NLP

Notes from Speech & Language Processing

Regular Expressions

- an algebraic notation for characterizing a set of strings.
- we have a pattern to search for and a corpus of texts to search through

Basic RE patterns

// - simplest kind of regular expression is a sequence of simple characters.

RE	Example Patterns Matched
/woodchucks/	"interesting links to woodchucks and lemurs"
/a/	"Mary Ann stopped by Mona's"
/!/	"You've left the burglar behind again!" said Nori

- **Figure 2.1** Some simple regex searches.

[] – square braces. The string of characters inside the braces specifies a disjunction of characters to match.

RE	Match	Example Patterns
/[wW]oodchuck/	Woodchuck or woodchuck	"Woodchuck"
/[abc]/	'a', 'b', *or* 'c'	"In uomini, in soldati"
/[1234567890]/	any digit	"plenty of 7 to 5"

- **Figure 2.2** The use of the brackets [] to specify a disjunction of characters.

RE	Match	Example Patterns Matched
/[A-Z]/	an upper case letter	"we should call it 'Drenched Blossoms' "
/[a-z]/	a lower case letter	"my beans were impatient to be hoed!"
/[0-9]/	a single digit	"Chapter 1: Down the Rabbit Hole"

- **Figure 2.3** The use of the brackets [] plus the dash - to specify a range.

- Using caret ^ to ignore characters:

RE	Match (single characters)	Example Patterns Matched
/[^A-Z]/	not an upper case letter	"Oyfn pripetchik"
/[^Ss]/	neither 'S' nor 's'	"I have no exquisite reason for't"
/[^.]/	not a period	"our resident Djinn"
/[e^]/	either 'e' or '^'	"look up ^ now"
/a^b/	the pattern 'a^b'	"look up a^ b now"

Figure 2.4 The caret ^ for negation or just to mean ^. See below re: the backslash for escaping the period.

? - means "the preceding character or nothing". It is an optional character.

RE	Match	Example Patterns Matched
/woodchucks?/	woodchuck or woodchucks	"woodchuck"
/colou?r/	color or colour	"color"

- **Figure 2.5** The question mark ? marks optionality of the previous expression.

* - commonly called the Kleene * (generally pronounced "cleany star"). "zero or more occurrences of the immediately previous character or regular expression". /[ab]*/ will match strings like *aaaa or ababab* or *bbbb*.

+ *Kleene* +, which means "one or more occurrences of the immediately preceding character or regular expression". /[0-9]+/ is the normal way to specify "a sequence of digits".

. wildcard expression, that matches any single character (except a carriage return)

RE	Match	Example Matches
/beg.n/	any character between *beg* and *n*	begin, beg'n, begun

Figure 2.6 The use of the period . to specify any character.

- suppose we want to find any line in which a particular word, for example, aardvark, appears twice. We can specify this with the regular expression /aardvark.*aardvark/

Anchors - the caret ˆ and the dollar sign $

- The caret ˆ has three uses:
  - to match the start of a line,
  - to indicate a negation inside of square brackets,
  - and just to mean a caret.
- The dollar sign $ matches the end of a line.
  - the pattern ␣$ is a useful pattern for matching a space at the end of a line, and /ˆThe dog\.$/ matches a line that contains only the phrase The dog. (We have to use the backslash here since we want the . to mean "period" and not the wildcard.

RE	Match
ˆ	start of line
$	end of line
\b	word boundary
\B	non-word boundary

Figure 2.7 Anchors in regular expressions.

  - \b matches a word boundary - /\bthe\b/ matches the word the but not the word other
  - \B matches a non-boundary.

Disjunction, Grouping, and Precedence

- the **disjunction operator**, also called the pipe symbol |. The pattern /cat|dog/ matches either the string cat or the string dog.
- How can I specify both guppy and guppies?
  - /guppy|ies/, cant work because that would match only the strings guppy and ies.
- Precedence - To make the disjunction operator apply only to a specific pattern, we need to use the parenthesis operators (and)
  - /gupp(y|ies)/ would specify that we meant the disjunction only to apply to the suffixes y and ies.
- Matching repeated text in a form, example - *Column 1 Column 2 Column 3*
  - /Column␣[0-9]+␣*/ will not match any column, instead it will match a single column followed by any number of spaces!
  - The star here applies only to the space ␣ that precedes it, not to the whole sequence.
  - /(Column␣ [0-9]+ ␣*)*/ to match the word Column, followed by a number and optional spaces, the whole pattern repeated zero or more times.
- Operator precedence

Parenthesis	()
Counters	* + ? {}
Sequences and anchors	the ^my end$
Disjunction	|

- Reducing the overall error rate for an application thus involves two antagonistic efforts:
  - Increasing precision (minimizing false positives)
  - Increasing recall (minimizing false negatives)

More Operators

.{} – gives the counted number of occurrences

- /{n,m}/ specifies from n to m occurrences of the previous char or expression, and /{n,}/ means at least n occurrences of the previous expression.

RE	Expansion	Match	First Matches
\d	[0-9]	any digit	Party␣of␣5
\D	[^0-9]	any non-digit	Blue␣moon
\w	[a-zA-Z0-9_]	any alphanumeric/underscore	Daiyu
\W	[^\w]	a non-alphanumeric	!!!!
\s	[␣\r\t\n\f]	whitespace (space, tab)	
\S	[^\s]	Non-whitespace	in␣Concord

Figure 2.8 Aliases for common sets of characters.

RE	Match
*	zero or more occurrences of the previous char or expression
+	one or more occurrences of the previous char or expression
?	exactly zero or one occurrence of the previous char or expression
{n}	*n* occurrences of the previous char or expression
{n,m}	from *n* to *m* occurrences of the previous char or expression
{n,}	at least *n* occurrences of the previous char or expression
{,m}	up to *m* occurrences of the previous char or expression

Figure 2.9 Regular expression operators for counting.

RE	Match	First Patterns Matched
*	an asterisk "*"	"K*A*P*L*A*N"
\.	a period "."	"Dr. Livingston, I presume"
\?	a question mark	"Why don't they come and lend a hand?"
\n	a newline	
\t	a tab	

Figure 2.10 Some characters that need to be backslashed.

