Language Modeling and Sentence Representation

Armand Joulin, Édouard Grave

Facebook AI Research
{ajoulin,egrave}@fb.com

What is this lecture about?

- 2 core standard problems in NLP:
  - Language modeling
  - Sentence representation

What is this lecture about?

- 2 core standard problems in NLP:
  - Language modeling
  - Sentence representation
- Applications of language modeling:
  - machine transaltion
  - speech recognition
  - anything where we need to generate text

What is this lecture about?

- 2 core standard problems in NLP:
  - Language modeling
  - Sentence representation
- Applications of language modeling:
  - machine transaltion
  - speech recognition
  - anything where we need to generate text
- Applications of sentence representation
  - question answering
  - fact checking
  - anything where we need to compare sentences

Plan of this lecture

- Language modeling:
  - Pre-deep learning
  - Standard neural Networks
  - Recurrent neural networks
- Sentence representation
  - BiLSTM
  - Transformer network
  - BERT

Slides on *n*-grams are inspired by Dan Jurafsky's class

https://web.stanford.edu/class/cs124/lec/

Introduction to language modeling

What is language modeling?

- **Language modeling:** learning a probability distribution of text

Why is language modeling important?

- Speech into text:

Why is language modeling important?

- Speech into text:

$$P(\text{"Vanilla ice cream"}) \text{ or } P(\text{"Vanilla, I scream"}) \text{ ?}$$

Why is language modeling important?

- Speech into text:

$$P(\text{"Vanilla ice cream"}) > P(\text{"Vanilla, I scream"}).$$

Why is language modeling important?

- Speech into text:

$$P(\text{"Vanilla ice cream"}) \; > \; P(\text{"Vanilla, I scream"}).$$

- Translating "Pleasantly surprised":

Why is language modeling important?

- Speech into text:

$$P(\text{"Vanilla ice cream"}) \;>\; P(\text{"Vanilla, I scream"}).$$

- Translating "Pleasantly surprised":

$$P(\text{"Agréablement surpris"}) \text{ or } P(\text{"Déçu en bien"}) \text{ ?}$$

Why is language modeling important?

- Speech into text:

$$P(\text{"Vanilla ice cream"}) \; > \; P(\text{"Vanilla, I scream"}).$$

- Translating "Pleasantly surprised":

$$P(\text{"Agréablement surpris"}) \; > \; P(\text{"Déçu en bien"}).$$

Why is language modeling important?

- Speech into text:

$$P(\text{"Vanilla ice cream"}) > P(\text{"Vanilla, I scream"}).$$

- Translating "Pleasantly surprised":

$$P(\text{"Agréablement surpris"}) > P(\text{"Déçu en bien"}).$$

- Image to text:

$$P(\text{"stop"}) > P(\text{"st0p"}).$$

What is language modeling?

- **Language modeling:** learning a probability distribution of text

What is language modeling?

- **Language modeling:** learning a probability distribution of text

- Text = sequence of "tokens":

  words, characters, groups of characters

What is language modeling?

- **Language modeling:** learning a probability distribution of text
- Text = sequence of "tokens":

  words, characters, groups of characters
- For example:

  $$\text{a cat} \quad = \{\text{a,cat}\}, \qquad \text{(words)}$$

What is language modeling?

- **Language modeling:** learning a probability distribution of text

- Text = sequence of "tokens":

  words, characters, groups of characters

- For example:

$$\begin{aligned} \text{a cat} \quad &= \{\text{a,cat}\}, \quad && \text{(words)} \\ &= \{\text{a, ,c,a,t}\}, \quad && \text{(characters)} \end{aligned}$$

What is language modeling?

- **Language modeling:** learning a probability distribution of text
- Text = sequence of "tokens":

  words, characters, groups of characters

- For example:

$$\begin{aligned} \text{a cat} &= \{\text{a,cat}\}, && \text{(words)} \\ &= \{\text{a, ,c,a,t}\}, && \text{(characters)} \\ &= \{\text{a, ,ca,t}\}. && \text{(groups)} \end{aligned}$$

What is language modeling?

- **Language modeling:** learning a probability distribution of text
- Text = sequence of "tokens":

  words, characters, groups of characters

- For example:

$$\begin{aligned} \text{a cat} \quad &= \{\text{a,cat}\}, && \text{(words)} \\ &= \{\text{a, ,c,a,t}\}, && \text{(characters)} \\ &= \{\text{a, ,ca,t}\}. && \text{(groups)} \end{aligned}$$

- For most of this lecture, we assume that tokens are words

What is language modeling?

- text = sequence of tokens = $\{w_1, \ldots, w_T\}$
- A language model estimates its probability: $P(w_1, \ldots, w_T)$

Probabilistic language model

- Sequence probability as a product of token probabilities:

$$P(w_1, \ldots, w_T) = \prod_{t=1}^{T} P(w_t \mid w_{t-1}, \ldots, w_1)$$

Probabilistic language model

- Sequence probability as a product of token probabilities:

$$P(w_1, \ldots, w_T) = \prod_{t=1}^{T} P(w_t \mid w_{t-1}, \ldots, w_1)$$

- Indeed we have:

$$P(a, b) = P(a)P(b \mid a)$$

Probabilistic language model

- Sequence probability as a product of token probabilities:

$$P(w_1, \ldots, w_T) = \prod_{t=1}^{T} P(w_t \mid w_{t-1}, \ldots, w_1)$$

- Indeed we have:

$$P(a, b) = P(a)P(b \mid a)$$

- Recursively applied to a sequence:

$$\begin{aligned} P(w_1, w_2, w_3) &= P(w_1)P(w_2, w_3 \mid w_1) \\ &= P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_2, w_1). \end{aligned}$$

Probabilistic language model

- Sequence probability as a product of token probabilities:

$$P(w_1, \ldots, w_T) = \prod_{t=1}^{T} P(w_t \mid w_{t-1}, \ldots, w_1)$$

- Indeed we have:

$$P(a, b) = P(a)P(b \mid a)$$

- Recursively applied to a sequence:

$$\begin{aligned} P(w_1, w_2, w_3) &= P(w_1)P(w_2, w_3 \mid w_1) \\ &= P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_2, w_1). \end{aligned}$$

- Language model = learn probability of upcoming token given past:

$$P(w_t \mid w_{t-1}, \ldots, w_1).$$

Probabilistic language model

- The probability $P(w_t \mid w_{t-1}, \ldots, w_1)$ depends on a vocabulary
- **vocabulary** $=$ the set of all unique tokens.

Probabilistic language model

- The probability $P(w_t \mid w_{t-1}, \ldots, w_1)$ depends on a vocabulary
- **vocabulary** $=$ the set of all unique tokens.
- Naively, the bigger a vocabulary is, the less probable a token is.

Probabilistic language model

- The probability $P(w_t \mid w_{t-1}, \ldots, w_1)$ depends on a vocabulary
- **vocabulary** = the set of all unique tokens.
- Naively, the bigger a vocabulary is, the less probable a token is.
- Example: $P(\text{"car"} \mid \text{I'm driving a"}) = ?$
  - 1 if vocabulary contains "car" but not "moto"
  - 0.5 if vocabulary contains "car" and "moto".

n-gram language models

Count based language model

- **Idea**:

  text is discrete $\rightarrow$ count occurences of words to form probabilities

- Advantages:
  - no learning, efficient and simple
  - does not require a lot of computational resources
  - works well in practice

Count based language model

How to compute probability from counting statistics:

Count based language model

How to compute probability from counting statistics:

- Count how many times a sequence of tokens occurs in dataset.

Count based language model

How to compute probability from counting statistics:

- Count how many times a sequence of tokens occurs in dataset.
- Compute probability from this count:

$$\begin{aligned} P(w_t \mid w_{t-1}, \ldots, w_1) &= \frac{P(w_1, \ldots, w_t)}{P(w_1, \ldots, w_{t-1})} \\ &= \frac{c(w_1 \cdots w_t)}{c(w_1 \cdots w_{t-1})} \end{aligned}$$

$c(w_1 \cdots c_T)$ is the number of occurences of the sequence $w_1 \cdots w_T$

Count based language model

- Example:

  Sentence “The moment one learns English” appears 35 in dataset

  Sentence “The moment one learns” appears 75 in dataset

Count based language model

- Example:
  Sentence "The moment one learns English" appears 35 in dataset
  Sentence "The moment one learns" appears 75 in dataset

$$
\begin{aligned}
P(\text{English} \mid \text{The moment one learns}) &= \frac{c\,(\text{The moment one learns English})}{c\,(\text{The moment one learns})} \\
&= \frac{35}{73} = 0.48
\end{aligned}
$$

Limitiations of count based language model

- Number of unique sentences increases with dataset size,
- Long sentences are rare: no good statistics for them

$\rightarrow$ **Too many sentences with not enough statistics**

Count based language model

- **Solution:** truncate past to a fixed size window
- For example:

$$P(\text{English} \mid \text{The moment one learns}) \approx P(\text{English} \mid \text{one learns})$$

- Implicit assumption:

  most important information about a word is in its recent history

- **Beware!** In general:

$$P(w_1, \ldots, w_T) \neq \prod_{t=1}^{T} P(w_t \mid w_{t-1}, \ldots, w_{t-n+1})$$

Count based language model

- **Truncated count based models = *n*-gram models**
- “n” refers to the size of past
- Examples:
  - Unigram:

$$P(w_1, \ldots, w_T) = \prod_{t=1}^{T} P(w_t)$$

  - Bigram:

$$P(w_1, \ldots, w_T) = \prod_{t=1}^{T} P(w_t \mid w_{t-1})$$

Count based language model

- **Truncated count based models = *n*-gram models**
- "n" refers to the size of past
- Examples:
  - Unigram:

$$P(w_1, \ldots, w_T) = \prod_{t=1}^{T} P(w_t)$$

  - Bigram:

$$P(w_1, \ldots, w_T) = \prod_{t=1}^{T} P(w_t \mid w_{t-1})$$

Count based language model: unigram

- Probability of a sentence with a unigram model:

$$P_U(w_1, \ldots, w_T) = \prod_{t=1}^{T} P(w_t) = \prod_{t=1}^{T} \frac{c(w_t)}{N}$$

N = total number of tokens in dataset
$c(w_t)$ = number of occurences of w_t in dataset

- **Unigram only uses word frequency**

- Example of text generation with this model:

the or is ball then car

Count based language model: bigram

- Probability of a sentence with a bigram model:

$$P_U(w_1, \ldots, w_T) = \prod_{t=1}^{T} P(w_t \mid w_{t-1}) = \prod_{t=1}^{T} \frac{c(w_{t-1} w_t)}{c(w_{t-1})}$$

$c(w_{t-1} w_t) =$ number of occurences of sequence $w_{t-1} w_t$

- **Predict a word just with the previous word**

Count based language model: bigram

- Example of text generation with bigram model:

new car parking lot of the

- "car" is generated from "new", "parking" from "car"...
- But "new" has no influence on "parking"

Count based language model

- Simple to extend to longer dependencies: trigrams, 4-grams...
- *n*-grams can be "good enough" in some cases
- **But *n*-grams cannot capture long term dependencies required to truely model language**

Estimating n-gram probabilites: an example

- Bigram:

$$P(w_t \mid w_{t-1}) = \frac{c(w_{t-1} w_t)}{c(w_{t-1})}$$

- Dataset:

$$<s> \textit{we sat in the house}$$
$$<s> \textit{we sat here we two and we said}$$
$$<s> \textit{how we wish we had something to do}$$

- Extract some probabilities:

$$P(\textit{sat} \mid \textit{we}) = 0.33,\ P(\textit{wish} \mid \textit{we}) = 0.17,\ P(\textit{in} \mid \textit{sat}) = 0.5$$

- $<$s$>$= token for beginning of sentence; $P(<\mathrm{s}>) = 1$.
- Compute sentence probability with them

Estimating n-gram probabilites: an example

- Extract count from Berkeley Restaurant dataset (9222 sentences)
- Unigram counts:

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

- Bigram counts:

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

Estimating n-gram probabilites: an example

- The bigram probabilities are obtained by dividing the bigram counts with the unigram counts:

$$P(w_2 \mid w_1) = \frac{c(w_1 w_2)}{c(w_1)}$$

- Resulting bigram probabilities:

	i	want	to	eat	chinese	food	lunch	spend
i	0.022	0.33	0	0.036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

Estimating *n*-gram probabilites: an example

- Example:

$$P(<\text{s}>\text{ i want chinese food})?$$

$<\text{s}>=$ token for beginning of sentence; $P(<\text{s}>) = 1$.

- Result:

$$\begin{aligned} P(<\text{s}>\text{ i want chinese food}) &= P(<\text{s}>)P(\text{i}|<\text{s}>)P(\text{want}|\text{i})P(\text{chinese}|\text{want})P(\text{food}|\text{chinese}) \\ &= 1 \times .25 \times 0.33 \times 0.0065 \times 0.52 \\ &= 0.00027885 \end{aligned}$$

Estimating *n*-gram probabilites: an example

	i	want	to	eat	chinese	food	lunch	spend
i	0.022	0.33	0	0.036	0	0	0	0.00079
...								
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

- Example:

$$P(<\text{s}> \text{ i bring my lunch to work})?$$

- Result:

$$\begin{aligned} P(<\text{s}> \text{ i bring my lunch to work}) &= P(<\text{s}>) \dots P(\text{to}|\text{lunch}) \dots \\ &= 1 \times \cdots \times 0 \times \dots \\ &= 0 \end{aligned}$$

- **Does not generalize well!**

Estimating *n*-gram probabilites: an example

- Simple fix = Add 1 to each bigram count

	i	want	to	eat	chinese	food	lunch	spend
i	6	828	1	10	1	1	1	3
want	3	1	609	2	7	7	6	2
to	3	1	5	687	3	1	7	212
eat	1	1	3	1	17	3	43	1
chinese	2	1	1	1	1	83	2	1
food	16	1	16	1	2	5	1	1
lunch	3	1	1	1	1	2	1	1
spend	2	1	2	1	1	1	1	1

- Laplace-smoothed bigrams:

$$\frac{c(w_i w_j) + 1}{c(w_i) + V},$$

where V = vocabulary size

Estimating n-gram probabilites

- Add mass to unrealistic bigram ("to to").
- Decrease probability of realistic bigram by factor V.
- Example: $P(\text{want} \mid \text{i})$ decreases from 0.33 to 0.21!

