

# Language Modeling and Sentence Representation

Edouard Grave, **Armand Joulin**

Facebook AI Research  
ajoulin@fb.com

# Plan of this lecture

- Language modeling
  - Standard  $n$ -gram language models
  - neural  $n$ -gram models
  - Recurrent neural networks
- Sentence representation
  - BiLSTM
  - Late and early fusion
  - Transformer network

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** assigning probability to a text
- A text is a sequence of tokens
- tokens can be words, characters or group of characters.
- For example:

$$\{\text{a cat}\} = \{\text{a}, \text{cat}\},$$

# What is language modeling

- **Language modeling** assigning probability to a text
- A text is a sequence of tokens
- tokens can be words, characters or group of characters.
- For example:

$$\begin{aligned}\{\text{a cat}\} &= \{\text{a, cat}\}, \\ &= \{\text{a, }, \text{c, a, t}\},\end{aligned}$$

# What is language modeling

- **Language modeling** assigning probability to a text
- A text is a sequence of tokens
- tokens can be words, characters or group of characters.
- For example:

$$\begin{aligned}\{\text{a cat}\} &= \{\text{a, cat}\}, \\ &= \{\text{a, }, \text{c, a, t}\}, \\ &= \{\text{a, }, \text{ca, t}\}.\end{aligned}$$

# What is language modeling

- **Language modeling** assigning probability to a text
- A text is a sequence of tokens
- tokens can be words, characters or group of characters.
- For example:

$$\begin{aligned}\{\text{a cat}\} &= \{\text{a, cat}\}, \\ &= \{\text{a, }, \text{c, a, t}\}, \\ &= \{\text{a, }, \text{ca, t}\}.\end{aligned}$$

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

# What is language modeling

- Given a sequence  $\{w_1, \dots, w_T\}$  of tokens, a language model estimates its probability:

$$P(w_1, \dots, w_T)$$

- $P$  depends on a **vocabulary**, i.e., the set of unique tokens.
- $P$  can be conditioned on an external variable, i.e.,  $P(.) = P(. \mid C)$



# Applications of language modeling

Language models are applied in several fields:

- Speech recognition:

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

- Machine translation:

$$\begin{aligned} P(\text{"Déçu en bien"} \mid \text{"Pleasantly surprised"}) &< \\ P(\text{"Agréablement surpris"} \mid \text{"Pleasantly surprised"}) \end{aligned}$$

- Optical Character Recognition:

$$P(\text{"m0ve fast"}) < P(\text{"move fast"})$$

# Probabilistic language model

- Sequence probability as a product of token probabilities:

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

# Probabilistic language model

- Sequence probability as a product of token probabilities:

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

- Indeed we have:

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

# Probabilistic language model

- Sequence probability as a product of token probabilities:

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

- 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:

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

- 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 models estimate probability of upcoming token given past:

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

*n*-gram language models

# Count based language model

Compute probability with counting statistics from a dataset:

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

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

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

# Count based language model

- Example:

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

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

Sentence “The moment one learns” appears 75 in dataset



# Limitations of count based language model

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

→ **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:

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

# Count based language model

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

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

- Bigram:

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

## Count based language model: unigram

- Probability of a sentence with a unigram model:

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

$N$  = total number of tokens in dataset

$c(w_t)$  = number of occurrences 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:

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

$c(w_{t-1}w_t)$  = number of occurrences 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 truly model language**

## Estimating $n$ -gram probabilities: an example

- bigram:

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

- Dataset:

*<s>we sat in the house*

*<s>we sat here we two and we said*

*<s>how we wish we had something to do*

- Extract some probabilities:

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

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



## Estimating $n$ -gram probabilities: 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 probabilities: 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 probabilities: an example

- Example:

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

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

- Result:

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

## Estimating $n$ -gram probabilities: 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:

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

- Result:

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

- Does not generalize well!

## Estimating $n$ -gram probabilities: 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 probabilities

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

→ Add-1 is not good in practice

# Good-Turing estimation

- **Idea** reallocate probability mass of  $n$ -grams that occur exactly  $c + 1$  times to  $n$ -grams that occur exactly  $c$  times
- reallocate mass of  $n$ -grams appearing once to unseen  $n$ -grams

→ **alternative to Add-1**

## Good-Turing estimation

- **Idea** reallocate probability mass of  $n$ -grams that occur exactly  $c + 1$  times to  $n$ -grams that occur exactly  $c$  times
- reallocate mass of  $n$ -grams appearing once to unseen  $n$ -grams

→ **alternative to Add-1**

- the adjusted count:

$$c^* = (c + 1) \frac{N_{c+1}}{N_c}$$

where  $N_c$  is the number of  $n$ -grams that appears exactly  $c$  times



# Good-Turing estimation

- **Idea** reallocate probability mass of  $n$ -grams that occur exactly  $c + 1$  times to  $n$ -grams that occur exactly  $c$  times
- reallocate mass of  $n$ -grams appearing once to unseen  $n$ -grams

## → **alternative to Add-1**

- the adjusted count:

$$c^* = (c + 1) \frac{N_{c+1}}{N_c}$$

where  $N_c$  is the number of  $n$ -grams that appears exactly  $c$  times

- $n$ -gram probability depends on  $c^*$  instead of  $c$

## Good-Turing estimation

- **Idea** reallocate probability mass of  $n$ -grams that occur exactly  $c + 1$  times to  $n$ -grams that occur exactly  $c$  times
- reallocate mass of  $n$ -grams appearing once to unseen  $n$ -grams

### → **alternative to Add-1**

- the adjusted count:

$$c^* = (c + 1) \frac{N_{c+1}}{N_c}$$

where  $N_c$  is the number of  $n$ -grams that appears exactly  $c$  times

- $n$ -gram probability depends on  $c^*$  instead of  $c$
- **Problem** What if  $N_{c+1} = 0$  (but  $N_c > 0$ )?