More complex RE

We plan to denote the dollars here:

- /$[0-9]+/ → $ character has a different function here than the end-of-line f() we discussed earlier.
- `/$[0-9]+\.[0-9][0-9]/` → deals with the decimal values now. Matches $199.99 but not $199
- `/(^|\W)$[0-9]+(\.[0-9][0-9])?\b/` → but this allows very high prices like $19999999.99, so to set some limits, we use the next step.
- `/(^|\W)$[0-9]{0,3}(\.[0-9][0-9])?\b/` → does the work.

Substitutionn, Capture groups & ELIZA

Substitution →

```
/the (.*)er they were, the \1er they will be/
```

- Here the \1 will be replaced by whatever string matched the first item in parentheses. So, this will match *the **bigger** they were, the **bigger** they will be*

but not *the **bigger** they were, the **faster** they will be*.

Capture Group →

```
/the (.*)er they (.*), the \1er we \2/
```

will match *the **faster** they **ran**, the **faster** we **ran***,
but not *the **faster** they **ran**, the **faster** we **ate***.

- This use of parentheses to store a pattern in memory is called a capture group.
- Every time a capture group is used (i.e., parentheses surround a pattern), the resulting match is stored in a numbered register.

Parentheses thus have a double function in regular expressions; they are used to group terms for specifying the order in which operators should apply, and they are used to capture something in a register.

Occasionally we might want to use parentheses for grouping, but don't want to capture the resulting pattern in a register. In that case we use a **non-capturing group**, which is specified by putting the commands ?: after the open paren, in the form (?: pattern).

Non-Capturing Groups →

```
/(?:some|a few) (people|cats) like some \1/
```

will match *some cats like some cats*
but not *some cats like some a few*.

Using these in ELISA:

```
s/.* I'M (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/
s/.* I AM (depressed|sad) .*/WHY DO YOU THINK YOU ARE \1/
s/.* all .*/IN WHAT WAY/
s/.* always .*/CAN YOU THINK OF A SPECIFIC EXAMPLE/
```

Lookahead & Assertions

These lookahead assertions make use of the (? syntax that we saw in the previous section for non-capture groups. The operator (?= *pattern*) is true if pattern occurs, but is zero-width, i.e. the match pointer doesn't advance. The operator (?! *pattern*) only returns true if a pattern does not match, but again is zero-width and doesn't advance the cursor.

- For example suppose we want to match, at the beginning of a line, any single word that doesn't start with "Volcano". We can use negative lookahead to do this:
  /^(?!Volcano)[A-Za-z]+/

Words

"I do uh main- mainly business data processing"

Disfluencies →

Fragment – The broken-off words (main-)

Filler/ filler pauses – words like uh and um (uh)

- in speech recognition in predicting the upcoming word, because they may signal that the speaker is restarting the clause or idea, and so for speech recognition they are treated as regular words.
- Depends on context whether to use them or ignore.

Lemma →

A lemma is a set of lexical forms having the same stem, the same major part-of-speech, and the same word sense

Wordform →

The wordform is the full inflected or derived form of the word.

How many words are there? We can find using two ways:

Word types →

Types are the number of distinct words in a corpus; if the set of words in the vocabulary is V, the number of types is the vocabulary size |V|.

Word Tokens →

Tokens are the total number N of running words.

Herden's Law/ Heaps' law→

The larger corpora we have, more word types we find. So there is relation between |V| & N.

$$|V| = kN^{\beta}$$

where k and β are positive constants, and $0 < \beta < 1$

β – depends on the corpus size and genre.

Corpora

The world has 7097 languages. It is important to test algorithms on more than one language, and particularly on languages with different properties.

Even when algorithms are developed beyond English, they tend to be developed for the official languages of large, industrialized nations (Chinese, Spanish, Japanese, German etc.).
The same language can have multiple varities based on the *region* and *social groups*.

Code switching - to use multiple languages in a single communicative act.

Text also reflects the demographic characteristics of the writer (or speaker): their age, gender, race, socioeconomic class can all influence the linguistic properties of the text we are processing.

Language changes over time, and for some languages we have good corpora of texts from different historical periods.

Building corpus → building a datasheet.
Datasheet:

- Motivation.
- Situation.
- Language variety.
- Speaker demographics
- Collection process
- Annotation process
- Distribution.

Text Normalization

At least three tasks are commonly applied as part of any normalization process:

1. Tokenizing (segmenting) words
2. Normalizing word formats
3. Segmenting sentences

Word Tokenization - the task of segmenting running text into words.

> We often want to break off punctuation as a separate token; commas are a useful piece of information for parsers, periods help indicate sentence boundaries. But we'll often want to keep the punctuation that occurs word internally, in examples like m.p.h., Ph.D., AT&T, and cap'n. Special characters and numbers will need to be kept in prices ($45.55) and dates (01/02/06); we don't want to segment that price into separate tokens of "45" and "55". And there are URLs (http://www.stanford.edu), Twitter hashtags (#nlproc), or email addresses (someone@cs.colorado.edu). Number expressions introduce other complications as well; while commas normally appear at word boundaries, commas are used inside numbers in English, every three digits: 555,500.50. Languages, and hence tokenization requirements, differ on this; many continental European languages like Spanish, French, and German, by contrast, use a comma to mark the decimal point, and spaces (or sometimes periods) where English puts commas, for example, 555 500,50.

Sandeep Kumar Kushwaha

A tokenizer can also be used to expand **clitic** contractions that are marked by apostrophes, for example, converting *what're* to the two tokens *what are*, and *we're* to *we are*.

Penn Treebank tokenization standard

used for the parsed corpora (treebanks) released by the Linguistic Data Consortium (LDC), the source of many useful datasets.

This standard separates out clitics (doesn't becomes does plus n't), keeps hyphenated words together, and separates out all punctuation.

Input: "The San Francisco-based restaurant," they said, "doesn't charge $10".

Output: "␣The␣San␣Francisco-based␣restaurant␣,␣"␣they␣said␣,␣ "␣does␣n't␣charge␣$␣10␣"␣.

- In practice, tokenization needs to be run before any other language processing, it needs to be very fast.
- The standard method for tokenization is therefore to use deterministic algorithms based on RE compiled into very efficient finite state automata.

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens; includes ], [
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Figure 2.12 A Python trace of regular expression tokenization in the NLTK Python-based natural language processing toolkit (Bird et al., 2009), commented for readability; the (?x) verbose flag tells Python to strip comments and whitespace. Figure from Chapter 3 of Bird et al. (2009).

Byte-Pair Encoding for Tokenization

Instead of defining tokens as words we can use our data to automatically tell us what the tokens should be.