$\rightarrow$ Add-1 is not good in practice

Backoff and Interpolation

- If no good statistics on long context: use shorter context
- **Backoff**: use trigram if enough data, else backoff to bigram.
- **Interpolation**: mix statistics of trigram, bigram and unigram.
- In practice interpolation works better

Backoff model

- **Backoff** estimates probability with longest reliable available *n*-gram
- It backs off through shorter and shorter *n*-grams until one is reliable
- Examples:
  - Katz's smoothing (Katz, 1987)
  - Stupid backoff model (Brants et al., 2007)

Stupid backoff

- A n-gram is reliable if it appears in the dataset
- If $c(w_{t-n+1} \cdots w_t) > 0$:

$$P_{bo}(w_t \mid w_{t-n+1}, \ldots, w_{t-1}) = \frac{c(w_{t-n+1} \cdots w_t)}{c(w_{t-n+1} \cdots w_{t-1})}.$$

- else backoff to $(n-1)$gram:

$$P_{bo}(w_t \mid w_{t-n+1}, \ldots, w_{t-1}) = 0.4 P_{bo}(w_t \mid w_{t-n+2}, \ldots, w_{t-1})$$

- Apply recursively until a existing n-gram is found
- **Problem:** probabilities do not sum to 1!
- But works well with a lot of data

Linear Interpolation

- Simple linear interpolation:

$$\begin{aligned} P_L(w_t \mid w_{t-1}, w_{t-2}) &= \lambda_1 P(w_t \mid w_{t-1}, w_{t-2}) + \\ &\quad \lambda_2 P(w_t \mid w_{t-1}) + \\ &\quad \lambda_3 P(w_t) \end{aligned}$$

- Conditioned interpolation:

$$\begin{aligned} P_L(w_t \mid w_{t-1}, w_{t-2}) &= \lambda_1(w_{t-1}, w_{t-2}) P(w_t \mid w_{t-1}, w_{t-2}) + \\ &\quad \lambda_2(w_{t-1}, w_{t-2}) P(w_t \mid w_{t-1}) + \\ &\quad \lambda_3(w_{t-1}, w_{t-2}) P(w_t) \end{aligned}$$

Kneser-Ney Smoothing (advanced)

- Cover most popular *n*-gram model: **Kneser-Ney Smoothing**
- Very efficient, run on CPUs. Best performing *n*-gram model,
- Available in many standard libraries:
  `https://kheafield.com/code/kenlm/estimation/`

Kneser-Ney Smoothing (advanced)

- Kneser-Ney is a recursive interpolation model

- The probability of a *n*-gram is:

$$P_{KN}(w_t \mid w_{t-n+1}^{t-1}) = f_{KN}(w_{t-n+1}^{t}) + \lambda(w_{t-n+1}^{t-1}) P_{KN}(w_t \mid w_{t-n+1}^{t-2})$$

where $w_{t-n+1}^{t} = w_{t-n+1} \cdots w_t$.

Kneser-Ney Smoothing (advanced)

- Kneser-Ney is a recursive interpolation model

- The probability of a *n*-gram is:

$$P_{KN}(w_t \mid w_{t-n+1}^{t-1}) = f_{KN}(w_{t-n+1}^{t}) + \lambda(w_{t-n+1}^{t-1}) P_{KN}(w_t \mid w_{t-n+1}^{t-2})$$

where $w_{t-n+1}^{t} = w_{t-n+1} \cdots w_t$.

- Recursively unroll to get the explicit probability:

$$\begin{aligned} P_t &= f_t + \lambda_t P_{t-1} \\ &= f_t + \lambda_t (f_{t-1} + \lambda_{t-1} P_{t-2}) \\ &= f_t + \lambda_t f_{t-1} + \cdots + \prod_{k=0}^{t} \lambda_k P_0 \end{aligned}$$

Kneser-Ney Smoothing: absolute discount

$$P_{KN}(w_t \mid w_{t-n+1}^{t-1}) = f_{KN}(w_{t-n+1}^{t}) + \lambda(w_{t-n+1}^{t-1}) P_{KN}(w_t \mid w_{t-n+1}^{t-2})$$

Kneser-Ney Smoothing: absolute discount

$$P_{KN}(w_t \mid w_{t-n+1}^{t-1}) = f_{KN}(w_{t-n+1}^{t}) + \lambda(w_{t-n+1}^{t-1}) P_{KN}(w_t \mid w_{t-n+1}^{t-2})$$

- The contribution of the current *n*-gram is:

$$f_{KN}(w_{t-n+1}^{t}) = \frac{\max(c(w_{t-n+1}^{t}) - d, 0)}{c(w_{t-n+1}^{t-1})}$$

where $d \leq 1$ is discount factor

Kneser-Ney Smoothing: absolute discount

$$P_{KN}(w_t \mid w_{t-n+1}^{t-1}) = f_{KN}(w_{t-n+1}^{t}) + \lambda(w_{t-n+1}^{t-1}) P_{KN}(w_t \mid w_{t-n+1}^{t-2})$$

- λ is the interpolation weight:

$$\lambda(w_{t-n+1}^{t-1}) = \frac{d}{c(w_{t-n+1}^{t-1})} \left| \{ w \mid c(w_{t-n+1}^{t-1} w) > 0 \} \right|$$

It depends on number of words that can appear after w_{t-n+1}^{t-1}

Kneser-Ney Smoothing: lower order distribution

- Let's consider the bigram case:

$$P_{KN}(w_t \mid w_{t-1}) = f_{KN}(w_{t-1}\ w_t) + \lambda(w_{t-1}) P_{KN}(w_t)$$

- How to define $P_{KN}(w_{t-1})$?
- Instead of unigram probability, define probability of unique context:

$$P_{KN}(w_t) = \frac{|\{w \mid c(w\ w_t) > 0\}|}{|\{w, w' \mid c(w\ w') > 0\}|}$$

- This distribution sum to 1 too.

Open versus closed vocabulary

- Closed vocabulary:
  - The vocabulary of the train set covers the vocabulary of the test set
  - The size of the vocabulary V is fixed
- Open vocabulary:
  - Vocabulary of test set is different from vocabulary of train set
  - We have Out Of Vocabulary (OOV) words
  - Train set is big and test set has same distribution: **OOVs are rare words**

Training with OOVs

- OOVs do not appear in the training set

$\rightarrow$ Need to simulate OOVs in the training set

- Create a $<$UNK$>$ token for unknown words
- Replace the rare words in the training vocabulary to $<$UNK$>$
- Rare words: words that appear less than some times (e.g. 10 times)
- Your model will learn to predict $<$UNK$>$ instead of rare words

- Your vocabulary + $<$UNK$>$ covers the test set.

Language models toolkits

Toolkits for standard *n*-grams based LM models

- SRILM: `http://www.speech.sri.com/projects/srilm`
- KenLM: `https://kheafield.com/code/kenlm`

All the *n*-gram models are implemented, simple to use and to deploy!

Evaluation for Language Modeling

- A standardized train/validation/test split
- A metric for model selection
- Build model on train, pick best model based on metric on validation

What is good metric for language modeling?

What is a good model?

- Best option: evaluate the model on a target downstream task
  - machine translation
  - speech recognition
  - ...
- Given two models, keep the one with best result on this task
- This is an **extrinsic** evaluation.

Extrinsic evaluation

Problems:

- Evaluation depends on many other components
- Time consuming
- May require several downstream tasks to assess quality of models

This is why we commonly use an **intrinsic** evaluation called **perplexity**

Intuition of Perplexity

With great power comes great ______

Model 1		Model 2		Model 3	
current	0.5	**responsability**	0.4	**responsability**	0.8
responsability	0.4	responsabilities	0.3	current	0.1
voltage	0.1	irresponsability	0.3	volt	0.1

What is the best model?

Intuition of Perplexity

With great power comes great ______

Model 1		Model 2		Model 3	
current	0.5	**responsability**	0.4	**responsability**	0.8
responsability	0.4	responsabilities	0.3	current	0.1
voltage	0.1	irresponsability	0.3	volt	0.1

What is the best model?

- Accuracy: 2 and 3
- Prec@2: 1, 2 and 3
- Highest probability: 3

Best language model assigns **highest probability** to correct word

Definition of Perplexity

- The perplexity PP of a sentence $W = (w_1, \ldots, w_T)$ is:

$$\begin{aligned} PP(W) &= P(w_1, \ldots, w_T)^{-\frac{1}{T}} \\ &= \prod_{t=1}^{T} P(w_t \mid w_{t-1}, \ldots, w_1)^{-\frac{1}{T}} \end{aligned}$$

- In the case of bigram model:

$$PP(W) = \prod_{t=1}^{T} P(w_t \mid w_{t-1})^{-\frac{1}{T}}$$

Perplexity and log likelihood

- The logarithm of the perplexity is equal to:

$$\log PP(w) = \log \left(\prod_{t=1}^{T} P(w_t \mid w_{t-1}, \ldots, w_1)^{-\frac{1}{T}} \right)$$

$$\log PP(W) = -\frac{1}{T} \sum_{t=1}^{T} \log P(w_t \mid w_{t-1}, \ldots, w_1)$$

- It is the negative log-likelihood of the sequence

- **In practice:** use second expression, then take the exp
  Avoid numerical underflow

Example of Perplexity

	Unigram	Bigram	Trigram
PP	962	170	109

Lower perplexity means better model
As expected, better model with longer n-grams

On the WJS dataset [Training = 38M tokens, Testing = 1.5M tokens]

Count based language model

- *n*-gram based language model works well with "enough data"
- But does not generalize well
- **Can we use machine learning instead?**

Machine learning and language modeling

Machine learning for language model

- We have an evaluation setting for ML
- Can we cast language modeling as a machine learning problem?

Preliminaries

- Supervised classification:
  - **Supervision**: Each input X has a fixed given output Y
  - **Classification**: Y represents a class label among k possibilities

- Language modeling:
  - The input X is the subset of the previous tokens $(w_1, \ldots, w_{t-1})$
  - The output Y is the current token w_t
  - The token w_t is a class label among V possibilities

$\rightarrow$ **Language modeling is a supervised classification problem**

Preliminaries: what loss function?

- Intrisic measure for language model: perplexity
- The log of the perplexity is the negative log-likelihood
- Minimizing **negative log-likelihood** optimizes for the right criterion!

Preliminaries: how to transform words as vectors?

- Assumption: fixed vocabulary of V words

Preliminaries: how to transform words as vectors?

- Assumption: fixed vocabulary of V words
- Word i maps to a V-dimensional vector w_i:

$$\mathrm{w}_i[j] = \begin{cases} 1 & \text{if } j = i, \\ 0 & \text{otherwise} \end{cases}$$

Preliminaries: how to transform words as vectors?

- Assumption: fixed vocabulary of V words
- Word i maps to a V-dimensional vector w_i:

$$\mathrm{w}_i[j] = \begin{cases} 1 & \text{if } j = i, \\ 0 & \text{otherwise} \end{cases}$$

- These word vectors are:
  - independent: $\mathrm{w}_i^T \mathrm{w}_j = 0$ if $i \neq j$
  - normalized: $\mathrm{w}_i^T \mathrm{w}_i = 1$

Preliminaries: how to transform words as vectors?