# 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  $c(w_{t-n+1} \cdots w_t) > 0$ :

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

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

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

- 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:

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

# Conditioned Interpolation

- Requires to split training dataset in two
- On one dataset, compute the  $n$ -gram probabilities
- On the second dataset, learn the  $\lambda$ 's that fits the best.
- Learning the  $\lambda$ 's is a logistic regression problem:

$$\max_{\lambda} \sum_i \log \left[ \sum_j \lambda_j(w_{i-1}, w_{i-2}) P(w_i \mid w_{i-1}, \dots, w_{i-3+j}) \right]$$

# Kneser-Ney Smoothing

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

- You recursively apply this formula to get the explicit probability



# Kneser-Ney Smoothing

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

- You recursively apply this formula to get the explicit probability
- Indeed (with simplified notations):

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

## Kneser-Ney Smoothing

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

# Kneser-Ney Smoothing

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

- 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  $d$  is discount factor

it works as a threshold too:  $f_{KN} = 0$  for any  $n$ -gram appearing less than  $d$  times.

# Kneser-Ney Smoothing

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

- $\lambda$  is the interpolation weight:

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

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

# 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
- 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.

# Cache model

- Ngram models compute statistics over the whole dataset
- Locally, this distribution can change!
- Example:
  - Full Dataset = all the news article about football
  - Current prediction is on an article about a specific team
  - The name of the players in this team are more likely to appear than in the full dataset
- **Cache model** keeps an history  $\mathcal{H}$  of recent past and adjust prediction to increase the probability of the words in this history:

$$P_C(w_t \mid w_{t-1}, w_{t-2}) = \lambda P(w_t \mid w_{t-1}, w_{t-2}) + (1 - \lambda) \frac{|w_t \in \mathcal{H}|}{|\mathcal{H}|}$$

# Large scale $n$ -gram models

**Pruning** keep only  $n$ -grams with counts  $>$  threshold

## **Memory efficiency**

- Store  $n$ -grams in tri trees
- Store indices instead of string
- Quantization: 4-8 bits instead of 8-byte floats

**Approximated LM** bloom filters instead of indices to retrieve  $n$ -grams



# 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 | <b>responsability</b> | 0.4 | <b>responsability</b> | 0.8 |
| <b>responsability</b> | 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  $W = (w_1, \dots, w_T)$  is:

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

- 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:

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

- It is the negative log-likelihood of the sequence

## Example of Perplexity

|           | Unigram | Bigram | Trigram |
|-----------|---------|--------|---------|
| <i>PP</i> | 962     | 170    | 109     |

Lower perplexity means better model

As expected, better model with longer  $n$ -grams



## 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  $(w_1, \dots, w_{t-1})$
  - The output  $Y$  is the current token  $w_t$
  - The token  $w_t$  is a class label among  $V$  possibilities
  - Language modeling can be casted as a supervised classification

## Preliminaries: loss function

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

## Preliminaries: words as vectors

- We assume a fixed vocabulary of  $V$  words
- we represent the  $i$ -th word by a  $V$  dimensional vector  $\mathbf{w}_i$ :

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

- These word vectors are:
  - independent:  $\mathbf{w}_i^T \mathbf{w}_j = 0$  if  $i \neq j$
  - normalized:  $\mathbf{w}_i^T \mathbf{w}_i = 1$
- We call this representation “one-hot vectors”
- For now on, the notation  $\mathbf{w}_t$  represents the one-hot vector of the word at the  $t$ -th position in the sentence

## A linear model for bigrams

- The input is the 1-hot vector of the previous word:  $\mathbf{x}_t = \mathbf{w}_{t-1}$
- The output is the 1-hot vector of the upcoming word:  $y_t = \mathbf{w}_t$
- **Linear model  $\mathbf{z} = \mathbf{Ax}$**
- Build a probability over all possible words:

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

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

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

## Pros of linear models over $n$ -grams

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

- Can learn the same statistics as those in the  $n$ -gram models
- We can put additional features into  $\mathbf{x}_t$  (e.g. from WordNet)
- Simple to implement



## Limitations of linear models

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

- The matrix  $\mathbf{A}$  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

# Neural bigram model

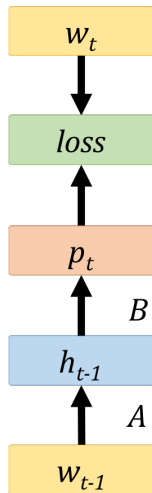
- feedforward network:

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

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

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

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



# Neural $n$ -gram model

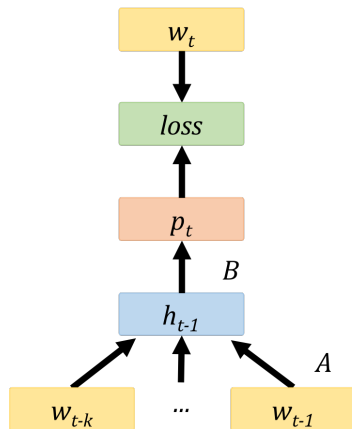
Generalization to any fixed  $n$ -gram size:

- The input is the concatenation of previous words:

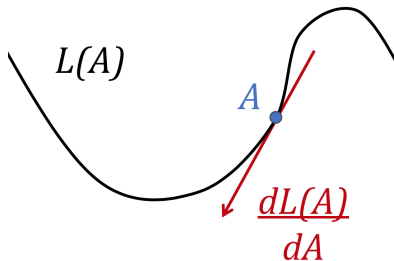
$$\mathbf{x}_t = [\mathbf{w}_{t-n+1}, \dots, \mathbf{w}_{t-1}]$$

- $\mathbf{A}$ :  $nV \times H$  matrix
- Minimization problem:

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



## Neural $n$ -gram model: training



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

# Neural $n$ -gram model: training

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

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

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

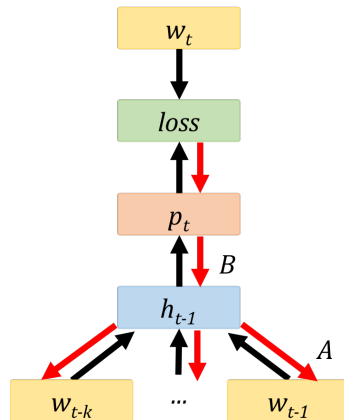
$$L_t(\mathbf{A}, \mathbf{B}) = \ell(y_t, f(\mathbf{B}\sigma(\mathbf{A}\mathbf{x}_t)))$$

- Update parameters:

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

# Computing the gradient with backpropagation

- Compute gradient with **backpropagation**
- Compute the error made by the model when predicting the next word
- Propagate this error back to all of the parameters of the network and the input



## Neural $n$ -gram model: backpropagation

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

## Neural $n$ -gram model: backpropagation

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

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



## Neural $n$ -gram model: backpropagation

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

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

## Neural $n$ -gram model: backpropagation

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

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

## Neural $n$ -gram model: backpropagation

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

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

## Neural $n$ -gram model: backpropagation

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

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

## Neural $n$ -gram model: backpropagation

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

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

## Neural $n$ -gram model: backpropagation

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

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

## Neural $n$ -gram model: backpropagation

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

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

## Neural $n$ -gram model: backpropagation

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

$$\begin{aligned}\frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{B}} &= \frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial \mathbf{B}} + \frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{w}} \frac{\partial \mathbf{w}}{\partial \mathbf{B}} \\ &= \frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{p}} \frac{\partial f(\mathbf{z})}{\partial \mathbf{B}} && \text{where } \mathbf{p} = f(\mathbf{z}) \\ &= \frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{p}} \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{B}} \\ &= \frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{p}} \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \frac{\partial (\mathbf{B}\sigma(\mathbf{A}\mathbf{x}))}{\partial \mathbf{B}} && \text{where } \mathbf{z} = \mathbf{B}\sigma(\mathbf{A}\mathbf{x})\end{aligned}$$



## Neural $n$ -gram model: backpropagation

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

$$\begin{aligned}\frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{B}} &= \frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial \mathbf{B}} + \frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{w}} \frac{\partial \mathbf{w}}{\partial \mathbf{B}} \\ &= \frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{p}} \frac{\partial f(\mathbf{z})}{\partial \mathbf{B}} && \text{where } \mathbf{p} = f(\mathbf{z}) \\ &= \frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{p}} \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{B}} \\ &= \frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{p}} \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \frac{\partial (\mathbf{B}\sigma(\mathbf{A}\mathbf{x}))}{\partial \mathbf{B}} && \text{where } \mathbf{z} = \mathbf{B}\sigma(\mathbf{A}\mathbf{x})\end{aligned}$$

## Neural $n$ -gram model: backpropagation

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

$$\begin{aligned}\frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{B}} &= \frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial \mathbf{B}} + \frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{w}} \frac{\partial \mathbf{w}}{\partial \mathbf{B}} \\ &= \frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{p}} \frac{\partial f(\mathbf{z})}{\partial \mathbf{B}} && \text{where } \mathbf{p} = f(\mathbf{z}) \\ &= \frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{p}} \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{B}} \\ &= \frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{p}} \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \frac{\partial (\mathbf{B}\sigma(\mathbf{A}\mathbf{x}))}{\partial \mathbf{B}} && \text{where } \mathbf{z} = \mathbf{B}\sigma(\mathbf{A}\mathbf{x}) \\ &= \frac{\partial \ell(\mathbf{w}, \mathbf{p})}{\partial \mathbf{p}} \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \sigma(\mathbf{A}\mathbf{x})\end{aligned}$$

## Neural $n$ -gram model: backpropagation

- Loss function:

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

- The gradients are:

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

with  $\mathbf{z}_t = \mathbf{B}\mathbf{h}_t$  and  $\mathbf{h}_{t-1} = \sigma(\mathbf{A}\mathbf{x}_t)$

# Neural $n$ -gram model: backpropagation

- Loss function:

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

- The gradients are:

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

with  $\mathbf{z}_t = \mathbf{B}\mathbf{h}_t$  and  $\mathbf{h}_{t-1} = \sigma(\mathbf{A}\mathbf{x}_t)$

- Note that some intermediate computations are shared to evaluate different gradients

# Regularization: dropout

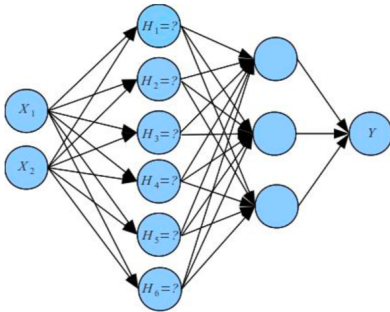
- Specialized units cause overfitting
- **Idea** force model to work even when some units are removed
- Same as activation mask over units
- we replace  $\mathbf{h}_t$  by:

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

where  $\mathbf{m}_t$  is a binary mask vector.