- NLP algorithms often learn some facts about language from one corpus (a training corpus) and then use these facts to make decisions about a separate test corpus and its language. Thus if our training corpus contains, say the words *low, new, newer*, but not *lower*, then if the word *lower* appears in our test corpus, our system will not know what to do with it.
  - To deal with this unknown word problem, modern tokenizers often automatically induce sets of tokens that include tokens smaller than words, called **subwords**.

- Subwords can be arbitrary substrings, or they can be meaning-bearing units like the morphemes -est or -er. (A morpheme is the smallest meaning-bearing unit of a language; for example the word *unlikeliest* has the morphemes *un-*, likely, and *-est*.)

- Two parts of tokenizer: a **token learner**, and a **token segmenter**.
  - a **token learner** - takes a raw training corpus and induces a vocabulary, a set of tokens.
  - a **token segmenter** - takes a raw test sentence and segments it into the tokens in the vocabulary.

- 3 algorithms are used:
  - **Byte-pair encoding:** BPE token learner begins with a vocabulary that is just the set of all individual characters. It then examines the training corpus, chooses the two symbols that are most frequently adjacent (say 'A', 'B'), adds a new merged symbol 'AB' to the vocabulary, and replaces every adjacent 'A' 'B' in the corpus with the new 'AB'. It continues to count and merge, creating new longer and longer character strings, until k merges have been done creating k novel tokens; k is thus a parameter of the algorithm

    The resulting vocabulary consists of the original set of characters plus k new symbols.

counts. Let's see its operation on the following tiny input corpus of 18 word tokens with counts for each word (the word *low* appears 5 times, the word *newer* 6 times, and so on), which would have a starting vocabulary of 11 letters:

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w
2 l o w e s t _	
6 n e w e r _	
3 w i d e r _	
2 n e w _	

The BPE algorithm first counts all pairs of adjacent symbols: the most frequent is the pair `e r` because it occurs in *newer* (frequency of 6) and *wider* (frequency of 3) for a total of 9 occurrences[1]. We then merge these symbols, treating `er` as one symbol, and count again:

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er
2 l o w e s t _	
6 n e w er _	
3 w i d er _	
2 n e w _	

Now the most frequent pair is `er _`, which we merge; our system has learned that there should be a token for word-final `er`, represented as `er_`:

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_
2 l o w e s t _	
6 n e w er_	
3 w i d er_	
2 n e w _	

Next n e (total count of 8) get merged to ne:

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne
2 l o w e s t _	
6 ne w er_	
3 w i d er_	
2 ne w _	

If we continue, the next merges are:

Merge	Current Vocabulary
(ne, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new
(l, o)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo
(lo, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low
(new, er_)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_
(low, _)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_, low_

function BYTE-PAIR ENCODING(strings C, number of merges k) **returns** vocab V

$V \leftarrow$ all unique characters in C # initial set of tokens is characters
for $i = 1$ **to** k **do** # merge tokens til k times
  $t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C
  $t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating
  $V \leftarrow V + t_{NEW}$ # update the vocabulary
  Replace each occurrence of t_L, t_R in C with t_{NEW} # and update the corpus
return V

Figure 2.13 The token learner part of the BPE algorithm for taking a corpus broken up into individual characters or bytes, and learning a vocabulary by iteratively merging tokens. Figure adapted from Bostrom and Durrett (2020).

- **unigram language modelling.**
- **WordPiece.**
- **SentencePiece** library that includes implementations of the first two of the three.

Word Normalization, Lemmatization and Stemming

Word normalization →

Putting words/tokens in a standard format, choosing a single normal form for words with multiple forms. [*USA* and *US* or *uh-huh* and *uhhuh*.]

Case folding →

Mapping everything to lowercase means that Woodchuck and woodchuck are represented identically, which is very helpful for generalization in many tasks

Not done in case of sentiment analysis and other text classification tasks, information extraction, and machine translation, in fact, case can be quite helpful and case folding is generally not done.

Lemmatization →

task of determining that two words have the same root, despite their surface differences. The words *am*, *are*, and *is* have the shared lemma *be*. the words *dinner* and *dinners* both have the lemma *dinner.*

- How to do lemmatization?
  - morphological parsing - is the study of the way words are built up from smaller meaning-bearing units called **morphemes.**
  - Morphemes has two classes:
    - Stems: contains the main meaning.
    - Affixes: adds additional meaning of various kinds.
  - Example: *cats* has 2 morphemes – *cat* and *-s.*

The Porter Stemmer

- simpler but cruder method, which mainly consists of chopping off word-final affixes.
- This naive version of morphological analysis is called stemming.

```
This was not the map we found in Billy Bones's chest, but
an accurate copy, complete in all things-names and heights
and soundings-with the single exception of the red crosses
and the written notes.
```

produces the following stemmed output:

```
Thi wa not the map we found in Billi Bone s chest but an
accur copi complet in all thing name and height and sound
with the singl except of the red cross and the written note
```

- Following rewrite rules are used for modification

ATIONAL → ATE (e.g., relational → relate)
ING → ϵ if stem contains vowel (e.g., motoring → motor)
SSES → SS (e.g., grasses → grass)

Sentence Segmentation

- The most useful cues for segmenting a text into sentences are punctuation, like periods, question marks, and exclamation points.
- Question marks and exclamation points are relatively unambiguous markers of sentence boundaries.
- Periods, on the other hand, are more ambiguous. "." is ambiguous between a sentence boundary marker and a marker of abbreviations like *Mr*. or *Inc.*
- So, sentence tokenization and word tokenization may be addressed jointly.

Minimum Edit Distance

- measuring how similar two strings are.
- graffe & giraffe – only one letter difference.

Stanford President Marc Tessier-Lavigne
Stanford University President Marc Tessier-Lavigne

- **Edit distance** gives us a way to quantify both of these intuitions about string similarity. More formally, the **minimum edit distance** between two strings is definedas the minimum number of editing operations (operations like insertion, deletion, substitution) needed to transform one string into another.
- Given two sequences, an **alignment** is a correspondence between substrings of the two sequences.

Figure 2.14 Representing the minimum edit distance between two strings as an **alignment**. The final row gives the operation list for converting the top string into the bottom string: d for deletion, s for substitution, i for insertion.

- Levenshtein distance –
  Assigning weights to these operations. Each operation has cost 1. So above the cost is 5.
  Another alternate to this:
  - each insertion or deletion has a cost of 1
  - substitutions are not allowed.