- Assumption: fixed vocabulary of V words
- Word i maps to a V-dimensional vector w_i:

$$\mathrm{w}_i[j] = \begin{cases} 1 & \text{if } j = i, \\ 0 & \text{otherwise} \end{cases}$$

- These word vectors are:
    - independent: $\mathrm{w}_i^T \mathrm{w}_j = 0$ if $i \neq j$
    - normalized: $\mathrm{w}_i^T \mathrm{w}_i = 1$

- We call this representation "one-hot vectors"
- w_t = one-hot vector of word at t-th position in sentence

Our first linear model for bigrams

- Input = one-hot vector of previous word: $\mathrm{x}_t = \mathrm{w}_{t-1}$
- Output = one-hot vector of next word: $y_t = \mathrm{w}_t$

Our first linear model for bigrams

- Input = one-hot vector of previous word: $\mathrm{x}_t = \mathrm{w}_{t-1}$
- Output = one-hot vector of next word: $y_t = \mathrm{w}_t$

- **Linear model** $\mathrm{z} = \mathrm{A}\mathrm{x}$
- Build a probability over all possible words:

$$f(\mathrm{z})[k] = \frac{\exp(\mathrm{z}[k])}{\sum_{i=1}^{V} \exp(\mathrm{z}[i])}$$

Our first linear model for bigrams

- Input = one-hot vector of previous word: $\mathrm{x}_t = \mathrm{w}_{t-1}$
- Output = one-hot vector of next word: $y_t = \mathrm{w}_t$

- **Linear model** $\mathrm{z} = \mathrm{A}\mathrm{x}$
- Build a probability over all possible words:

$$f(\mathrm{z})[k] = \frac{\exp(\mathrm{z}[k])}{\sum_{i=1}^{V} \exp(\mathrm{z}[i])}$$

- A cross-entropy loss: $\ell(\mathrm{q}, \mathrm{p}) = -\mathrm{q}^T \log(\mathrm{p})$

Our first linear model for bigrams

- Input = one-hot vector of previous word: $\mathrm{x}_t = \mathrm{w}_{t-1}$
- Output = one-hot vector of next word: $y_t = \mathrm{w}_t$

- **Linear model** $\mathrm{z} = \mathrm{A}\mathrm{x}$
- Build a probability over all possible words:

$$f(\mathrm{z})[k] = \frac{\exp(\mathrm{z}[k])}{\sum_{i=1}^{V} \exp(\mathrm{z}[i])}$$

- A cross-entropy loss: $\ell(\mathrm{q}, \mathrm{p}) = -\mathrm{q}^T \log(\mathrm{p})$
- Learning linear bigram model:

$$\min_{\mathrm{A} \in \mathbb{R}^{V \times V}} \; \frac{1}{T} \sum_{t=1}^{T} \ell(\mathrm{y}_t, f(\mathrm{A}\mathrm{x}_t))$$

Pros of linear models over n-grams

$$\min_{A \in \mathbb{R}^{V \times V}} \frac{1}{T} \sum_{t=1}^{T} \ell(y_t, f(Ax_t))$$

- Can learn the same statistics as those in the n-gram models
- We can put additional features into x_t (e.g. from WordNet)
- Simple to implement

Limitations of linear models

$$\min_{\mathsf{A}\in\mathbb{R}^{V\times V}} \frac{1}{T}\sum_{t=1}^{T}\ell(\mathsf{y}_t, \mathsf{A}\mathsf{x}_t)$$

- The matrix A is $O(V^2)$
- Example: Penn Treebank $V = 10\text{k} \rightarrow 100,000,000$ parameters
- Difficult and slow to scale to longer n-grams

Limitations of linear models

- What if we replace A by two smaller matrices B and C?

Limitations of linear models

- What if we replace A by two smaller matrices B and C?

$$\min_{\mathrm{B,\ C}} \frac{1}{T} \sum_{t=1}^{T} \ell(\mathrm{y}_t, \mathrm{CBx}_t)$$

with B and C^T of dimension $V \times K$ ($K << V$)?

Limitations of linear models

- What if we replace A by two smaller matrices B and C?

$$\min_{\mathrm{B,\ C}} \frac{1}{T}\sum_{t=1}^{T}\ell(\mathrm{y}_t, \mathrm{CBx}_t)$$

with B and C^T of dimension $V \times K$ ($K << V$)?

- Bad: Not convex anymore, no guarantee on solution

Limitations of linear models

- What if we replace A by two smaller matrices B and C?

$$\min_{\text{B, C}} \frac{1}{T} \sum_{t=1}^{T} \ell(\mathrm{y}_t, \mathrm{CBx}_t)$$

with B and C^T of dimension $V \times K$ ($K << V$)?

- Bad: Not convex anymore, no guarantee on solution
- Good: fits in memory and faster to run!

Limitations of linear models

- What if we replace A by two smaller matrices B and C?

$$\min_{\mathrm{B,\ C}} \frac{1}{T} \sum_{t=1}^{T} \ell(\mathrm{y}_t, \mathrm{CBx}_t)$$

with B and C^T of dimension $V \times K$ ($K << V$)?

- Bad: Not convex anymore, no guarantee on solution
- Good: fits in memory and faster to run!

- But if not convex: why are we keeping the model linear?

Neural bigram model

- Feedforward network:

$$\begin{aligned} \mathsf{h}_{t-1} &= \sigma(\mathsf{A}\mathsf{w}_{t-1}) \\ \mathsf{p}_t &= f(\mathsf{B}\mathsf{h}_{t-1}) \end{aligned}$$

$\sigma(x) = 1/(1 + \exp(-x))$ pointwise sigmoid function

Neural bigram model

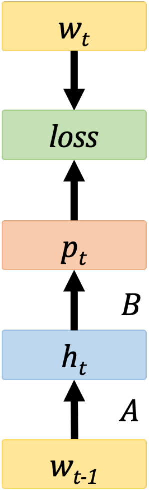
- Feedforward network:

$$\begin{aligned} \mathrm{h}_{t-1} &= \sigma(\mathrm{A}\mathrm{w}_{t-1}) \\ \mathrm{p}_t &= f(\mathrm{B}\mathrm{h}_{t-1}) \end{aligned}$$

$\sigma(x) = 1/(1 + \exp(-x))$ pointwise sigmoid function

- A: $V \times H$ matrix; B: $H \times V$ matrix
- $H \ll V$
- Minimization problem:

$$\min_{\mathrm{A},\,\mathrm{B}} \frac{1}{T} \sum_{t=1}^{T} \ell(\mathrm{w}_t, f(\mathrm{B}\sigma(\mathrm{A}\mathrm{w}_{t-1})))$$



Neural n-gram model

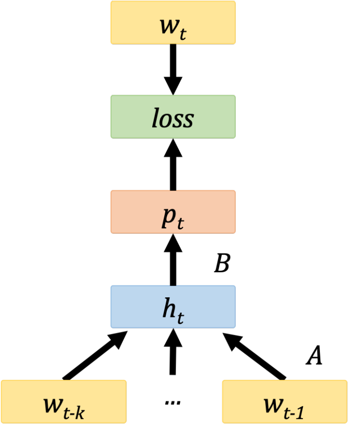
Generalization to any fixed n-gram size:

- The input is the contactenation of previous words:

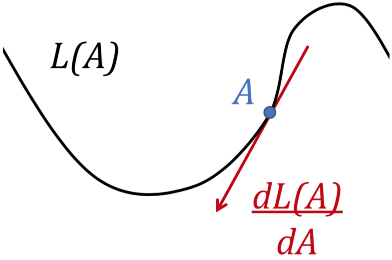
$$\mathrm{x}_t = [\mathrm{w}_{t-n+1}, \dots, \mathrm{w}_{t-1}]$$

- A: $nV \times H$ matrix
- Minimization problem:

$$\min_{\mathrm{A},\,\mathrm{B}} \frac{1}{T} \sum_{t=1}^{T} \ell(w_t, f(\mathrm{B}\sigma(\mathrm{A}\mathrm{x}_t)))$$



Neural n-gram model: training



- Loss function: $L(A,B) = \frac{1}{T}\sum_{t=1}^{T} \ell(w_t, f(\mathrm{B}\sigma(\mathrm{A}\mathrm{x}_t)))$
- This loss is differentiable in A and B
- Minimize the loss by updating parameters in direction of the gradient

Neural n-gram model: training

- Gradient descent:
  - Compute full loss $L(\mathsf{A}, \mathsf{B})$
  - Update parameters:

$$\mathsf{A} \leftarrow \mathsf{A} - \eta \frac{\partial L}{\partial \mathsf{A}}$$

  - $\eta > 0$ is the learning rate
- Stochastic gradient descent (SGD):
  - Instead of gradient on the full loss $L(\mathsf{A}, \mathsf{B})$
  - Randomly sample an example t
  - Partial loss

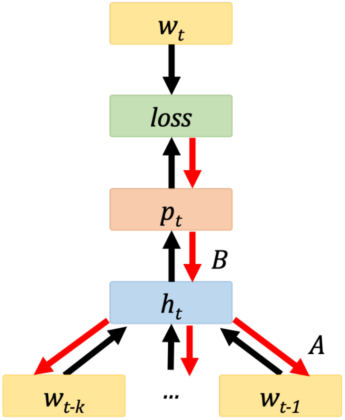
$$L_t(\mathsf{A}, \mathsf{B}) = \ell(y_t, f(\mathsf{B}\sigma(\mathsf{A}\mathsf{x}_t)))$$

  - Update parameters:

$$\mathsf{A} \leftarrow \mathsf{A} - \eta \frac{\partial L_t}{\partial \mathsf{A}}$$

Computing the gradient with backpropagation

- Compute gradient with **backpropagation**
- Compute error made by network when predicting next word
- Propagate error back to all parameters in network



Neural *n*-gram model: backpropagation

- We have $\mathsf{z} = \mathsf{B}\sigma(\mathsf{Ax})$ and $\mathsf{p} = f(\mathsf{z})$
- Loss for one example: $\ell(\mathsf{w}, \mathsf{p}) = \ell(\mathsf{w}, f(\mathsf{B}\sigma(\mathsf{Ax})))$

Neural n-gram model: backpropagation

- We have $\mathsf{z} = \mathsf{B}\sigma(\mathsf{Ax})$ and $\mathsf{p} = f(\mathsf{z})$
- Loss for one example: $\ell(\mathsf{w}, \mathsf{p}) = \ell(\mathsf{w}, f(\mathsf{B}\sigma(\mathsf{Ax})))$
- The gradient of the loss w.r.t. B with chain rule:

$$\frac{\partial \ell(\mathsf{w}, \mathsf{p})}{\partial \mathsf{B}} =$$

Neural n-gram model: backpropagation

- We have $\mathrm{z} = \mathrm{B}\sigma(\mathrm{Ax})$ and $\mathrm{p} = f(\mathrm{z})$
- Loss for one example: $\ell(\mathrm{w}, \mathrm{p}) = \ell(\mathrm{w}, f(\mathrm{B}\sigma(\mathrm{Ax})))$
- The gradient of the loss w.r.t. B with chain rule:

$$\frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{B}} = \frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{p}} \frac{\partial \mathrm{p}}{\partial \mathrm{B}} + \frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{w}} \frac{\partial \mathrm{w}}{\partial \mathrm{B}}$$

Neural *n*-gram model: backpropagation

- We have $\mathrm{z} = \mathrm{B}\sigma(\mathrm{Ax})$ and $\mathrm{p} = f(\mathrm{z})$
- Loss for one example: $\ell(\mathrm{w}, \mathrm{p}) = \ell(\mathrm{w}, f(\mathrm{B}\sigma(\mathrm{Ax})))$
- The gradient of the loss w.r.t. B with chain rule:

$$\frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{B}} = \frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{p}} \frac{\partial \mathrm{p}}{\partial \mathrm{B}} + \frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{w}} \frac{\partial \mathrm{w}}{\partial \mathrm{B}}$$

Neural *n*-gram model: backpropagation

- We have $\mathsf{z} = \mathsf{B}\sigma(\mathsf{Ax})$ and $\mathsf{p} = f(\mathsf{z})$
- Loss for one example: $\ell(\mathsf{w}, \mathsf{p}) = \ell(\mathsf{w}, f(\mathsf{B}\sigma(\mathsf{Ax})))$
- The gradient of the loss w.r.t. B with chain rule:

$$\frac{\partial \ell(\mathsf{w}, \mathsf{p})}{\partial \mathsf{B}} = \frac{\partial \ell(\mathsf{w}, \mathsf{p})}{\partial \mathsf{p}} \frac{\partial \mathsf{p}}{\partial \mathsf{B}} + \frac{\partial \ell(\mathsf{w}, \mathsf{p})}{\partial \mathsf{w}} \frac{\partial \mathsf{w}}{\partial \mathsf{B}}$$

Neural *n*-gram model: backpropagation

- We have $\mathrm{z} = \mathrm{B}\sigma(\mathrm{Ax})$ and $\mathrm{p} = f(\mathrm{z})$
- Loss for one example: $\ell(\mathrm{w}, \mathrm{p}) = \ell(\mathrm{w}, f(\mathrm{B}\sigma(\mathrm{Ax})))$
- The gradient of the loss w.r.t. B with chain rule:

$$\begin{aligned}\frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{B}} &= \frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{p}} \frac{\partial \mathrm{p}}{\partial \mathrm{B}} + \frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{w}} \frac{\partial \mathrm{w}}{\partial \mathrm{B}} \\ &= \frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{p}} \frac{\partial f(\mathrm{z})}{\partial \mathrm{B}} \qquad \text{where} \quad \mathrm{p} = f(\mathrm{z})\end{aligned}$$

Neural *n*-gram model: backpropagation

- We have $\mathrm{z} = \mathrm{B}\sigma(\mathrm{Ax})$ and $\mathrm{p} = f(\mathrm{z})$
- Loss for one example: $\ell(\mathrm{w}, \mathrm{p}) = \ell(\mathrm{w}, f(\mathrm{B}\sigma(\mathrm{Ax})))$
- The gradient of the loss w.r.t. B with chain rule:

$$\frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{B}} = \frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{p}} \frac{\partial \mathrm{p}}{\partial \mathrm{B}} + \frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{w}} \frac{\partial \mathrm{w}}{\partial \mathrm{B}}$$

$$= \frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{p}} \frac{\partial f(\mathrm{z})}{\partial \mathrm{B}} \qquad \text{where} \quad \mathrm{p} = f(\mathrm{z})$$