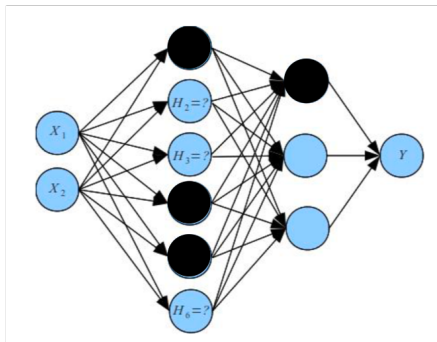
- This binary mask is randomly drawn for each time step

# Regularization: dropout



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

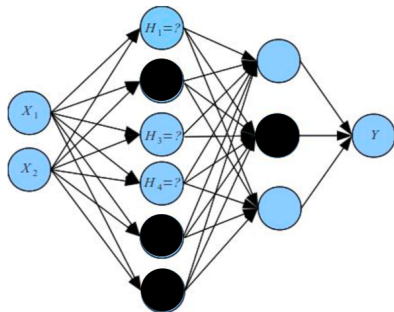
# Regularization: dropout



Iteration 1

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

# Regularization: dropout



Iteration 2

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



## Regularization: tied input-output matrix

- Input matrix  $\mathbf{A}$  is a  $V \times H$  matrix
  - Output matrix  $\mathbf{B}$  is a  $H \times V$  matrix
  - For large vocabulary ( $V \gg 1$ ), these matrices holds most of the parameters of the model
- This may lead to some overfitting
- **Solution** tie the input and output weights (Press and Wolf, 2016):

$$\mathbf{B} = \mathbf{A}^T$$

# 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

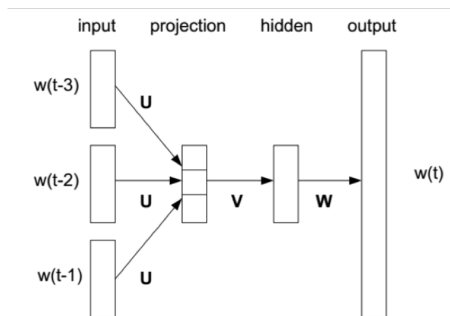
## Dealing with large vocabulary: class-based softmax

- Use a class-based softmax (see lecture on text classification)
- partition vocabulary in  $L$  subvocabulary,  $V_k$  is  $k$ -th subvocabulary

$$P(C = c_k \mid \mathbf{h}_t)P(w_t = k \mid \mathbf{h}_t, C = c_k)$$

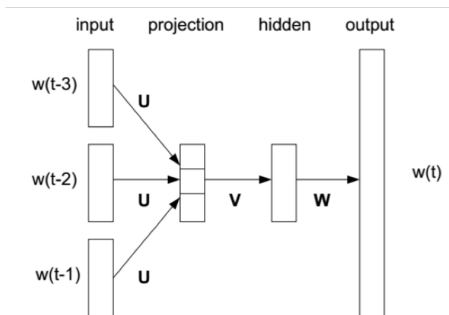
- Typically we take  $L = \sqrt{V}$
  - Subvocabularies are selected to have the same frequency.
  - frequency of subvocabulary is the sum of the frequency of its words
  - Frequency of words varies greatly (“the” versus “bardiwac”)
- size of subvocabularies varies greatly, but large subvocabularies are visited rarely

## 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  $\mathbf{U}\mathbf{w}_t$  is a distributed word representation
- These representations are passed through a feedforward network

## Neural $n$ -gram model: example



- The equations are:

$$\mathbf{x}_{t-k} = \sigma(\mathbf{U}\mathbf{w}_{t-k}) \quad (\text{distributed representation})$$

$$\mathbf{h}_{t-1} = \sigma(\mathbf{V}[\mathbf{x}_{t-3}, \mathbf{x}_{t-2}, \mathbf{x}_{t-1}]) \quad (\text{hidden representation})$$

$$\mathbf{p}_t = f(\mathbf{W}\mathbf{h}_{t-1}) \quad (\text{output probability})$$

## 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

## 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)

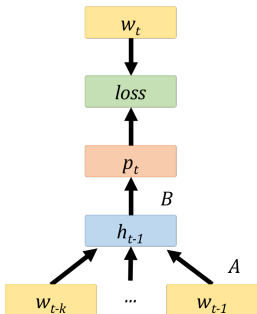


# Recurrent Neural Network

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

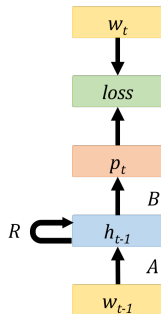
## Feedforward

$$\mathbf{h}_{t-1} = \sigma(\mathbf{A}[\mathbf{w}_{t-k}, \dots, \mathbf{w}_{t-1}])$$
$$\mathbf{p}_t = f(\mathbf{B}\mathbf{h}_{t-1})$$