  This above is equivalent to allowing substitution, but giving each substitution a cost of 2 since any substitution can be represented by one insertion and one deletion

$$D[i,j] = \min \begin{cases} D[i-1,j]+1 \\ D[i,j-1]+1 \\ D[i-1,j-1] + \begin{cases} 2; & \text{if } source[i] \neq target[j] \\ 0; & \text{if } source[i] = target[j] \end{cases} \end{cases}$$

- The Minimum Edit Distance Algorithm –

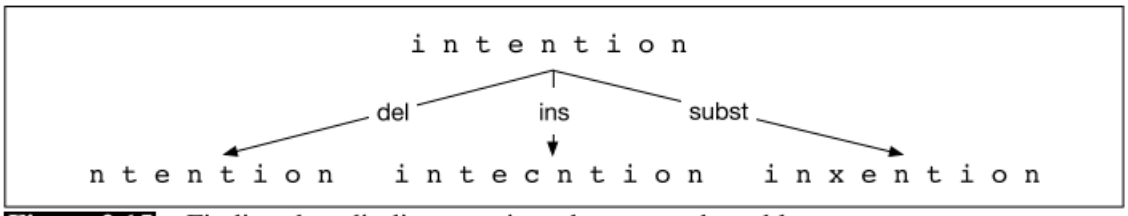


Figure 2.15 Finding the edit distance viewed as a search problem

The space of all possible edits is enormous, so we can't search naively.

However, lots of distinct edit paths will end up in the same state (string), so rather than recomputing all those paths, we could just remember the shortest path to a state each time dynamic we saw it. We can do this by using dynamic programming.

DP -- that apply a table-driven method to solve problems by combining solutions to sub-problems. Algos using DP - *Viterbi algorithm*. *CKY algo*.

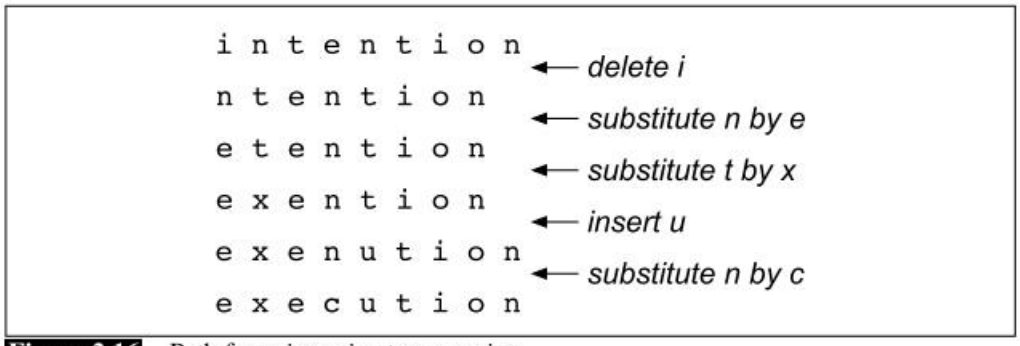


Figure 2.16 Path from *intention* to *execution*.

```
function MIN-EDIT-DISTANCE(source, target) returns min-distance

  n ← LENGTH(source)
  m ← LENGTH(target)
  Create a distance matrix distance[n+1,m+1]

  # Initialization: the zeroth row and column is the distance from the empty string
  D[0,0] = 0
  for each row i from 1 to n do
    D[i,0] ← D[i-1,0] + del-cost(source[i])
  for each column j from 1 to m do
    D[0,j] ← D[0,j-1] + ins-cost(target[j])

  # Recurrence relation:
  for each row i from 1 to n do
      for each column j from 1 to m do
        D[i,j] ← MIN( D[i−1,j] + del-cost(source[i]),
                      D[i−1,j−1] + sub-cost(source[i], target[j]),
                      D[i,j−1] + ins-cost(target[j]))
  # Termination
  return D[n,m]
```

Figure 2.17 The minimum edit distance algorithm, an example of the class of dynamic programming algorithms. The various costs can either be fixed (e.g., $\forall x, \text{ins-cost}(x) = 1$) or can be specific to the letter (to model the fact that some letters are more likely to be inserted than others). We assume that there is no cost for substituting a letter for itself (i.e., $\text{sub-cost}(x,x) = 0$).

- Definition of MEDA –
  Given two strings, the source string X of length n, and target string Y of length m, we'll define D[i, j] as the edit distance between X[1..i] and Y[1.. j], i.e., the first i characters of X and the first j characters ofY. The edit distance between X and Y is thus D[n,m]

$$D[i,j] = \min \begin{cases} D[i-1,j] + \text{del-cost}(source[i]) \\ D[i,j-1] + \text{ins-cost}(target[j]) \\ D[i-1,j-1] + \text{sub-cost}(source[i], target[j]) \end{cases}$$

  - MEDA is useful for finding error corrections.
  - Helps in finding the alignment between two string.

- Alignment –
  In speech recognition, minimum edit distance alignment is used to compute the word error rate.

  Alignment plays a role in machine translation, in which sentences in a parallel corpus (a corpus with a text in two languages) need to be matched to each other.

Figure 2.19 also shows the intuition of how to compute this alignment path. The computation proceeds in two steps. In the first step, we augment the minimum edit distance algorithm to store backpointers in each cell. The backpointer from a cell points to the previous cell (or cells) that we came from in entering the current cell. We've shown a schematic of these backpointers in Fig. 2.19. Some cells have multiple backpointers because the minimum extension could have come from multiple previous cells. In the second step, we perform a **backtrace**. In a backtrace, we start from the last cell (at the final row and column), and follow the pointers back through the dynamic programming matrix. Each complete path between the final cell and the initial cell is a minimum distance alignment. Exercise 2.7 asks you to modify the minimum edit distance algorithm to store the pointers and compute the backtrace to output an alignment.

