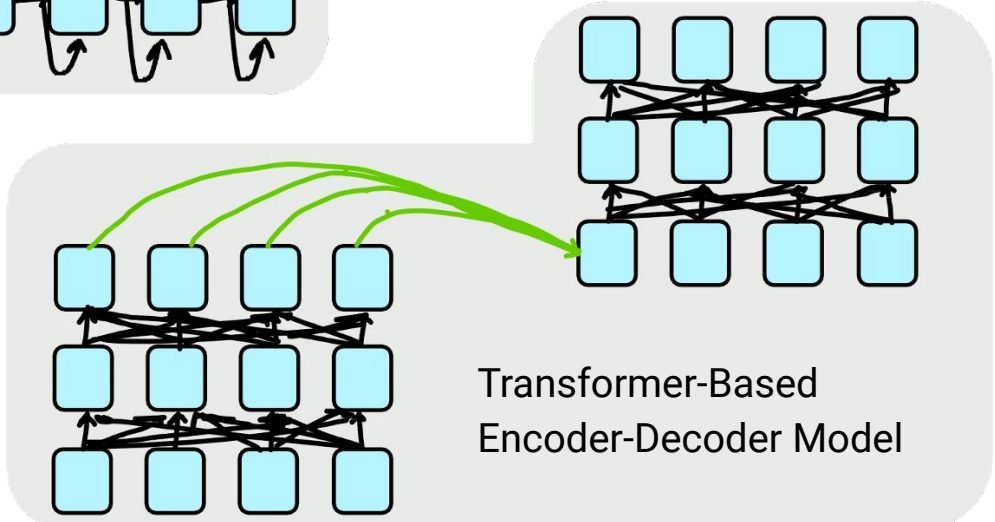
Computational Dependencies for Recurrence vs. Attention

RNN-Based Encoder-Decoder Model with Attention



Transformer Advantages:

- # unparallelizable operations does not increase with sequence length.
- Each word interacts with each other, so maximum interaction distance is O(1).



Transformer-Based Encoder-Decoder Model

Up Next …

- **Mon Sept 9:** Tutorial on “From Pytorch to Hugging Face: How to run your own LLM”.

  Topics: Pytorch framework and Hugging Face Transformers library.

- **Wed Sept 11:** Lecture on The Transformer Architecture.

  Topics: Self-Attention, Transformer Block, Input/Output Processing, Types of Transformers, Training and Inference, Case Studies (BERT, GPT-2, Mixtral).