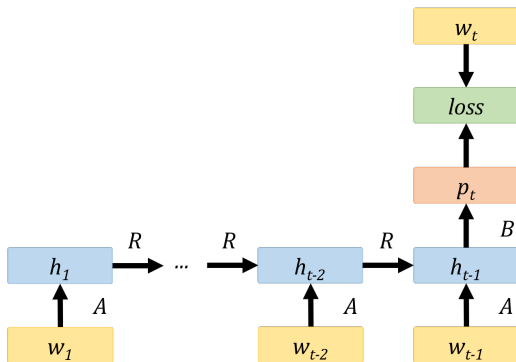


## Recurrent Network

$$\mathbf{h}_{t-1} = \sigma(\mathbf{A}\mathbf{w}_{t-1} + \mathbf{R}\mathbf{h}_{t-2})$$
$$\mathbf{p}_t = f(\mathbf{B}\mathbf{h}_{t-1})$$

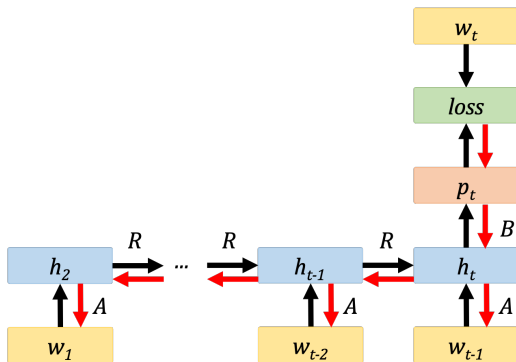


# Recurrent Neural Network



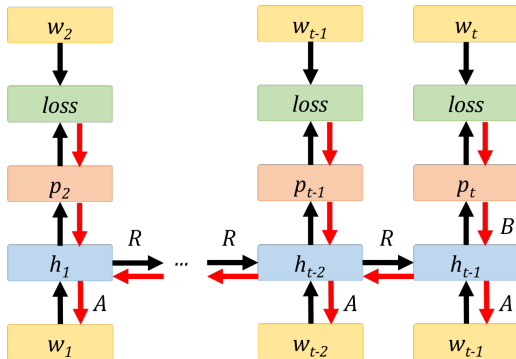
- Recurrent equation:  $\mathbf{h}_t = \sigma(\mathbf{A}[\mathbf{h}_{t-1}, \mathbf{w}_t])$
- Unfold over time: **very deep feedforward with weight sharing**
- Potentially capture long term dependencies

# Recurrent Neural Network: training



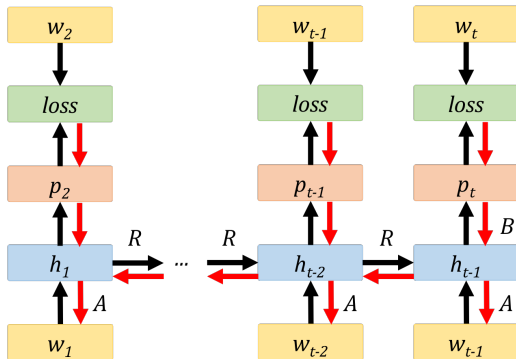
- **Backpropagation through time (BPTT)**: same as backpropagation through a very deepfeedforward network

# Recurrent Neural Network: training



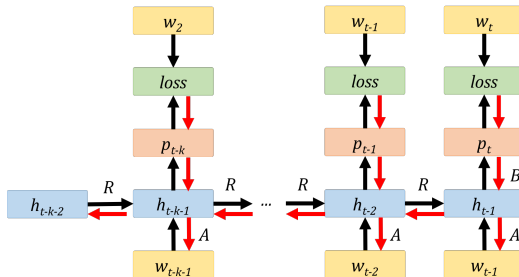
- **batch BPTT**: forward/backward for many words simultaneously

# 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                                    | <b>125</b> |

- 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:

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

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

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



# RNN: Exploding gradient

- The matrix  $\mathbf{R}$  are not directly multiplied in the partial derivatives
- Impossible to lower bound partial derivative norms
- Popular incorrect argument:

$$\prod_k [\text{diag}(\sigma'(\mathbf{z}_k))\mathbf{R}] \approx \mathbf{R}^k$$

- we cannot lowerbound  $\text{diag}(\sigma'(\mathbf{z}_k))$  nor permute it with  $\mathbf{R}$
- Even if we could,  $\mathbf{R}^k$  is not informative (e.g., nilpotent matrices)
- However the intuition is still correct: if the maximum singular value of  $\mathbf{R}$  is such that  $\lambda_{\max} \gg 1$ : **the gradient might explode**

## RNN: Exploding gradient

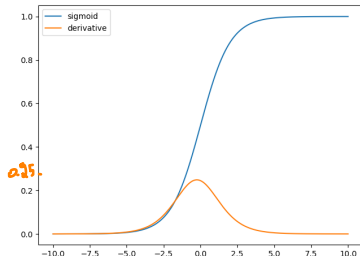
- Consequence: hard to learn a RNN with gradient descent
- **Exploding gradient is an optimization problem**
- Simple hack to fix this problem: **gradient clipping**:

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

with  $\mu > 0$

- it bounds the norm of a gradient  $G$  to be at most  $\mu$

# RNN: Vanishing gradient



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

# RNN: Vanishing gradient

- Putting  $\mathbf{R}$  and  $\sigma'$  together, we have:

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

- Partial derivative is such that

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

- If  $\lambda_{\max} < 4$ : **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 is a model problem, not an optimization problem**
- Solutions require a change in the structure of the model

## Long Short Term Memory (LSTM)

# Long Short Term Memory (LSTM)

- Vanilla RNN:

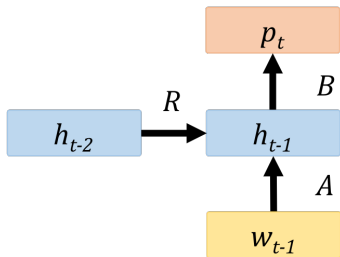
$$\mathbf{h}_t = \sigma(A\mathbf{w}_t + \mathbf{R}\mathbf{h}_{t-1})$$

- Any function could work:

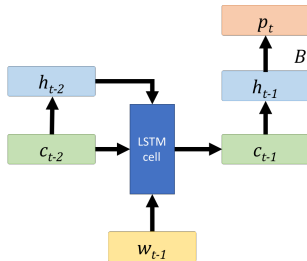
$$\mathbf{h}_t = \phi(\mathbf{w}_t, \mathbf{h}_{t-1})$$

- Preferably  $\phi$  should be mostly differentiable and reduces the vanishing gradient problem

# Long Short Term Memory (LSTM)



RNN

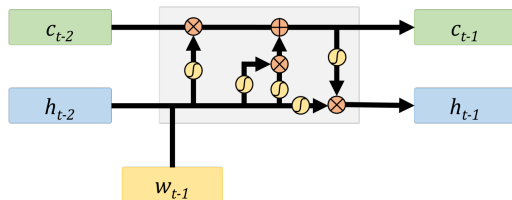


LSTM

- 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:

$$\mathbf{c}_t = f_t \circ \mathbf{c}_{t-1} + i_t \circ \tanh(\mathbf{A}\mathbf{w}_t + \mathbf{R}\mathbf{h}_{t-1})$$

$$\mathbf{h}_t = o_t \circ \tanh(\mathbf{W}\mathbf{c}_t)$$

Using  $\tanh h(x) = 2\sigma(2x) - 1$  instead of  $\sigma$  is not important.

- with:

$$f_t = \sigma(\mathbf{A}_f\mathbf{w}_{t-1} + \mathbf{R}_f\mathbf{h}_{t-1})$$

forget gate

$$i_t = \sigma(\mathbf{A}_i\mathbf{w}_{t-1} + \mathbf{R}_i\mathbf{h}_{t-1})$$

input gate

$$o_t = \sigma(\mathbf{A}_o\mathbf{w}_{t-1} + \mathbf{R}_o\mathbf{h}_{t-1})$$

output gate

## Attempt at explaining LSTM

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

$$\mathbf{c}_t = f_t \circ \mathbf{c}_{t-1} + i_t \circ \tanh(\mathbf{A}\mathbf{w}_t + \mathbf{R}\mathbf{h}_{t-1})$$

$$\mathbf{h}_t = \tanh(\mathbf{W}\mathbf{c}_t)$$

- This way,  $\mathbf{h}_t$  only depends on  $\mathbf{c}_t$

## Attempt at explaining the memory cell update

- This is an “hand-wavy” explanation of these equations
- A standard RNN update is:

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

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

$$\mathbf{c}_t = \mathbf{c}_{t-1} + \underbrace{\tanh(\mathbf{A}\mathbf{w}_t + \mathbf{R}\mathbf{c}_{t-1})}_{\text{Same as RNN}}$$

- Let us unroll the computation over time:

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

Limitation  
/ All past info  
have same  
importance

- 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

as bias is int by data

How to make our data unbiased as possible

1 FastText oVV

Vecch Biases in data

- Possible solution: use a discount factor:

- Loc 2 VPC VS

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

Sent 2 VcL

$\eta$  should be in  $[0, 1]$

3-6

- We now have:

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

- **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:

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

- 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  $\mathbf{c}_t$

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

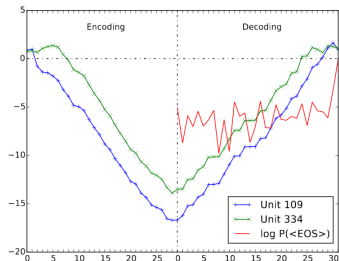
- but the correct equation is:

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

- Why do we need two different variables?
- $\mathbf{h}_t = \tanh(W\mathbf{c}_t) \rightarrow \mathbf{h}_t$  is  $\mathbf{c}_t$  rescaled to  $[-1, 1]$ :
- The benefits are:
  - Rescaling  $\mathbf{h}_t$  avoids gradient explosion
  - Keeping  $\mathbf{c}_t$  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 subtracting 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              | <b>115</b> |

- Penn Treebank dataset
- LSTM outperforms RNN



# Sentence Representation

# From word to sentence representation

- **Distributed word vectors** Assign a fixed size vector to a word
  - Example: word2vec, fasttext, PPMI+SVD...
- **Goal** Build a similar representation for sentences
- Several difficulties:
  - Sentences have variable length
  - Sentences are much richer than words

# Simple sentence representation

- A sentence is a sequence of words  $w_1, \dots, w_T$
- each word has a distributed word vector:  $\mathbf{w}_1, \dots, \mathbf{w}_T$
- Average these vectors to form a sentence representation:

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

- This is a Bag of Words (BoW) representation

# Extensions of simple sentence representation

- Replace average with another operation, e.g., take the max value per dimension:

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

- Add additional features:
  - distributed representation of  $n$ -grams or subwords
  - features from WordNet...
- Use a representation of words that depends on context

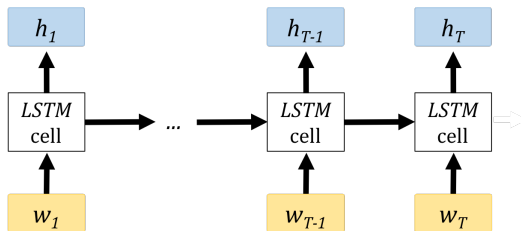
# Extensions of simple sentence representation

- Replace average with another operation, e.g., take the max value per dimension:

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

- Add additional features:
  - distributed representation of  $n$ -grams or subwords
  - features from WordNet...
- Use a representation of words that depends on context

## Using an LSTM for sentence representation



- Apply LSTM on a sentence to produce sequence of vectors

$$\mathbf{h}_1, \dots, \mathbf{h}_T$$

- Use these representations in the BoW sentence representation:

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

- LSTM computes representations with history, not future

# Bidirectional LSTM (BiLSTM)

- **BiLSTM = 2 LSTMs running on opposite direction**

- $\overrightarrow{LSTM}$  runs forward on sequence to produce its representations:

$$\overrightarrow{\mathbf{h}}_1, \dots, \overrightarrow{\mathbf{h}}_T$$

- $\overleftarrow{LSTM}$  runs backward on sequence to produce its representations:

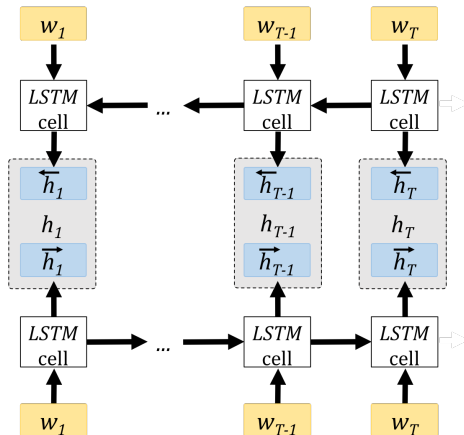
$$\overleftarrow{\mathbf{h}}_1, \dots, \overleftarrow{\mathbf{h}}_T$$

- The output of biLSTM is the concatenation of both representations:

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

- These vectors are called “contextualized word vectors” (Peters et al., 2018).

# Bidirectional LSTM (BiLSTM)



- A BoW sentence representation from an biLSTM is:

$$\mathbf{s} = \frac{1}{T} \sum_{t=1}^T \mathbf{h}_t = \frac{1}{T} \sum_{t=1}^T [\overrightarrow{\mathbf{h}}_t, \overleftarrow{\mathbf{h}}_t]$$



# Training of a biLSTM as a Language model

- Train both LSTMs independently
  - The forward  $\overrightarrow{LSTM}$  is train to predict the upcoming word:

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

- The backward  $\overleftarrow{LSTM}$  is train to predict the previous word:

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

- Equivalently the biLSTM can be trained with joint objective:

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

- Both solutions merge the LSTMs at the last layer  $\rightarrow$  **late fusion**

# Deep biLSTM architecture with early fusion

- Deep biLSTM: more than one layer
- We merge forward and backward representations at each layer  $k$ :

$$\begin{aligned}\vec{\mathbf{h}}_t^k &= \overrightarrow{LSTM}(\mathbf{h}_t^{k-1}), \\ \overleftarrow{\mathbf{h}}_t^k &= \overleftarrow{LSTM}(\mathbf{h}_t^{k-1}). \\ \mathbf{h}_t^k &= [\vec{\mathbf{h}}_t^k, \overleftarrow{\mathbf{h}}_t^k],\end{aligned}$$

- the hidden states  $\mathbf{h}_t^k$  depends on the past and the future
- impossible to train this model with language modeling!
- This is often referred to as **early fusion**

# Deep biLSTM with early fusion

- 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

## Cloze procedure in practice (Devlin et al., 2018)

**Sentence** The cat is drinking milk in the kitchen

## Cloze procedure in practice (Devlin et al., 2018)

**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 in practice (Devlin et al., 2018)

**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 in practice (Devlin et al., 2018)

**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>

# Transformer Networks



## Self attention: motivation

- In recurrent networks, we have

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, w_t).$$

- RNNs encode the whole history in single vector  $\mathbf{h}_{t-1}$
- Instead, can we use **all** word representations to compute  $\mathbf{h}_t$ ?
- Technical challenge:  
    need to combine a **variable** number of representations!

# Combining vectors with attention

- Solution: use the (self) attention mechanism
- Given a set of vectors  $\mathbf{w}_1, \dots, \mathbf{w}_T \in \mathbb{R}^d$  representing words

$$\mathbf{h}_t = \sum_{i=1}^T a_{it} \mathbf{V} \mathbf{w}_i$$

where  $\sum_{i=1}^T a_{it} = 1$ .