	#	e	x	e	c	u	t	i	o	n
#	0	← 1	← 2	← 3	← 4	← 5	← 6	← 7	← 8	← 9
i	↑ **1**	↖←↑ 2	↖←↑ 3	↖←↑ 4	↖←↑ 5	↖←↑ 6	↖←↑ 7	↖ 6	← 7	← 8
n	↑ 2	↖←↑ **3**	↖←↑ 4	↖←↑ 5	↖←↑ 6	↖←↑ 7	↖←↑ 8	↑ 7	↖←↑ 8	↖ 7
t	↑ 3	↖←↑ 4	↖←↑ **5**	↖←↑ 6	↖←↑ 7	↖←↑ 8	↖ 7	←↑ 8	↖←↑ 9	↑ 8
e	↑ 4	↖ 3	← 4	↖← **5**	← **6**	← 7	←↑ 8	↖←↑ 9	↖←↑ 10	↑ 9
n	↑ 5	↑ 4	↖←↑ 5	↖←↑ 6	↖←↑ 7	↖←↑ **8**	↖←↑ 9	↖←↑ 10	↖←↑ 11	↖↑ 10
t	↑ 6	↑ 5	↖←↑ 6	↖←↑ 7	↖←↑ 8	↖←↑ 9	↖ **8**	← 9	← 10	←↑ 11
i	↑ 7	↑ 6	↖←↑ 7	↖←↑ 8	↖←↑ 9	↖←↑ 10	↑ 9	↖ **8**	← 9	← 10
o	↑ 8	↑ 7	↖←↑ 8	↖←↑ 9	↖←↑ 10	↖←↑ 11	↑ 10	↑ 9	↖ **8**	← 9
n	↑ 9	↑ 8	↖←↑ 9	↖←↑ 10	↖←↑ 11	↖←↑ 12	↑ 11	↑ 10	↑ 9	↖ **8**

Figure 2.19 When entering a value in each cell, we mark which of the three neighboring cells we came from with up to three arrows. After the table is full we compute an **alignment** (minimum edit path) by using a **backtrace**, starting at the **8** in the lower-right corner and following the arrows back. The sequence of bold cells represents one possible minimum cost alignment between the two strings. Diagram design after Gusfield (1997).

N – Gram Language Models

Probabilities are also important for augmentative and alternative communication systems.

People often use such AAC devices if they are physically unable to speak or sign but can instead use eye gaze or other specific movements to select words from a menu to be spoken by the system. Word prediction can be used to suggest likely words for the menu.

Models that assign probabilities to sequences of words are called language models or LMs.

N-gram

- sequence of n words.
- the task of computing P(w|h), the probability of a word w given some history h.
- Suppose the history h is "*its water is so transparent that*" and we want to know the probability that the next word is *the*:

$$P(the|its\ water\ is\ so\ transparent\ that).$$

  - Method 1 — estimate this probability is from relative frequency counts.
    count the number of times we see its water is so transparent that, and count the number of times this is followed by the. This would be answering the question "Out of the times we saw the history h, how many times was it followed by the word w", as follows:

$$P(the|its\ water\ is\ so\ transparent\ that) = \frac{C(its\ water\ is\ so\ transparent\ that\ the)}{C(its\ water\ is\ so\ transparent\ that)}$$

    - Drawbacks –
      - Language is creative.
      - New sentences are created all time.
      - we won't always be able to count entire sentences
  - Method 2 –
    To represent the probability of a particular random variable *Xi* taking on the value "the", or P(Xi = "the"), we will use the simplification P(the).
    We'll represent a sequence of N words either as w1 . . .wn or w1:n (so the expression w1:n−1 means the string w1,w2, ...,wn−1).
    For the joint probability of each word in a sequence having a particular value P(X =w1,Y = w2,Z = w3, ...,W = wn) we'll use P(w1,w2, ...,wn).

    To compute probabilities of entire sequences like P(w1,w2, ...,wn), we can decompose this probability using the **chain rule of probability**:

$$\begin{aligned} P(X_1 \ldots X_n) &= P(X_1)P(X_2|X_1)P(X_3|X_{1:2}) \ldots P(X_n|X_{1:n-1}) \\ &= \prod_{k=1}^{n} P(X_k|X_{1:k-1}) \end{aligned}$$

    Applying the chain rule to words, we get

$$\begin{aligned} P(w_{1:n}) &= P(w_1)P(w_2|w_1)P(w_3|w_{1:2}) \ldots P(w_n|w_{1:n-1}) \\ &= \prod_{k=1}^{n} P(w_k|w_{1:k-1}) \end{aligned}$$

The bigram model, for example, approximates the probability of a word given all the previous words P(wn|w1:n−1) by using only the conditional probability of the preceding word P(wn|wn−1).

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-1})$$

- The assumption that the probability of a word depends only on the previous word is called a **Markov assumption.**
- Markov model are probabilistic models that assume we can predict the probability of future unit without looking too far in the past. E.g. bigram & trigram.
- General equation of n-gram:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-N+1:n-1}) \quad (3.8)$$

– **Maximum likelihood estimation** or **MLE** -- estimate these bigrams or n-gram probabilities.
  - We get the MLE estimate for the parameters of an n-gram model by getting counts from a corpus, and normalizing the counts so that they lie between 0 and 1.
  - E.g. to compute a particular bigram probability of a word y given a previous word x, we'll compute the count of the bigram C(xy) and normalize by the sum of all the bigrams that share the same first word x:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)}$$

  - Instead of focusing on the probability of word in the whole corpus, we use something as shown:

	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

Figure 3.1 Bigram counts for eight of the words (out of $V = 1446$) in the Berkeley Restaurant Project corpus of 9332 sentences. Zero counts are in gray.

This can provide probability as follows:

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	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

Figure 3.2 Bigram probabilities for eight words in the Berkeley Restaurant Project corpus of 9332 sentences. Zero probabilities are in gray.

When using higher n-gram models, we have to consider more probabilities, and multiplication of these can cause numeric underflow.
So, we use logarithms because we get numbers that are not as small. Adding in log space is equivalent to multiplying in linear space, so we combine log probabilities by adding them. The result of doing all computation and storage in log space is that we only need to convert back into probabilities if we need to report them at the end; then we can just take the exp of the logprob:

$$p_1 \times p_2 \times p_3 \times p_4 = \exp(\log p_1 + \log p_2 + \log p_3 + \log p_4)$$

Evaluating the language models

Extrinsic evaluation:

to evaluate the performance of a language model - embed it in an application and measure how much the application improves. This is an end-to-end evaluation.

Unfortunately, running big NLP systems end-to-end is often very expensive.

Intrinsic evaluation:

An intrinsic evaluation metric is one that measures the quality of a model independent of any application.

So if we are given a corpus of text and want to compare two different n-gram models,

- we divide the data into training and test sets,
- train the parameters of both models on the training set,
- and then compare how well the two trained models fit the test set.

"fit the test set" → whichever model assigns a higher probability to the test set—meaning it more accurately predicts the test set—is a better model.