Neural n-gram model: backpropagation

- We have $\mathsf{z} = \mathsf{B}\sigma(\mathsf{Ax})$ and $\mathsf{p} = f(\mathsf{z})$
- Loss for one example: $\ell(\mathsf{w}, \mathsf{p}) = \ell(\mathsf{w}, f(\mathsf{B}\sigma(\mathsf{Ax})))$
- The gradient of the loss w.r.t. B with chain rule:

$$
\begin{aligned}
\frac{\partial \ell(\mathsf{w},\mathsf{p})}{\partial \mathsf{B}} &= \frac{\partial \ell(\mathsf{w},\mathsf{p})}{\partial \mathsf{p}} \frac{\partial \mathsf{p}}{\partial \mathsf{B}} + \frac{\partial \ell(\mathsf{w},\mathsf{p})}{\partial \mathsf{w}} \frac{\partial \mathsf{w}}{\partial \mathsf{B}} \\
&= \frac{\partial \ell(\mathsf{w},\mathsf{p})}{\partial \mathsf{p}} \frac{\partial f(\mathsf{z})}{\partial \mathsf{B}} && \text{where} \quad \mathsf{p} = f(\mathsf{z}) \\
&= \frac{\partial \ell(\mathsf{w},\mathsf{p})}{\partial \mathsf{p}} \frac{\partial f(\mathsf{z})}{\partial \mathsf{z}} \frac{\partial \mathsf{z}}{\partial \mathsf{B}}
\end{aligned}
$$

Neural n-gram model: backpropagation

- We have $\mathrm{z} = \mathrm{B}\sigma(\mathrm{Ax})$ and $\mathrm{p} = f(\mathrm{z})$
- Loss for one example: $\ell(\mathrm{w}, \mathrm{p}) = \ell(\mathrm{w}, f(\mathrm{B}\sigma(\mathrm{Ax})))$
- The gradient of the loss w.r.t. B with chain rule:

$$
\begin{aligned}
\frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{B}} &= \frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{p}} \frac{\partial \mathrm{p}}{\partial \mathrm{B}} + \frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{w}} \frac{\partial \mathrm{w}}{\partial \mathrm{B}} \\
&= \frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{p}} \frac{\partial f(\mathrm{z})}{\partial \mathrm{B}} \qquad\qquad \text{where} \quad \mathrm{p} = f(\mathrm{z}) \\
&= \frac{\partial \ell(\mathrm{w}, \mathrm{p})}{\partial \mathrm{p}} \frac{\partial f(\mathrm{z})}{\partial \mathrm{z}} \frac{\partial \mathrm{z}}{\partial \mathrm{B}}
\end{aligned}
$$

Neural *n*-gram model: backpropagation

- We have $\mathrm{z} = \mathrm{B}\sigma(\mathrm{Ax})$ and $\mathrm{p} = f(\mathrm{z})$
- Loss for one example: $\ell(\mathrm{w}, \mathrm{p}) = \ell(\mathrm{w}, f(\mathrm{B}\sigma(\mathrm{Ax})))$
- The gradient of the loss w.r.t. B with chain rule:

$$
\begin{aligned}
\frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{B}} &= \frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{p}} \frac{\partial \mathrm{p}}{\partial \mathrm{B}} + \frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{w}} \frac{\partial \mathrm{w}}{\partial \mathrm{B}} \\
&= \frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{p}} \frac{\partial f(\mathrm{z})}{\partial \mathrm{B}} && \text{where} \quad \mathrm{p} = f(\mathrm{z}) \\
&= \frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{p}} \frac{\partial f(\mathrm{z})}{\partial \mathrm{z}} \frac{\partial \mathrm{z}}{\partial \mathrm{B}} \\
&= \frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{p}} \frac{\partial f(\mathrm{z})}{\partial \mathrm{z}} \frac{\partial\left(\mathrm{B}\sigma(\mathrm{Ax})\right)}{\partial \mathrm{B}} && \text{where} \quad \mathrm{z} = \mathrm{B}\sigma(\mathrm{Ax})
\end{aligned}
$$

Neural *n*-gram model: backpropagation

- We have $\mathrm{z} = \mathrm{B}\sigma(\mathrm{Ax})$ and $\mathrm{p} = f(\mathrm{z})$
- Loss for one example: $\ell(\mathrm{w}, \mathrm{p}) = \ell(\mathrm{w}, f(\mathrm{B}\sigma(\mathrm{Ax})))$
- The gradient of the loss w.r.t. B with chain rule:

$$
\begin{aligned}
\frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{B}} &= \frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{p}} \frac{\partial \mathrm{p}}{\partial \mathrm{B}} + \frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{w}} \frac{\partial \mathrm{w}}{\partial \mathrm{B}} \\
&= \frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{p}} \frac{\partial f(\mathrm{z})}{\partial \mathrm{B}} && \text{where} \quad \mathrm{p} = f(\mathrm{z}) \\
&= \frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{p}} \frac{\partial f(\mathrm{z})}{\partial \mathrm{z}} \frac{\partial \mathrm{z}}{\partial \mathrm{B}} \\
&= \frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{p}} \frac{\partial f(\mathrm{z})}{\partial \mathrm{z}} \frac{\partial \left(\mathrm{B}\sigma(\mathrm{Ax})\right)}{\partial \mathrm{B}} && \text{where} \quad \mathrm{z} = \mathrm{B}\sigma(\mathrm{Ax})
\end{aligned}
$$

Neural *n*-gram model: backpropagation

- We have $\mathrm{z} = \mathrm{B}\sigma(\mathrm{Ax})$ and $\mathrm{p} = f(\mathrm{z})$
- Loss for one example: $\ell(\mathrm{w}, \mathrm{p}) = \ell(\mathrm{w}, f(\mathrm{B}\sigma(\mathrm{Ax})))$
- The gradient of the loss w.r.t. B with chain rule:

$$
\begin{aligned}
\frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{B}} &= \frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{p}} \frac{\partial \mathrm{p}}{\partial \mathrm{B}} + \frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{w}} \frac{\partial \mathrm{w}}{\partial \mathrm{B}} \\
&= \frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{p}} \frac{\partial f(\mathrm{z})}{\partial \mathrm{B}} && \text{where} \quad \mathrm{p} = f(\mathrm{z}) \\
&= \frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{p}} \frac{\partial f(\mathrm{z})}{\partial \mathrm{z}} \frac{\partial \mathrm{z}}{\partial \mathrm{B}} \\
&= \frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{p}} \frac{\partial f(\mathrm{z})}{\partial \mathrm{z}} \frac{\partial \left(\mathrm{B}\sigma(\mathrm{Ax})\right)}{\partial \mathrm{B}} && \text{where} \quad \mathrm{z} = \mathrm{B}\sigma(\mathrm{Ax}) \\
&= \frac{\partial \ell(\mathrm{w},\mathrm{p})}{\partial \mathrm{p}} \frac{\partial f(\mathrm{z})}{\partial \mathrm{z}} \sigma(\mathrm{Ax})
\end{aligned}
$$

Neural n-gram model: backpropagation

- Loss function:

$$\frac{1}{T}\sum_{t=1}^{T}\ell(\mathrm{w}_t, f(\mathrm{B}\sigma(\mathrm{A}\mathrm{x}_t)))$$

- The gradients are:

$$\begin{aligned}
\frac{\partial L}{\partial \mathrm{B}} &= \frac{1}{T}\sum_{t=1}^{T}\frac{\partial \ell}{\partial \mathrm{z}_t}\frac{\partial \mathrm{z}_t}{\partial \mathrm{B}} \\
\frac{\partial L}{\partial \mathrm{A}} &= \frac{1}{T}\sum_{t=1}^{T}\frac{\partial \ell}{\partial \mathrm{z}_t}\frac{\partial \mathrm{z}_t}{\partial \mathrm{h}_t}\frac{\partial \mathrm{h}_t}{\partial \mathrm{A}}
\end{aligned}$$

with $\mathrm{z}_t = \mathrm{B}\mathrm{h}_t$ and $\mathrm{h}_t = \sigma(\mathrm{A}\mathrm{x}_t)$

Neural n-gram model: backpropagation

- Loss function:

$$\frac{1}{T}\sum_{t=1}^{T}\ell(\mathrm{w}_t, f(\mathrm{B}\sigma(\mathrm{A}\mathrm{x}_t)))$$

- The gradients are:

$$\begin{aligned}\frac{\partial L}{\partial \mathrm{B}} &= \frac{1}{T}\sum_{t=1}^{T}\frac{\partial \ell}{\partial \mathrm{z}_t}\frac{\partial \mathrm{z}_t}{\partial \mathrm{B}}\\ \frac{\partial L}{\partial \mathrm{A}} &= \frac{1}{T}\sum_{t=1}^{T}\frac{\partial \ell}{\partial \mathrm{z}_t}\frac{\partial \mathrm{z}_t}{\partial \mathrm{h}_t}\frac{\partial \mathrm{h}_t}{\partial \mathrm{A}}\end{aligned}$$

with $\mathrm{z}_t = \mathrm{B}\mathrm{h}_t$ and $\mathrm{h}_t = \sigma(\mathrm{A}\mathrm{x}_t)$

- **Intermediate computations shared between different gradients**

Neural n-gram model: backpropagation

- Loss function:

$$\frac{1}{T}\sum_{t=1}^{T}\ell(\mathrm{w}_t, f(\mathrm{B}\sigma(\mathrm{A}\mathrm{x}_t)))$$

- The gradients are:

$$\begin{aligned}
\frac{\partial L}{\partial \mathrm{B}} &= \frac{1}{T}\sum_{t=1}^{T}\frac{\partial \ell}{\partial \mathrm{z}_t}\frac{\partial \mathrm{z}_t}{\partial \mathrm{B}} \\
\frac{\partial L}{\partial \mathrm{A}} &= \frac{1}{T}\sum_{t=1}^{T}\frac{\partial \ell}{\partial \mathrm{z}_t}\frac{\partial \mathrm{z}_t}{\partial \mathrm{h}_t}\frac{\partial \mathrm{h}_t}{\partial \mathrm{A}}
\end{aligned}$$

with $\mathrm{z}_t = \mathrm{B}\mathrm{h}_t$ and $\mathrm{h}_t = \sigma(\mathrm{A}\mathrm{x}_t)$

- Intermediate computations shared between different gradients

Backpropagation = chain rule + storing computation

Regularization: dropout

- Specialized units cause overfitting

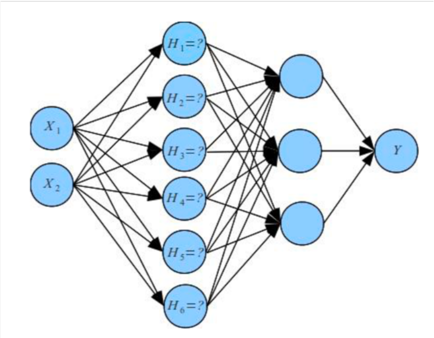
- **Idea** force model to work even when some units are removed

- Same as activation mask over units
- we replace h_t by:

$$\hat{\mathsf{h}}_t = \mathsf{h}_t \odot \mathsf{m}_t$$

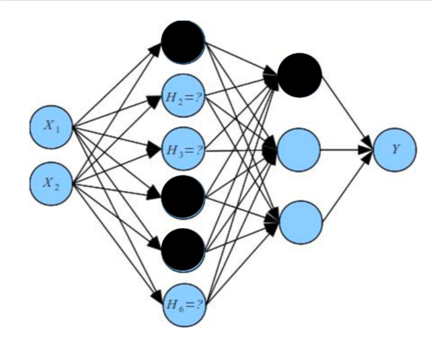
  where m_t is a binary mask vector.
- This binary mask is randomly drawn for each time step

Regularization: dropout



- Units are dropped:
  - with probability p.
  - independently
  - only during training

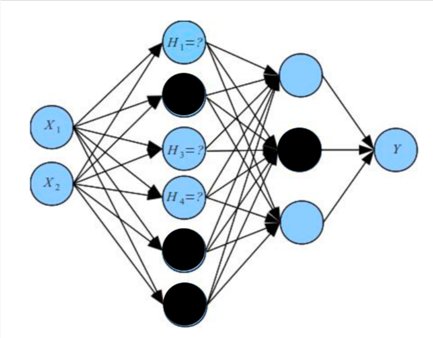
Regularization: dropout



Iteration 1

- Units are dropped:
  - with probability p.
  - independently
  - only during training
- Dropped unit are in black

Regularization: dropout



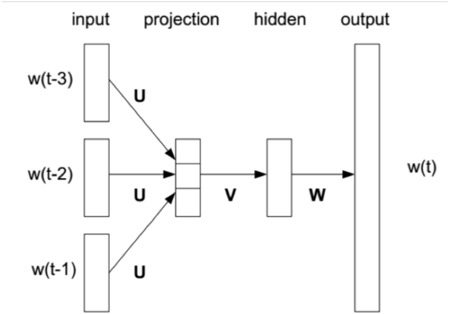
Iteration 2

- Units are dropped:
  - with probability p.
  - independently
  - only during training
- Dropped unit are in black

Dealing with large vocabulary

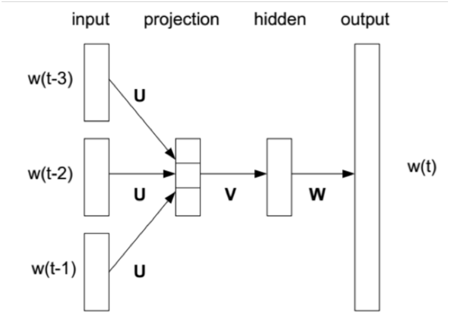
- At each time step, we compute probability over a vocabulary
- If the vocabulary size V is big, computing this probability is very slow
- Similar to text classification with large number of classes
- **Solution:** use class-based softmax (see prev. lecture)