- We could use  $a_{it} = \frac{1}{T}$  and get bag of words

## Combining vectors with attention

- Introducing matrix  $\mathbf{W} \in \mathbb{R}^{d \times T}$  where columns correspond to  $\mathbf{w}_i$ ,

$$\mathbf{h}_t = \mathbf{VW}\mathbf{a}_t$$

- And finally

$$\mathbf{H} = \mathbf{VWA}$$

## Combining vectors with attention

- How to compute the matrix **A**?

$$\mathbf{A} = \text{softmax}(\mathbf{W}^\top \mathbf{K}^\top \mathbf{Q} \mathbf{W})$$

where the softmax is applied column-wise.

- Why softmax? to get positive entries, and columns summing to 1.
- Why  $\mathbf{W}^\top \mathbf{K}^\top \mathbf{Q} \mathbf{W}$ ? Bilinear form over the input

# Combining vectors with attention

- Putting everything together:

$$\mathbf{H} = \mathbf{V}\mathbf{W}\text{softmax}(\mathbf{W}^\top \mathbf{K}^\top \mathbf{Q}\mathbf{W})$$

where  $\mathbf{H}, \mathbf{W} \in \mathbb{R}^{d \times T}$  and  $\mathbf{V}, \mathbf{K}, \mathbf{Q} \in \mathbb{R}^{d \times d}$

- $\mathbf{V}, \mathbf{K}, \mathbf{Q}$  are parameters to be learned.
- This operation is called self-attention
- It can be generalized to **multiple heads**:
  - Split input vectors into  $n$  subvectors of dimension  $d/n$ ,
  - Apply self attention (with different  $\mathbf{V}, \mathbf{K}, \mathbf{Q}$ ) over these smaller vectors
  - Concatenate the results to get back  $d$  dimensional vectors

# Transformer network

Transformer block:

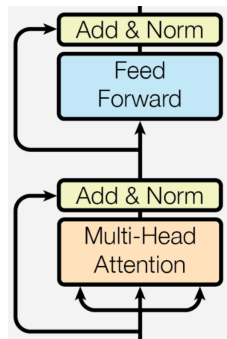
- Multi-head attention layer with skip connection and normalization
- Followed by feed forward with skip connection and normalization

Skip connection+normalization:

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

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

where **norm** normalize the input



Vaswani et al.  
(2017)

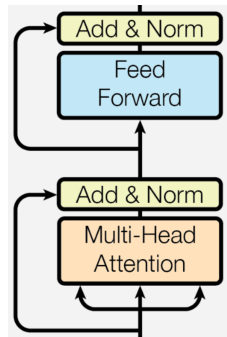
# Transformer network

## Feed forward block

- Two layer network, with ReLU activation

$$\mathbf{y} = \mathbf{W}_2 \text{ReLU}(\mathbf{W}_1 \mathbf{x})$$

- Usually,  $\mathbf{W}_1 \in \mathbb{R}^{4d \times d}$  and  $\mathbf{W}_2 \in \mathbb{R}^{d \times 4d}$
- 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
- **Solution:** add position encodings.
- Replace the matrix  $\mathbf{W}$  by  $\mathbf{W} + \mathbf{E}$ , where  $\mathbf{E} \in \mathbb{R}^{d \times T}$
- $\mathbf{E}$  can be learned, or defined using sin and cos:

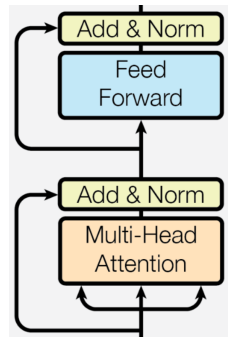
$$e_{2i,j} = \sin\left(\frac{j}{10000^{2i/d}}\right)$$
$$e_{2i+1,j} = \cos\left(\frac{j}{10000^{2i/d}}\right)$$



# 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)

# Masking for Transformer Language Models

- In transformer,  $\mathbf{h}_t$  depends on **all** inputs
- Could not be used as is for language modeling
- Solution: use mask in attention, to only use past
  
- Reminder:

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

Hence,  $\mathbf{a}_{it}$  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  $\mathbf{a}_{it} = 0$  for  $i \geq t$

# Masked softmax

- We introduce the masked softmax operator
- Given an input  $\mathbf{x}$  and a binary mask  $\mathbf{m}$ ,

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

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

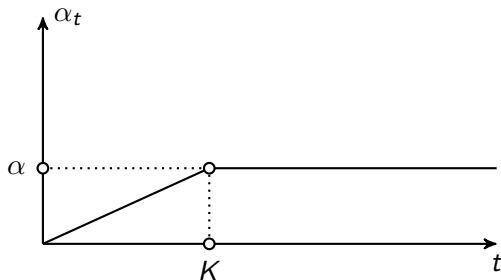
$$\text{softmax}(\mathbf{x} + \log(\mathbf{m}))$$

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

# 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  $\alpha$

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

where  $K$  is the “warm-up” parameter

# 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 (Wang et al., 2018) 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.

## 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 | <b>80.5</b> |

- 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



# Conclusion

- Neural networks have been very successful in language modeling
- They are also dominant in applications (machine translation sentence representation...)
- Big progress in architectures until recently with early fusion and transformers

# 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)*.
- Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.
- 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*.

## References II

- Press, O. and Wolf, L. (2016). Using the output embedding to improve language models. *arXiv preprint arXiv:1608.05859*.
- 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.
- Taylor, W. L. (1953). Cloze procedure: A new tool for measuring readability. *Journalism Bulletin*, 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.
- Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. (2018). Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.