Perplexity (PP)

In practice we don't use raw probability as our metric for evaluating language models, but a variant called perplexity. For a test set W = w1w2 . . .wN,

$$\begin{aligned}\text{PP}(W) &= P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}}\end{aligned}$$

using chain rule, we get:

$$\text{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1 \ldots w_{i-1})}}$$

computing PP(W) with a bigram language model we get:

$$\text{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}}$$

Since there is the inverse, the higher the conditional probability of the word sequence, the lower the PP.

PP can also be represented as, weighted average branching factor of a language. The branching factor of a language is the number of possible next words that can follow any word. E.g. for the task of recognising the digits 0-9 given each digit occurs with = probability (1/10). The PP for this mini-language is 10 as shown below:

$$\begin{aligned}\text{PP}(W) &= P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} \\ &= \left(\frac{1}{10}^{N}\right)^{-\frac{1}{N}} \\ &= \frac{1}{10}^{-1} \\ &= 10\end{aligned}$$

But suppose when 0 is more frequent than others, the perplexity of this test set is lower since most of the time the next num will be zero which is very predictable.

Thus, although the branching factor is still 10, the perplexity or weighted branching factor is smaller

PP is also closely related to the information-theoretic notion of entropy.

The table below shows the perplexity of a 1.5 million word WSJ test set according to each of these grammars. the more information the n-gram gives us about the word sequence, the lower the perplexity.

	Unigram	Bigram	Trigram
Perplexity	962	170	109

The perplexity of two language models is only comparable if they use identical vocabularies.

An (intrinsic) improvement in perplexity does not guarantee an (extrinsic) improvement in the performance of a language processing task like speech recognition or machine translation. Nonetheless, because perplexity often correlates with such improvements, it is commonly used as a quick check on an algorithm. But a model's improvement in perplexity should always be confirmed by an end-to-end evaluation of a real task before concluding the evaluation of the model.

Sampling sentences from a language model

Sampling from a distribution means to choose random points according to their likelihood.

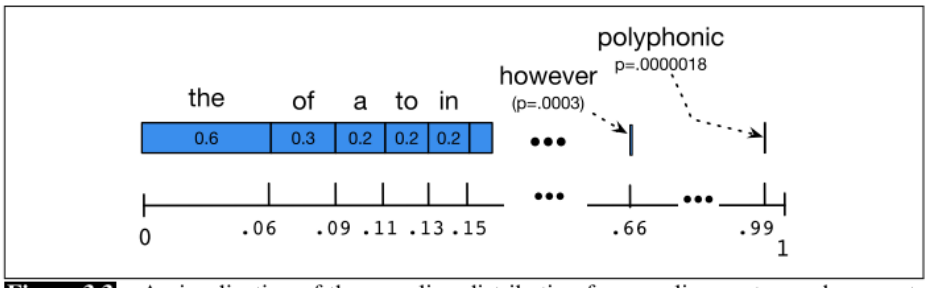


Figure 3.3 A visualization of the sampling distribution for sampling sentences by repeatedly sampling unigrams. The blue bar represents the frequency of each word. The number line shows the cumulative probabilities. If we choose a random number between 0 and 1, it will fall in an interval corresponding to some word. The expectation for the random number to fall in the larger intervals of one of the frequent words (*the*, *of*, *a*) is much higher than in the smaller interval of one of the rare words (*polyphonic*).

Generalization and Zeros

1 gram	–To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have –Hill he late speaks; or! a more to leg less first you enter
2 gram	–Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow. –What means, sir. I confess she? then all sorts, he is trim, captain.
3 gram	–Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done. –This shall forbid it should be branded, if renown made it empty.
4 gram	–King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in; –It cannot be but so.

Figure 3.4 Eight sentences randomly generated from four n-grams computed from Shakespeare's works. All characters were mapped to lower-case and punctuation marks were treated as words. Output is hand-corrected for capitalization to improve readability.

1 gram	Months the my and issue of year foreign new exchange's september were recession exchange new endorsed a acquire to six executives
2 gram	Last December through the way to preserve the Hudson corporation N. B. E. C. Taylor would seem to complete the major central planners one point five percent of U. S. E. has already old M. X. corporation of living on information such as more frequently fishing to keep her
3 gram	They also point to ninety nine point six billion dollars from two hundred four oh six three percent of the rates of interest stores as Mexico and Brazil on market conditions

Figure 3.5 Three sentences randomly generated from three n-gram models computed from 40 million words of the *Wall Street Journal*, lower-casing all characters and treating punctuation as words. Output was then hand-corrected for capitalization to improve readability.

Since, both of them are trained on English-like sentences but different corpora. There is no overlap in the generated sentences. So statistical models are likely to be pretty useless as predictors if the training set and test sets are different as Shakespeare and WSJ.

So choice of genre is important, we pick the training corpus based upon use-case.

Matching genres and dialects is still not sufficient. Our models may still be subject to the problem of **sparsity**.

Suppose our training corpus had these words:

denied the allegations:	5
denied the speculation:	2
denied the rumors:	1
denied the report:	1

But suppose our test set has phrases like:

1. denied the offer
2. denied the loan

Our model will incorrectly estimate that the P(offer|denied the) is 0! This is *zero probability n-grams* problem. This can occur due to two reasons:

- their presence means we are underestimating the probability of all sorts of words that might occur
- if the probability of any word in the test set is 0, the entire probability of the test set is 0

In such cases we cannot calculate the PP, as we can't divide by 0!

Unknown words

In a **closed vocabulary** system the test set can only contain words from this lexicon, and there will be no unknown words.

In other cases we have to deal with unseen words, (unknown words) ~ out of vocabulary (OOV) words.

The percentage of OOV words that appear in the test set is called the OOV rate.

An open vocabulary system is one in which we model these potential unknown words in the test set by adding a pseudo-word called <UNK>

Ways to train the probabilities of the unknown word model <UNK>

- turn the problem back into a closed vocabulary one by choosing a fixed vocabulary in advance:
  1. **Choose a vocabulary** (word list) that is fixed in advance.
  2. **Convert** in the training set any word that is not in this set (any OOV word) to the unknown word token <UNK> in a text normalization step.
  3. **Estimate** the probabilities for <UNK> from its counts just like any other regular word in the training set.
- in situations where we don't have a prior vocabulary in advance, is to create such a vocabulary implicitly, replacing words in the training data by <UNK> based on their frequency.

Smoothing

What do we do with words that are in our vocabulary (they are not unknown words) but appear in a test set in an unseen context.