The neural *n*-gram model from Bengio et al. (2003)



- Their model has one more hidden layer to embed one-hot vectors into low dimensional space
- Resulting vector $\mathrm{U}w_t$ is a distributed word representation
- These representations are passed through a feedforward network

Neural *n*-gram model: example



- The equations are:

$$
\begin{aligned}
x_{t-k} &= \sigma(Uw_{t-k}) && \text{(distributed representation)} \\
h_{t-1} &= \sigma(V[x_{t-3}, x_{t-2}, x_{t-1}]) && \text{(hidden representation)} \\
p_t &= f(Wh_{t-1}) && \text{(output probability)}
\end{aligned}
$$

Neural n-gram model: example

Model	Perplexity
Kneser-Ney 5-gram	141
Neural n-gram (Bengio et al., 2003)	140

- Neural n-gram perform as as well as Kneser-Ney 5-gram
- Requires much less parameters

Penn Treebank dataset

Neural *n*-gram model: pros and cons

Pros:

- Performs as well as best count based language models
- Need less parameters
- Naturally generalize to unseen *n*-grams

Cons:

- Number of parameters grows with the window size of *n*-gram
- Memory of the past limited to *n*-gram window size

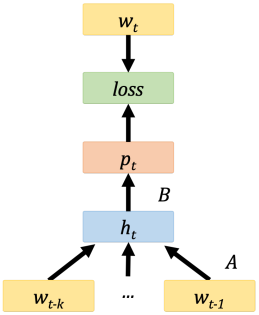
Recurrent Neural Network (RNN)

Recurrent Neural Network

- Recurrent network: Keep memory of past in the hidden variables

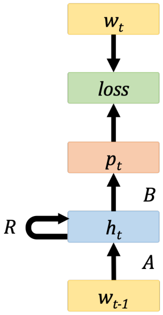
Feedforward

$$h_{t-1} = \sigma\left(A[w_{t-k}, \dots, w_{t-1}]\right)$$
$$p_t = f(Bh_{t-1})$$

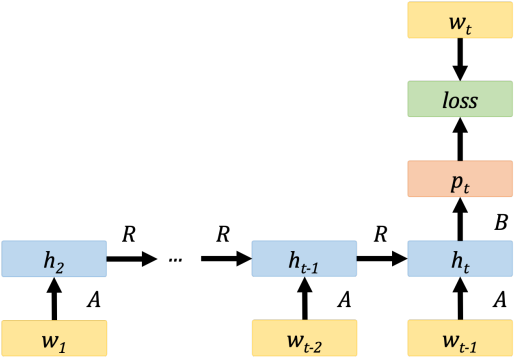


Recurrent Network

$$h_{t-1} = \sigma\left(Aw_{t-1} + Rh_{t-2}\right)$$
$$p_t = f(Bh_{t-1})$$

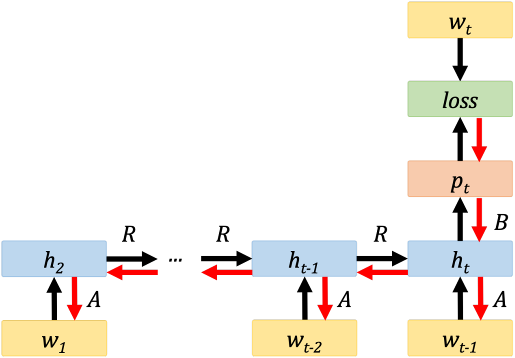


Recurrent Neural Network



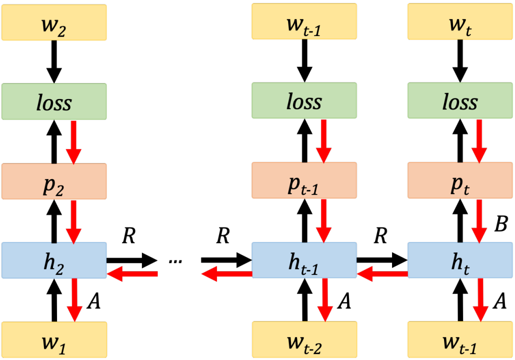
- Recurrent equation: $\mathrm{h}_t = \sigma\left(\mathrm{A}[\mathrm{h}_{t-1}, \mathrm{w}_t]\right)$
- Unfold over time: **very deep feedforward with weight sharing**
- Potentially capture long term dependencies

Recurrent Neural Network: training



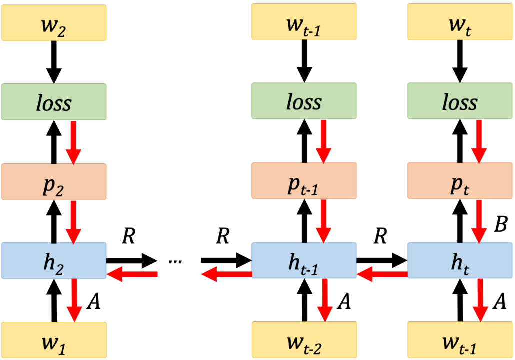
- **Backpropagation through time (BPTT)**: same as backpropagation through a very deepfeedforward network

Recurrent Neural Network: training



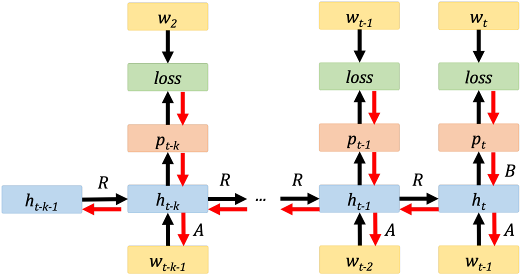
- **batch BPTT**: forward/backward for many words simultaneously

Recurrent Neural Network: training



- **Problem with BPTT**: Computing 1 gradient is $O(T)$. Too slow.

Recurrent Neural Network: training



- **Truncated BPTT**: Go back in time for k step: $O(k)$.

RNN: results

Model	Perplexity
Kneser-Ney 5-gram	141
Neural *n*-gram (Bengio et al., 2003)	140
RNN	**125**

- Penn Treebank dataset
- RNN outperforms *n*-gram models
- Faster at test time: does not depend on *n*-gram length

RNN: Vanishing and exploding gradients

- Consider the partial derivatives of the gradient:

$$\frac{\partial \ell(\mathsf{w}_t, \mathsf{p}_t)}{\partial \mathsf{h}_2} = \frac{\partial \ell(\mathsf{w}_t, \mathsf{p}_t)}{\partial \mathsf{p}_t} \frac{\partial \mathsf{p}_t}{\partial \mathsf{h}_{t-1}} \underbrace{\frac{\partial \mathsf{h}_{t-1}}{\partial \mathsf{h}_{t-2}} \cdots \frac{\partial \mathsf{h}_3}{\partial \mathsf{h}_2}}_{T\,\text{terms}}$$

RNN: Vanishing and exploding gradients

- Consider the partial derivatives of the gradient:

$$\frac{\partial \ell(\mathsf{w}_t, \mathsf{p}_t)}{\partial \mathsf{h}_2} = \frac{\partial \ell(\mathsf{w}_t, \mathsf{p}_t)}{\partial \mathsf{p}_t} \frac{\partial \mathsf{p}_t}{\partial \mathsf{h}_{t-1}} \underbrace{\frac{\partial \mathsf{h}_{t-1}}{\partial \mathsf{h}_{t-2}} \cdots \frac{\partial \mathsf{h}_3}{\partial \mathsf{h}_2}}_{T \text{ terms}}$$

- Each term: $\frac{\partial \mathsf{h}_k}{\partial \mathsf{h}_{k-1}} = \mathrm{diag}(\sigma'(A\mathsf{w}_k + \mathsf{R}\mathsf{h}_{k-1}))\mathsf{R}$

RNN: Vanishing and exploding gradients

- Consider the partial derivatives of the gradient:

$$\frac{\partial \ell(\mathsf{w}_t, \mathsf{p}_t)}{\partial \mathsf{h}_2} = \frac{\partial \ell(\mathsf{w}_t, \mathsf{p}_t)}{\partial \mathsf{p}_t} \frac{\partial \mathsf{p}_t}{\partial \mathsf{h}_{t-1}} \underbrace{\frac{\partial \mathsf{h}_{t-1}}{\partial \mathsf{h}_{t-2}} \cdots \frac{\partial \mathsf{h}_3}{\partial \mathsf{h}_2}}_{T\,\text{terms}}$$

- Each term: $\frac{\partial \mathsf{h}_k}{\partial \mathsf{h}_{k-1}} = \mathrm{diag}(\sigma'(A\mathsf{w}_k + \mathsf{R}\mathsf{h}_{k-1}))\mathsf{R}$
- So the gradient is a serie of multiplication of R and $\mathrm{diag}(\sigma')$:

$$\frac{\partial \ell(\mathsf{w}_t, \mathsf{p}_t)}{\partial \mathsf{h}_2} = \frac{\partial \ell(\mathsf{w}_t, \mathsf{p}_t)}{\partial \mathsf{p}_t} \frac{\partial \mathsf{p}_t}{\partial \mathsf{h}_{t-1}} \prod_t \left[\mathrm{diag}(\sigma'(\mathsf{z}_t)\mathsf{R}\right]$$

RNN: Exploding gradient

- The matrix R are not direclty multiplied in the partial derivatives
- Impossible to lower bound partial derivative norms
- Popular incorrect argument:

$$\prod_k \left[\mathrm{diag}(\sigma'(\mathrm{z}_k))\mathrm{R}\right] \approx \mathrm{R}^k$$

- we cannot lowerbound $\mathrm{diag}(\sigma'(\mathrm{z}_k))$ nor permute it with R
- Even if we could, R^k is not informative (e.g., nilpotent matrices)

- However the intuition is still correct: if the maximum singluar value of R is such that $\lambda_{\max} \gg 1$: **the gradient might explode**

RNN: Exploding gradient

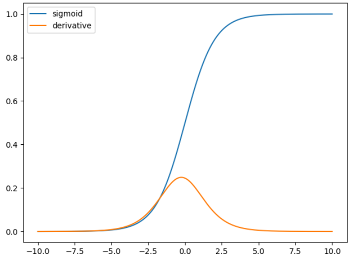
- Consequence: hard to learn a RNN with gradient descent
- **Exploding gradient is an optimization problem**
- Simple hack to fix this problem: **gradient clipping**:

$$G = \min(\mu, \|G\|)\frac{G}{\|G\|}$$

  with $\mu > 0$

- it bounds the norm of a gradient G to be at most μ

RNN: Vanishing gradient



- The derivative of σ is mostly close to 0: each multiplication by $\text{diag}(\sigma')$ likely adds 0 to the partial derivative

RNN: Vanishing gradient

- Putting R and σ' together, we have:

$$\|\mathrm{diag}(\sigma'(\mathrm{z}_k))\mathrm{R}\| \leq \max_x |\sigma'(x)||\lambda_{\mathrm{max}}| \leq 0.25|\lambda_{\mathrm{max}}|$$

RNN: Vanishing gradient

- Putting R and σ' together, we have:

$$\|\mathrm{diag}(\sigma'(\mathrm{z}_k))\mathrm{R}\| \leq \max_x |\sigma'(x)||\lambda_{\mathrm{max}}| \leq 0.25|\lambda_{\mathrm{max}}|$$

- Partial derivative is such that

$$\|\prod_k \mathrm{diag}(\sigma'(\mathrm{z}_k))\mathrm{R}\| \leq 0.25^k \lambda_{\mathrm{max}}^k$$

- If $\lambda_{\mathrm{max}} < 4$: **partial derivatives vanish to 0 rapidly.**
- The bound depends on the non-linearity

RNN: Vanishing gradient

- Consequence of vanishing gradient: long distance information cannot be retained by an RNN
- The flow of information decays exponentially $\rightarrow$ short memory span
- **Vanishing gradient: model problem, not optimization problem**
- Solutions require a change in the structure of the model

Long Short Term Memory (LSTM)

Long Short Term Memory (LSTM)

- Vanilla RNN:

$$\mathsf{h}_t = \sigma(A\mathsf{w}_t + \mathsf{R}\mathsf{h}_{t-1})$$

Long Short Term Memory (LSTM)

- Vanilla RNN:

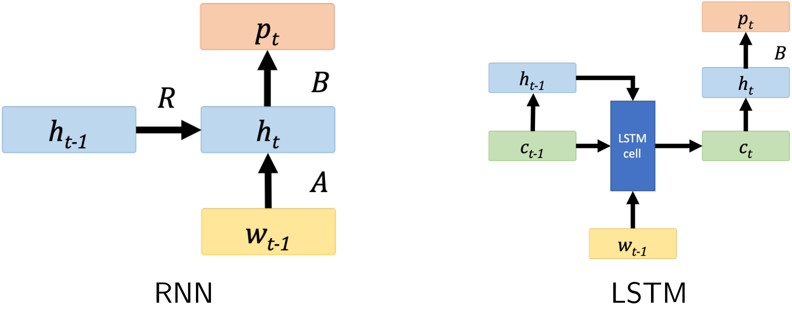
$$\mathsf{h}_t = \sigma(A\mathsf{w}_t + \mathsf{R}\mathsf{h}_{t-1})$$

- Any function could work:

$$\mathsf{h}_t = \phi(\mathsf{w}_t, \mathsf{h}_{t-1})$$

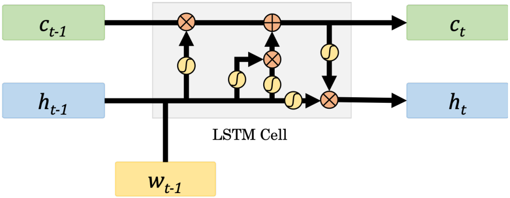
- Preferably ϕ should be mostly differentiable and reduces the vanishing gradient problem

Long Short Term Memory (LSTM)



- LSTM introduces an additional hidden variable c_t called the "memory cell"

Long Short Term Memory (LSTM)

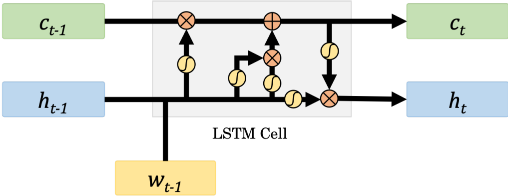


Inspired by "Understanding LSTM Networks", Olah, 2016.

- The LSTM equations are:

$$
\begin{aligned}
\mathrm{c}_t &= f_t \circ \mathrm{c}_{t-1} + i_t \circ \tanh(A\mathrm{w}_t + \mathrm{R}\mathrm{h}_{t-1}) \\
\mathrm{h}_t &= o_t \circ \tanh(W\mathrm{c}_t)
\end{aligned}
$$

Long Short Term Memory (LSTM)



Inspired by "Understanding LSTM Networks", Olah, 2016.

- The LSTM equations are:

$$
\begin{aligned}
\mathrm{c}_t &= f_t \circ \mathrm{c}_{t-1} + i_t \circ \tanh(A\mathrm{w}_t + \mathrm{R}\mathrm{h}_{t-1}) \\
\mathrm{h}_t &= o_t \circ \tanh(W\mathrm{c}_t)
\end{aligned}
$$

- with:

$$
\begin{aligned}
f_t &= \sigma(A_f \mathrm{w}_{t-1} + R_f \mathrm{h}_{t-1}) && \text{forget gate} \\
i_t &= \sigma(A_i \mathrm{w}_{t-1} + R_i \mathrm{h}_{t-1}) && \text{input gate} \\
o_t &= \sigma(A_o \mathrm{w}_{t-1} + R_o \mathrm{h}_{t-1}) && \text{output gate}
\end{aligned}
$$

Attempt at explaining LSTM

- The output gate is not crucial $\rightarrow$ we drop it from this explanation
- The equations are thus the following:

$$\begin{aligned} \mathrm{c}_t &= f_t \circ \mathrm{c}_{t-1} + i_t \circ \tanh(A\mathrm{w}_t + \mathrm{R}\mathrm{h}_{t-1}) \\ \mathrm{h}_t &= \tanh(W\mathrm{c}_t) \end{aligned}$$

- This way, h_t only depends on c_t

Attempt at explaining the memory cell update

- **This is an "hand-wavy" explanation of these equations**
- A standard RNN update is:

$$\mathrm{c}_t = \tanh(A\mathrm{w}_t + \mathrm{R}\mathrm{c}_{t-1})$$

Attempt at explaining the memory cell update

- **This is an "hand-wavy" explanation of these equations**
- A standard RNN update is:

$$c_t = \tanh(Aw_t + Rc_{t-1})$$

- A simple way to keep longer memory of past is to add a linear part:

$$c_t = c_{t-1} + \underbrace{\tanh(Aw_t + Rc_{t-1})}_{\text{Same as RNN}}$$

Attempt at explaining the memory cell update

- **This is an "hand-wavy" explanation of these equations**
- A standard RNN update is:

$$c_t = \tanh(Aw_t + Rc_{t-1})$$

- A simple way to keep longer memory of past is to add a linear part:

$$c_t = c_{t-1} + \underbrace{\tanh(Aw_t + Rc_{t-1})}_{\text{Same as RNN}}$$

- Let us unroll the computation over time:

$$c_t = \sum_{i=0}^{t} \tanh(Aw_i + Rc_{i-1})$$

Attempt at explaining the memory cell update

- **This is an "hand-wavy" explanation of these equations**
- A standard RNN update is:

$$c_t = \tanh(Aw_t + Rc_{t-1})$$

- A simple way to keep longer memory of past is to add a linear part:

$$c_t = c_{t-1} + \underbrace{\tanh(Aw_t + Rc_{t-1})}_{\text{Same as RNN}}$$

- Let us unroll the computation over time:

$$c_t = \sum_{i=0}^{t} \tanh(Aw_i + Rc_{i-1})$$

- The linear part allows more influence of past on the current update
- **Problem**: past information is "as important as recent one". After T step, a new word contribution is weighted as only $1/T$ at most.

LSTM: memory cell update

- Possible solution: use a discount factor:

$$\mathsf{c}_t = \eta \mathsf{c}_{t-1} + \tanh(A\mathsf{w}_t + \mathsf{R}\mathsf{c}_{t-1})$$

η should be in $[0, 1]$

LSTM: memory cell update

- Possible solution: use a discount factor:

$$\mathsf{c}_t = \eta \mathsf{c}_{t-1} + \tanh(A\mathsf{w}_t + \mathsf{R}\mathsf{c}_{t-1})$$

η should be in $[0, 1]$

- We now have:

$$\mathsf{c}_t = \sum_{i=0}^{t} \eta^{t-i} \tanh(A\mathsf{w}_i + \mathsf{R}\mathsf{c}_{i-1})$$

- **Problem**: This falls back to "vanishing gradient problem"

LSTM: memory cell update

- Instead, LSTM learns what to store and the importance of the past by learning the weighting:

$$\mathrm{c}_t = f(\mathrm{w}_t,\ \mathrm{c}_{t-1}) \circ \mathrm{c}_{t-1} + i(\mathrm{w}_t,\ \mathrm{c}_{t-1}) \circ \tanh(A\mathrm{w}_t + \mathrm{R}\mathrm{c}_{t-1})$$

- The forget gate weights the contribution of the past
- The input gates weights the contribution of the current word

LSTM: memory cell update

- So far, we have written the equation in terms of c_t

$$c_t = f(w_t,\ c_{t-1}) \circ c_{t-1} + i(w_t,\ c_{t-1}) \circ \tanh(Aw_t + Rc_{t-1})$$

LSTM: memory cell update

- So far, we have written the equation in terms of c_t

$$\mathrm{c}_t = f(\mathrm{w}_t,\ \mathrm{c}_{t-1}) \circ \mathrm{c}_{t-1} + i(\mathrm{w}_t,\ \mathrm{c}_{t-1}) \circ \tanh(A\mathrm{w}_t + \mathrm{R}\mathrm{c}_{t-1})$$

- But the correct equation is:

$$\mathrm{c}_t = f(\mathrm{w}_t,\ \mathrm{h}_{t-1}) \circ \mathrm{c}_{t-1} + i(\mathrm{w}_t,\ \mathrm{h}_{t-1}) \circ \tanh(A\mathrm{w}_t + \mathrm{R}\mathrm{h}_{t-1})$$

LSTM: memory cell update

- So far, we have written the equation in terms of c_t

$$c_t = f(w_t,\ c_{t-1}) \circ c_{t-1} + i(w_t,\ c_{t-1}) \circ \tanh(Aw_t + Rc_{t-1})$$

- But the correct equation is:

$$c_t = f(w_t,\ h_{t-1}) \circ c_{t-1} + i(w_t,\ h_{t-1}) \circ \tanh(Aw_t + Rh_{t-1})$$

- Why do we need two different variables?
- $h_t = \tanh(Wc_t) \rightarrow h_t$ is c_t rescaled to $[-1, 1]$:
- The benefits are:
  - Rescaling h_t avoids gradient explosion
  - Keeping c_t value unbounded allows to learn more patterns, e.g., allows to count

Counting in LSTM

- Counting means that a LSTM can do internally simple arithmetical operation (adding and substracting numbers)
- There are evidences that some memory cells can act as a counter
- This is very interesting for tasks:
  - Learning a latent parser
  - Checking parenthesis in a computer program
  - Storing length of a sentence

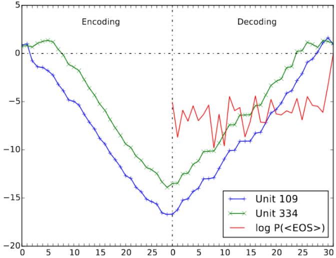


Figure: Evidence from Shi et al. (2016) that some LSTM cells store sentence length in a machine translation system.

LSTM: results

Model	Perplexity
Kneser-Ney 5-gram	141
Neural *n*-gram	140
RNN	125
LSTM	**115**

- Penn Treebank dataset
- LSTM outperforms RNN

Sentence Representation

From word to sentence representation

- **(Previous lecture) Word vectors:** map words to vectors
  Example: word2vec, fasttext, PPMI+SVD...

- **Goal:** Can we build similar representation for sentences?

From word to sentence representation

- **(Previous lecture) Word vectors:** map words to vectors
  Example: word2vec, fasttext, PPMI+SVD...

- **Goal:** Can we build similar representation for sentences?

- Several difficulties:
  - Sentences have variable length
  - There is infinite number of sentences, not words
  - Sentences are much richer than words

Simple sentence representation

- A sentence = sequence of words
- Each word has a distributed word vector: $\mathsf{w}_1, \ldots, \mathsf{w}_T$
- Average these vectors to form a sentence representation:

$$\mathsf{s} = \frac{1}{T} \sum_{t=1}^{T} \mathsf{w}_t$$

- This is a Bag of Words (BoW) representation

Simple sentence representation

Examples of extensions:

- Replace average with taking max value per dimension:

$$\mathrm{s}(i) = \max_{t \in [1,T]} \mathrm{w}_t(i)$$

- Add other features:
  - distributed representation of *n*-grams or subwords (e.g., fasttext)
  - features from WordNet...

- Make the word vectors depend on context

Simple sentence representation

Examples of extensions:

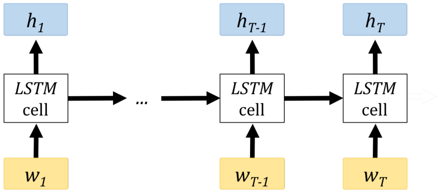
- Replace average with taking max value per dimension:

$$s(i) = \max_{t \in [1,T]} w_t(i)$$

- Add other features:
  - distributed representation of *n*-grams or subwords (e.g., fasttext)
  - features from WordNet...

- Make the word vectors depend on context

Using an LSTM for sentence representation

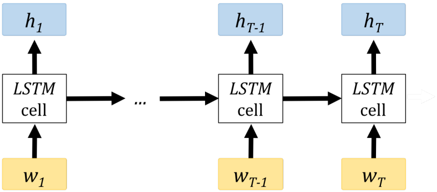


- Apply LSTM on sentence $\rightarrow$ sequence of word vectors $\mathrm{h}_1, \ldots, \mathrm{h}_T$

- Apply Bag-of-Word sentence representation:

$$\mathrm{s} = \frac{1}{T} \sum_{t=1}^{T} \mathrm{h}_t$$

Using an LSTM for sentence representation



- Apply LSTM on sentence $\rightarrow$ sequence of word vectors $\mathrm{h}_1, \ldots, \mathrm{h}_T$
- Apply Bag-of-Word sentence representation:

$$\mathrm{s} = \frac{1}{T} \sum_{t=1}^{T} \mathrm{h}_t$$

- **Problem:** These word representations depend on past, not future

Simple solution: Bidirectional LSTM (BiLSTM)

- **BiLSTM = 2 LSTMs running on opposite direction**
- **Past:** $\overrightarrow{LSTM}$ runs forward on sequence:

$$\overrightarrow{\mathrm{h}}_1, \ldots, \overrightarrow{\mathrm{h}}_T$$

- **Future:** $\overleftarrow{LSTM}$ runs backward on sequence:

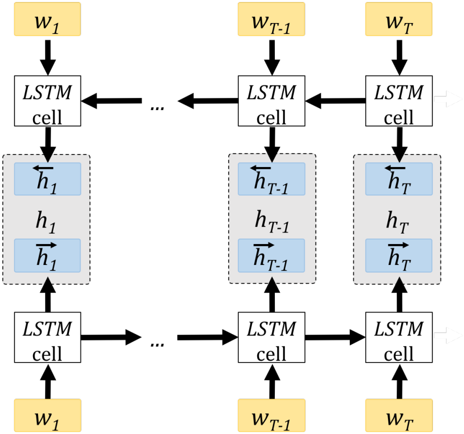
$$\overleftarrow{\mathrm{h}}_1, \ldots, \overleftarrow{\mathrm{h}}_T$$

- **Past+Future:** biLSTM is the concatenation of both:

$$\mathrm{h}_t = [\overrightarrow{\mathrm{h}}_t, \overleftarrow{\mathrm{h}}_t]$$

- Called "contextualized word vectors" (Peters et al., 2018).