To keep a language model from assigning zero probability to these unseen events, we'll have to shave off a bit of probability mass from some more frequent events and give it to the events we've never seen. This modification is called **smoothing** or **discounting**.

Laplace Smoothing (LS) – add one smoothing

- add one to all the n-gram counts, before we normalize them into probabilities. All the counts that used to be zero will now have a count of 1, the counts of 1 will be 2, and so on.
- LS performance is not as good enough to be used in the modern n-gram models.
- wi - word
  ci - count
  N - total number of word tokens
  V – words in vocab

  unsmoothed maximum likelihood estimates of the unigram probability

$$P(w_i) = \frac{c_i}{N}$$

  - Since, each word in the vocab was incremented, ,we need to adjust the denominator to take into account extra V obs.

$$P_{\text{Laplace}}(w_i) = \frac{c_i + 1}{N + V}$$

  - Instead of changing numerator or denominator, it is convenient to describe how a smoothing algorithm affects the numerator, by defining an adjusted count c∗.

$$c_i^* = (c_i + 1)\frac{N}{N + V}$$

    We can now turn c_i^* into a probability P_{i^*} by normalizing by N.
  - A related way to view smoothing is as discounting (lowering) some non-zero counts in order to get the probability mass that will be assigned to the zero counts.
  - Thus, instead of referring to the discounted counts c∗, we might describe a smoothing algorithm in terms of a relative discount dc, the ratio of the discounted counts to the original counts:

$$d_c = \frac{c^*}{c}$$

	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

Figure 3.6 Add-one smoothed bigram counts for eight of the words (out of $V = 1446$) in the Berkeley Restaurant Project corpus of 9332 sentences. Previously-zero counts are in gray.

	i	want	to	eat	chinese	food	lunch	spend
i	0.0015	0.21	0.00025	0.0025	0.00025	0.00025	0.00025	0.00075
want	0.0013	0.00042	0.26	0.00084	0.0029	0.0029	0.0025	0.00084
to	0.00078	0.00026	0.0013	0.18	0.00078	0.00026	0.0018	0.055
eat	0.00046	0.00046	0.0014	0.00046	0.0078	0.0014	0.02	0.00046
chinese	0.0012	0.00062	0.00062	0.00062	0.00062	0.052	0.0012	0.00062
food	0.0063	0.00039	0.0063	0.00039	0.00079	0.002	0.00039	0.00039
lunch	0.0017	0.00056	0.00056	0.00056	0.00056	0.0011	0.00056	0.00056
spend	0.0012	0.00058	0.0012	0.00058	0.00058	0.00058	0.00058	0.00058

Figure 3.7 Add-one smoothed bigram probabilities for eight of the words (out of $V = 1446$) in the BeRP corpus of 9332 sentences. Previously-zero probabilities are in gray.

- It is often convenient to reconstruct the count matrix so we can see how much a smoothing algorithm has changed the original counts, these adjusted counts can be computed by -

$$c^*(w_{n-1}w_n) = \frac{[C(w_{n-1}w_n)+1] \times C(w_{n-1})}{C(w_{n-1})+V} \qquad (3.24)$$

	i	want	to	eat	chinese	food	lunch	spend
i	3.8	527	0.64	6.4	0.64	0.64	0.64	1.9
want	1.2	0.39	238	0.78	2.7	2.7	2.3	0.78
to	1.9	0.63	3.1	430	1.9	0.63	4.4	133
eat	0.34	0.34	1	0.34	5.8	1	15	0.34
chinese	0.2	0.098	0.098	0.098	0.098	8.2	0.2	0.098
food	6.9	0.43	6.9	0.43	0.86	2.2	0.43	0.43
lunch	0.57	0.19	0.19	0.19	0.19	0.38	0.19	0.19
spend	0.32	0.16	0.32	0.16	0.16	0.16	0.16	0.16

Figure 3.8 Add-one reconstituted counts for eight words (of $V = 1446$) in the BeRP corpus of 9332 sentences. Previously-zero counts are in gray.

The sharp change in counts and probabilities occurs because too much probability mass is moved to all the zeros.

Add k-smoothing

One alternative to add-one smoothing is to move a bit less of the probability mass from the seen to the unseen events.
Instead of adding 1 to each count, we add a fractional count k (.5? .05? .01?).

$$P^*_{\text{Add-k}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+k}{C(w_{n-1})+kV}$$

Add-k smoothing requires that we have a method for choosing k; this can be done, for example, by optimizing on a **devset**.

Applications: text classification
Bad with: language modeling, generating counts with poor variances and often inappropriate discounts.

Backoff and Interpolation

The discounting we have been discussing so far can help solve the problem of zero frequency n-grams. But there is an additional source of knowledge we can draw on.

Sometimes using less context is a good thing, helping to generalize more for contexts that the model hasn't learned much about.

There are two ways to use this n-gram "hierarchy":

- In **backoff**, we use the trigram if the evidence is sufficient, otherwise we use the bigram, otherwise the unigram. In other words, we only "back off" to a lower-order n-gram if we have zero evidence for a higher-order n-gram.
- By contrast, in **interpolation**, we always mix the probability estimates from all the n-gram estimators, weighing and combining the trigram, bigram, and unigram counts.

In simple linear interpolation, we combine different order n-grams by linearly interpolating them.

$$\begin{aligned}\hat{P}(w_n|w_{n-2}w_{n-1}) &= \lambda_1 P(w_n) \\ &\quad +\lambda_2 P(w_n|w_{n-1}) \\ &\quad +\lambda_3 P(w_n|w_{n-2}w_{n-1})\end{aligned} \qquad (3.26)$$

here, λs must sum up to 1.

In a slightly more sophisticated version of linear interpolation, each λ weight is computed by conditioning on the context.

If we have particularly accurate counts for a particular bigram, we assume that the counts of the trigrams based on this bigram will be more trustworthy, so we can make the λs for those trigrams higher and thus give that trigram more weight in the interpolation.

$$\begin{aligned}\hat{P}(w_n|w_{n-2}w_{n-1}) &= \lambda_1(w_{n-2:n-1})P(w_n) \\ &\quad +\lambda_2(w_{n-2:n-1})P(w_n|w_{n-1}) \\ &\quad +\lambda_3(w_{n-2:n-1})P(w_n|w_{n-2}w_{n-1})\end{aligned}$$

λs can be set → Both the simple interpolation and conditional interpolation λs are learned from a **held-out corpus**.