Bidirectional LSTM (BiLSTM)



BoW from biLSTM: $\mathrm{s} = \frac{1}{T}\sum_{t=1}^{T}\mathrm{h}_t = \frac{1}{T}\sum_{t=1}^{T}[\overrightarrow{\mathrm{h}}_t, \overleftarrow{\mathrm{h}}_t]$

Training biLSTM with Language modeling

- Train both LSTMs independently:
  - The forward $\overrightarrow{LSTM}$ predicts upcoming word:

$$P_{\text{forward}}(w_t \mid w_{t-1}, \ldots, w_1)$$

  - The backward $\overleftarrow{LSTM}$ predicts previous word:

$$P_{\text{backward}}(w_t \mid w_{t+1}, \ldots, w_T)$$

- Equivalent to train biLSTM with joint objective:

$$P_{\text{forward}}(w_t \mid w_{t-1}, \ldots, w_1) + P_{\text{backward}}(w_t \mid w_{t+1}, \ldots, w_T)$$

- LSTMs are merged at the last layer $\rightarrow$ **late fusion**

Transformer Networks

Motivations

- BiLSTM = concatenation of two sequence models
- Not designed to learn word representation from whole context
- Can we build a model that directly look at the whole context?
- Insipration: c-bow for word representations

Self attention: motivation

- LSTM: $\mathrm{y}_t = f(\mathrm{h}_{t-1}, w_t) \rightarrow$ whole past in 1 vector, h_{t-1}

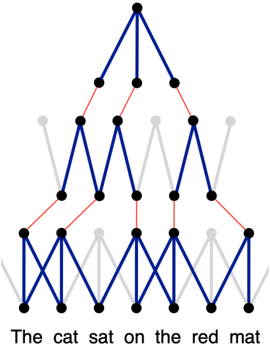
Self attention: motivation

- LSTM: $\mathrm{y}_t = f(\mathrm{h}_{t-1}, w_t) \rightarrow$ whole past in 1 vector, h_{t-1}
- biLSTM: $\mathrm{y}_t = f(\mathrm{h}_{t-1}, \mathrm{h}_{t+1}, w_t) \rightarrow$ whole past+future in 2 vectors.

Self attention: motivation

- LSTM: $\mathrm{y}_t = f(\mathrm{h}_{t-1}, w_t) \rightarrow$ whole past in 1 vector, h_{t-1}
- biLSTM: $\mathrm{y}_t = f(\mathrm{h}_{t-1}, \mathrm{h}_{t+1}, w_t) \rightarrow$ whole past+future in 2 vectors.
- Can we use all past+future vectors?

Convolutional Neural Networks?

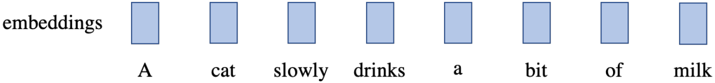


- Pros
  - easy to parallelize
  - exploits local context
- Cons
  - limited context
  - hard to capture long term dependency

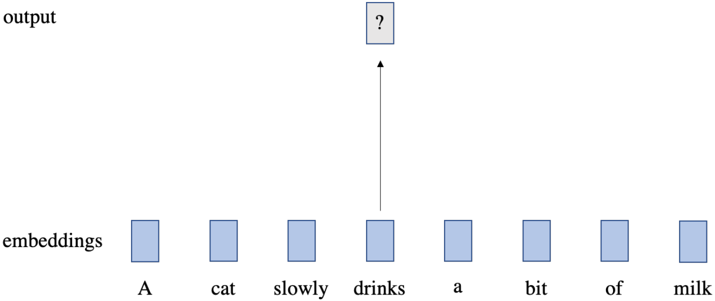
Combining vectors with attention

- Goal: use all the context to update a word
- Idea: look for the most important words in the context
- Solution: self-attention on the sequence of inputs

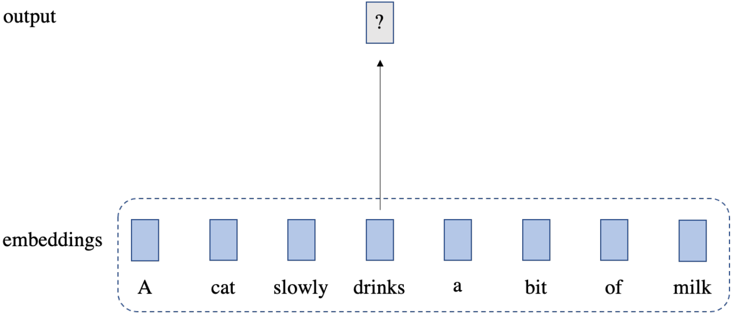
Combining vectors with attention



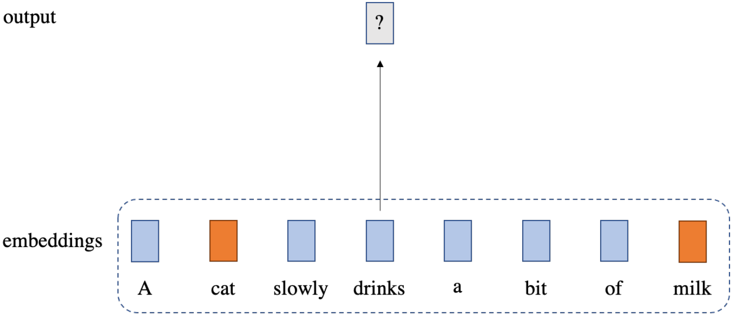
Combining vectors with attention



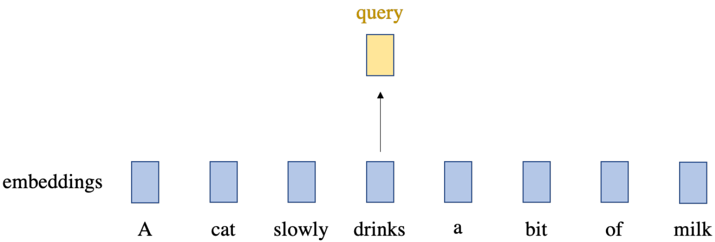
Combining vectors with attention



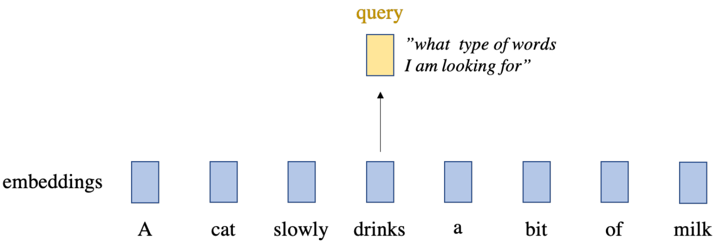
Combining vectors with attention



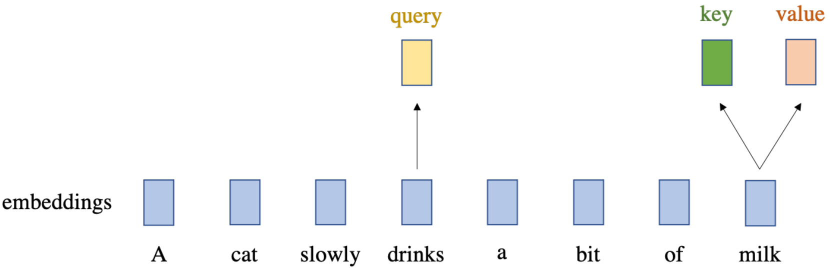
Combining vectors with attention



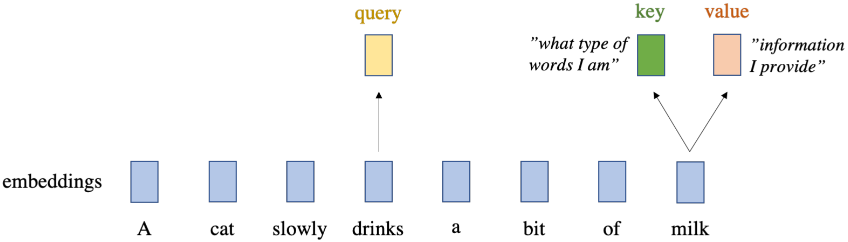
Combining vectors with attention



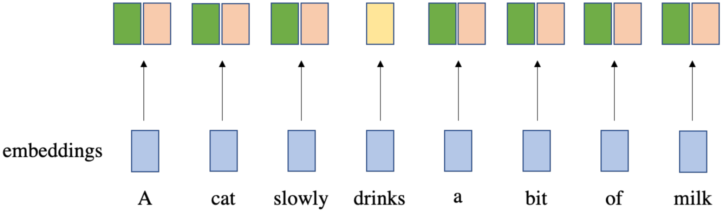
Combining vectors with attention



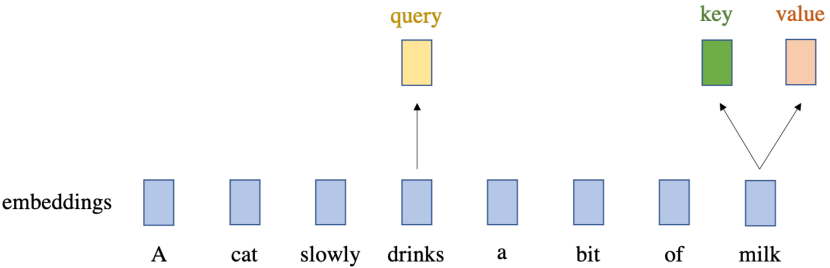
Combining vectors with attention



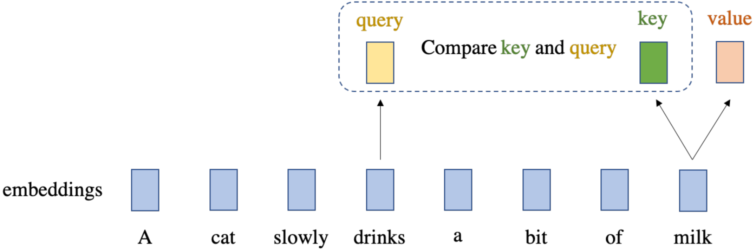
Combining vectors with attention



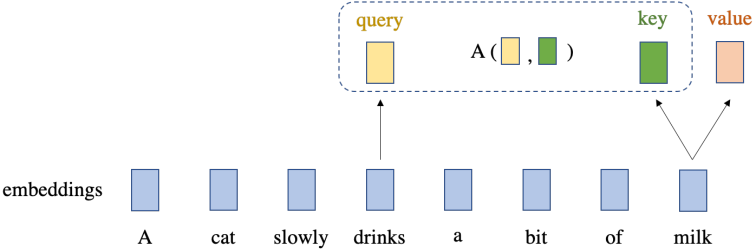
Combining vectors with attention



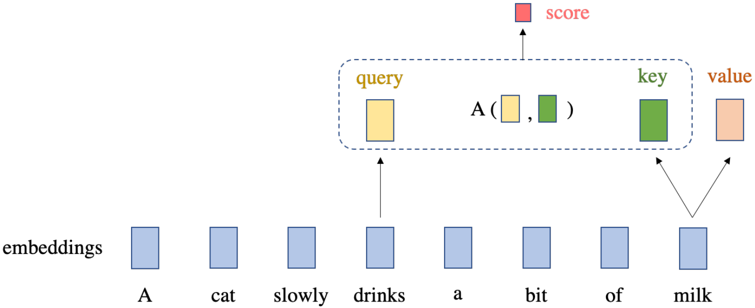
Combining vectors with attention



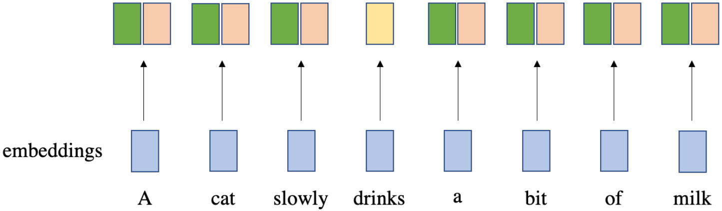
Combining vectors with attention



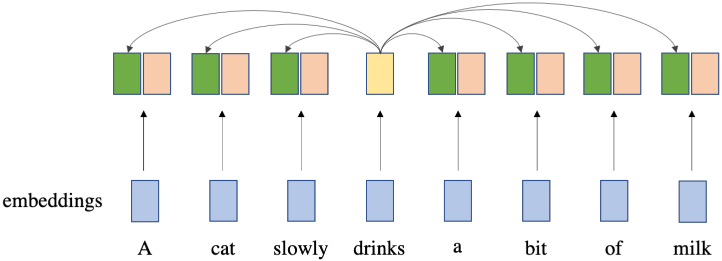
Combining vectors with attention



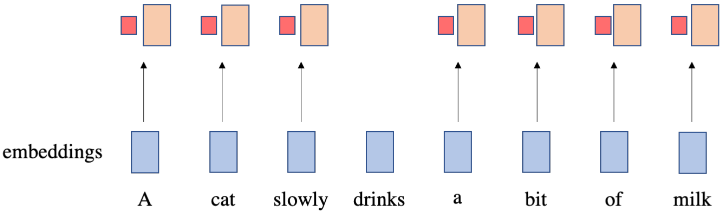
Combining vectors with attention



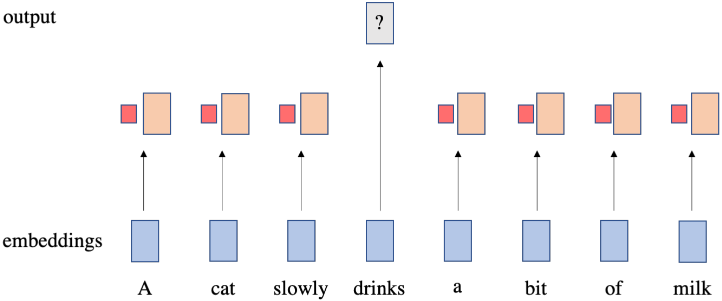
Combining vectors with attention



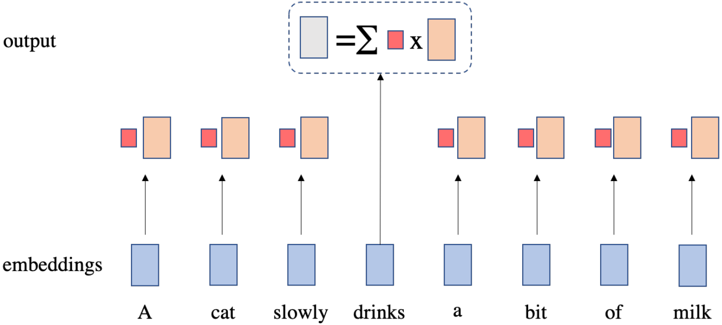
Combining vectors with attention



Combining vectors with attention



Combining vectors with attention



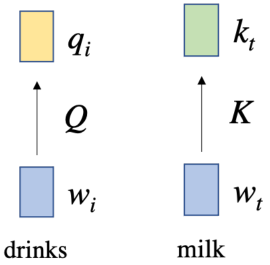
Combining vectors with attention

- "query vector" for word i ("drinks"):

$$q_i = Qw_i$$

- "key vector" for word t ("milk"):

$$k_t = Kw_t$$



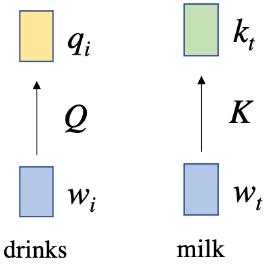
Combining vectors with attention

- "query vector" for word i ("drinks"):

$$\mathrm{q}_i = \mathrm{Q}\mathrm{w}_i$$

- "key vector" for word t ("milk"):

$$\mathrm{k}_t = \mathrm{K}\mathrm{w}_t$$



- Their similarity score is then:

$$s_{it} = \mathrm{q}_i^\top \mathrm{k}_t$$

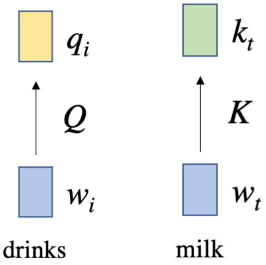
Combining vectors with attention

- “query vector” for word i (“drinks”):

$$\mathrm{q}_i = \mathrm{Q}\mathrm{w}_i$$

- “key vector” for word t (“milk”):

$$\mathrm{k}_t = \mathrm{K}\mathrm{w}_t$$



- Their similarity score is then:

$$s_{it} = \mathrm{q}_i^\top \mathrm{k}_t$$

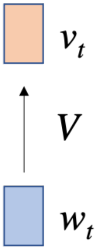
- Normalize over sequence with softmax:

$$a_{it} = \frac{\exp(s_{it})}{\sum_k \exp(s_{ik})}$$

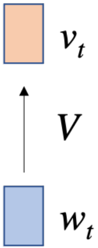
Combining vectors with attention

- "value vector" for word t ("milk"):

$$\mathrm{v}_t = \mathrm{V}\mathrm{w}_t$$



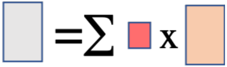
Combining vectors with attention



- "value vector" for word t ("milk"):

$$\mathrm{v}_t = \mathrm{V}\mathrm{w}_t$$

- Finally, compute output for "drinks":



$$\mathrm{y}_i = \sum_t a_{it}\mathrm{v}_t$$

Efficient self-attention with matrix operations

- Compute query, key and value matrix:

$$\mathsf{QW}, \quad \mathsf{KW}, \quad \mathsf{VW}$$

Efficient self-attention with matrix operations

- Compute query, key and value matrix:

$$\mathsf{QW}, \quad \mathsf{KW}, \quad \mathsf{VW}$$

- Compute attention weights

$$\mathsf{A} = \texttt{softmax}(\mathsf{W}^\top \mathsf{K}^\top \mathsf{QW})$$

where softmax is applied column-wise

Efficient self-attention with matrix operations

- Compute query, key and value matrix:

$$\mathsf{QW}, \quad \mathsf{KW}, \quad \mathsf{VW}$$

- Compute attention weights

$$\mathsf{A} = \mathtt{softmax}(\mathsf{W}^\top \mathsf{K}^\top \mathsf{QW})$$

  where softmax is applied column-wise

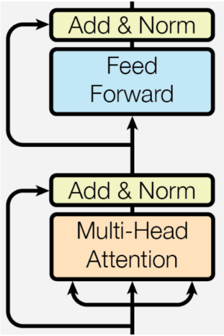
- Then, output is obtained with

$$\mathsf{Y} = \mathsf{VW}\mathtt{softmax}(\mathsf{W}^\top \mathsf{K}^\top \mathsf{QW})$$

Transformer network

Transformer block:

- Multi-head attention layer with skip connection and normalization
- Followed by feed forward with skip connection and normalization



Vaswani et al. (2017)

Skip connection+normalization:

- Given a network block nn and input x
- The output y is computed as

$$y = \mathrm{norm}(x + \mathrm{nn}(x))$$

where norm normalize the input

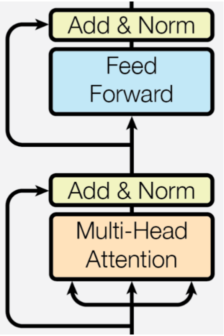
Transformer network

Multi-head self-attention block

- Split each input vector into k non overlapping sub-vectors

$\rightarrow$ the T input vectors of dimension d are split into k sets of T vectors of dimension d/k

- k self-attention layer run in parallel
- The d/k dimension output vector are concatenated back into a d dimensional vector



Vaswani et al. (2017)

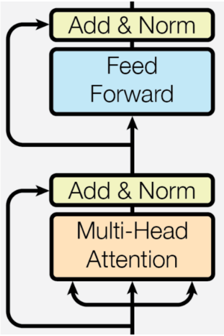
Transformer network

Feed forward block

- Two layer network, with ReLU activation

$$y = W_2 \texttt{ReLU}(W_1 x)$$

- Usually, $W_1 \in \mathbb{R}^{4d \times d}$ and $W_2 \in \mathbb{R}^{d \times 4d}$
- i.e. hidden layer of dimension $4d$.



Vaswani et al. (2017)

Position embeddings

- **Limitation:** self attention does not take position into account!
- Indeed, shuffling the input gives the same results

Position embeddings

- **Limitation:** self attention does not take position into account!
- Indeed, shuffling the input gives the same results

- **Solution:** add position encodings.
- Replace the matrix W by $\mathsf{W} + \mathsf{E}$, where $\mathsf{E} \in \mathbb{R}^{d \times T}$

Position embeddings

- **Limitation:** self attention does not take position into account!
- Indeed, shuffling the input gives the same results

- **Solution:** add position encodings.
- Replace the matrix W by W + E, where $E \in \mathbb{R}^{d \times T}$

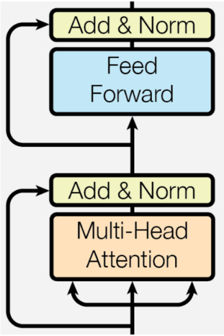
- E can be learned, or defined using sin and cos:

$$e_{2i,j} = \sin\left(\frac{j}{10000^{2i/d}}\right)$$
$$e_{2i+1,j} = \cos\left(\frac{j}{10000^{2i/d}}\right)$$

Transformer network

Transformer network:

- Word embeddings + Position embeddings
- Then N transformer blocks (e.g. $N = 12$)
- Softmax classifier (e.g. for language modeling)

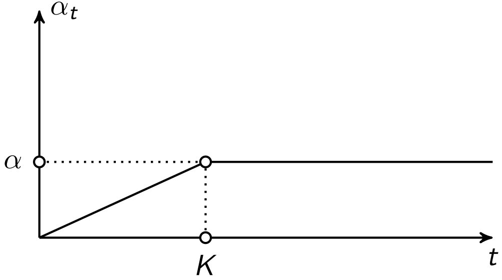


Vaswani et al. (2017)

Training of a Transformer

- In practice, transformers are very unstable during training
- If the learning rate is too large, it diverges
- However if the learning rate is too small, it does not learn well

Training of a Transformer



Learning rate scheduler $(\alpha_t)_t$

- Set a target learning rate α

$$\alpha_t = \min(1, \ \frac{t}{K})\alpha$$

where K is the "warm-up" parameter

BERT: Transformers with early fusion

Introduction

- We consider a deep Transformer, i.e. with more than one layer.
- Embeddings sees the past and the future: impossible to train with language modeling!
- These models fuse information from past and future early in layers $\rightarrow$ **early fusion**

Cloze procedure

- A task to train models with early fusion: **Cloze procedure** Taylor (1953)
- **Key idea** remove words from the input and predict them with the remaining input
- Share similarities with the training of the cbow model for distributed word vectors
- Transformer + Cloze Procedure = **BERT**

Cloze procedure

Sentence The cat is drinking milk in the kitchen

Cloze procedure

Sentence The cat is drinking milk in the kitchen

Input The cat <MASK> drinking <MASK> in the kitchen

- randomly replace 15% of words in sentence with a <MASK> token

Cloze procedure

Sentence	The cat is drinking milk in the kitchen
Input	The cat <MASK> drinking <MASK> in the kitchen
Targets	{"is", "milk"}

- randomly replace 15% of words in sentence with a <MASK> token
- Take the masked words as targets for the model to predict

Cloze procedure

Sentence	The cat is drinking milk in the kitchen
Input	The cat mushroom drinking shoes in the kitchen
Targets	{"is", "milk"}

- randomly replace 15% of words in sentence with a <MASK> token
- Take the masked words as targets for the model to predict
- Extension: use random words from vocabulary instead of <MASK>

Cloze procedure

Sentence The cat is drinking milk in the kitchen

Input The cat <MASK> drinking <MASK> in the kitchen

Targets {“is”, “milk”}

- randomly replace 15% of words in sentence with a <MASK> token
- Take the masked words as targets for the model to predict
- Extension: use random words from vocabulary instead of <MASK>
- Related to noisy autoencoders

Transformer network for sentence representation

- If attention span over the whole sequence, it is a early fusion model
- There is no masking in this case
- Similar to bi-LSTM, Transformer trained with a Cloze procedure

Sentence	The cat is drinking milk in the kitchen
input	The cat <MASK> drinking <MASK> in the kitchen
targets	{"is", "milk"}

- Popular model: BERT (**?**)

Evaluation of sentence representations

- Apply representation on downstream tasks like text classification
- Compare representation of similar sentences (e.g. obtained from paraphrasing)
- Identify relations between sentences: is one the negation of the other? Does one imply the other?
- Question answering: are the embeddings of a question and its answer similar?

GLUE: a benchmark for sentence representations

GLUE (**?**) contains 11 tasks covering:

- Single-Sentence Tasks (e.g., text classification)
- Similarity and Paraphrase Tasks
- Inference tasks, i.e., predicting relations between sentences (e.g., coreference, NLI,...)

Caveat of GLUE finetuning of models on each task is allowed.

Leaderboard available at `https://gluebenchmark.com/`

GLUE: a benchmark for sentence representations

Model	Avg. Acc.
CBoW	58.9
BiLSTM with late fusion	64.2
Transformer with late fusion	72.8
Transformer with early fusion	**80.5**

- CBoW is a Bag-of-Word representation on top of word GloVe vectors
- **Beware!** Numbers are not directly comparable because models are trained on different datasets

Transformers for Language Modeling

Masking for Transformer Language Models

- In transformer, h_t depends on all inputs
- Could not be used as is for language modeling
- Solution: use mask in attention, to only use past

Masking for Transformer Language Models

- In transformer, h_t depends on all inputs
- Could not be used as is for language modeling
- Solution: use mask in attention, to only use past

- Reminder:

$$\begin{aligned} H &= VW\text{softmax}(W^\top K^\top QW) \\ &= VWA \end{aligned}$$

  Hence, a_{it} is weight of input i in representation of position t
- We want representation at time t to only depends on $i \leq t$
- We could enforce $a_{it} = 0$ for $i \geq t$

Masked softmax

- We introduce the masked softmax operator
- Given an input x and a binary mask m,

$$[\mathrm{masked\_softmax}(\mathrm{x}, \mathrm{m})]_i = \frac{m_i \exp(x_i)}{\sum_{i=1}^{d} m_i \exp(x_i)}$$

- Still sums to one, $m_i = 0$ implies $[\mathrm{masked\_softmax}(\mathrm{x}, \mathrm{m})]_i = 0$

Masked softmax

- We introduce the masked softmax operator
- Given an input x and a binary mask m,

$$[\mathrm{masked\_softmax}(\mathrm{x}, \mathrm{m})]_i = \frac{m_i \exp(x_i)}{\sum_{i=1}^{d} m_i \exp(x_i)}$$

- Still sums to one, $m_i = 0$ implies $[\mathrm{masked\_softmax}(\mathrm{x}, \mathrm{m})]_i = 0$
- Sometimes implemented as:

$$\mathrm{softmax}(\mathrm{x} + \log(\mathrm{m}))$$

- **Beware:** do not learn the mask (e.g. PyTorch: `register_buffer`)

Transformer network for Language Modeling: Results

Model	bpc
LSTM	1.25
Transformer	**1.07**

- Text8
- Character level language modeling
- bpc = bit per character.

References I

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *JMLR*.

Brants, T., Popat, A. C., Xu, P., Och, F. J., and Dean, J. (2007). Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*.

Katz, S. M. (1987). Estimation of probabilities from sparse data for the language model component of a speech recognizer. *ICASSP*.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*.

Shi, X., Knight, K., and Yuret, D. (2016). Why neural translations are the right length. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2278–2282.

References II

Taylor, W. L. (1953). Cloze procedure: A new tool for measuring readability. *Journalism quarterly*, 30(4):415–433.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.