A held-out corpus is an additional training corpus that we use to set hyperparameters like these λ values, by choosing the λ values that maximize the likelihood of the held-out corpus.

- So we fix n-gram probabilities and then search for the λ values that—when plugged into Eq. 3.26—give us the highest probability of the held-out set.
- Other ways to calculate the optimal set of λs;
    - EM algorithm
      an iterative learning algorithm that converges on locally optimal λs

In a backoff n-gram model, if the n-gram we need has zero counts, we approximate it by backing off to the (N-1)-gram. We continue backing off until we reach a history that has some counts.

In order for a backoff model to give a correct probability distribution, we have to discount the higher-order n-grams to save some probability mass for the lower order n-grams.
In addition to this explicit discount factor, we'll need a function α to distribute this probability mass to the lower order n-grams.

This kind of backoff with discounting is also called **Katz backoff**.

$$P_{\mathrm{BO}}(w_n|w_{n-N+1:n-1}) = \begin{cases} P^*(w_n|w_{n-N+1:n-1}), & \text{if } C(w_{n-N+1:n}) > 0 \\ \alpha(w_{n-N+1:n-1})P_{\mathrm{BO}}(w_n|w_{n-N+2:n-1}), & \text{otherwise.} \end{cases} \quad (3.29)$$

Katz backoff is often combined with a smoothing method called **Good-Turing**.
The combined Good-Turing backoff algorithm involves quite detailed computation for estimating the Good-Turing smoothing and the P* and α values.

Kneser-Ney Smoothing

One of the best performing n-gram smoothing methods is the interpolated Kneser-Ney algorithm.

Uses concept of– **absolute discounting**.

Consider an n-gram that has count 4. We need to discount this count by some amount. But how much should we discount it? Church and Gale's clever idea was to look at a held-out corpus and just see what the count is for all those bigrams that had count 4 in the training set.

Bigram count in training set	Bigram count in heldout set
0	0.0000270
1	0.448
2	1.25
3	2.24
4	3.23
5	4.21
6	5.23
7	6.21
8	7.21
9	8.26

Figure 3.9 For all bigrams in 22 million words of AP newswire of count 0, 1, 2,...,9, the counts of these bigrams in a held-out corpus also of 22 million words.

Note that except for the held-out counts for 0 and 1, all other bigram counts could be estimated pretty well by subtracting 0.75 from the count in the training set!

Absolute discounting formalizes this intuition by subtracting the fixed absolute discount d from each count. since we have good estimates already for the very high counts, a small discount d won't affect them much. It will mainly modify the smaller counts, for which we don't necessarily trust the estimate anyway.

$$P_{\text{AbsoluteDiscounting}}(w_i|w_{i-1}) = \frac{C(w_{i-1}w_i) - d}{\sum_v C(w_{i-1}\,v)} + \lambda(w_{i-1})P(w_i)$$

The first term is the discounted bigram, and the second term is the unigram with an interpolation weight λ.

- instead of P(w), which answers the question "How likely is w?", we'd like to create a unigram model that we might call PCONTINUATION, which answers the question "How likely is w to appear as a novel continuation?"
- Every bigram type was a novel continuation the first time it was seen. The number of times a word w appears as a novel continuation can be expressed as:

$$P_{\text{CONTINUATION}}(w) \propto |\{v : C(vw) > 0\}|$$

Turning this count to probability now –

$$P_{\text{CONTINUATION}}(w) = \frac{|\{v : C(vw) > 0\}|}{|\{(u', w') : C(u'w') > 0\}|}$$

Huge Language Models and Stupid Backoff

COCA is a balanced corpora, meaning that it has roughly equal numbers of words from different genres: web, newspapers, spoken conversation transcripts, fiction, and so on, drawn from the period 1990-2019.

Efficiency

- instead of storing each word as string – 64-bit hash number, with the words themselves stored on disk.
- Probabilities are generally quantized using only 4-8 bits (instead of 8-byte floats)
- n-grams are stored in reverse tries

- Pruning can be used to shrink the n-gram LM, based on counts greater than threshold, or using entropy to prune the less important n-grams; build approximate language models using techniques like **Bloom filters.**
- efficient language model toolkits like **KenLM** use sorted arrays, efficiently combine probabilities and backoffs in a single value, and use merge sorts to efficiently build the probability tables in a minimal number of passes through a large corpus.

- Above toolkits show that with very large language models a much simpler algorithm may be sufficient. The algorithm is called **stupid backoff**. Stupid backoff gives up the idea of trying to make the language model a true probability distribution. There is no discounting of the higher-order probabilities. If a higher-order n-gram has a zero count, we simply backoff to a lower order n-gram, weighed by a fixed (context-independent) weight. This algorithm does not produce a probability distribution as S:

$$S(w_i|w_{i-k+1:i-1}) = \begin{cases} \dfrac{\text{count}(w_{i-k+1:i})}{\text{count}(w_{i-k+1:i-1})} & \text{if count}(w_{i-k+1:i}) > 0 \\ \lambda S(w_i|w_{i-k+2:i-1}) & \text{otherwise} \end{cases} \quad (3.40)$$

The backoff terminates in the unigram, which has probability $S(w) = \frac{count(w)}{N}$. Brants et al. (2007) find that a value of 0.4 worked well for λ.

Advanced: Perplexity's Relation to Entropy

Entropy is a measure of information. Given a random variable X ranging over whatever we are predicting (words, letters, parts of speech, the set of which we'll call χ) and with a particular probability function, call it p(x), the entropy of the random variable X is

$$H(X) = -\sum_{x \in \chi} p(x) \log_2 p(x) \quad (3.41)$$

To save info about 8 numbers, we can use 3 bits of space. A better way is using the entropy.

Suppose that the spread is the actual distribution of the bets placed and that we represent it as the prior probability of each horse as follows:

Horse 1	$\frac{1}{2}$	Horse 5	$\frac{1}{64}$
Horse 2	$\frac{1}{4}$	Horse 6	$\frac{1}{64}$
Horse 3	$\frac{1}{8}$	Horse 7	$\frac{1}{64}$
Horse 4	$\frac{1}{16}$	Horse 8	$\frac{1}{64}$

The entropy of the random variable X that ranges over horses gives us a lower bound on the number of bits and is:

$$\begin{aligned} H(X) &= -\sum_{i=1}^{i=8} p(i) \log p(i) \\ &= -\tfrac{1}{2}\log\tfrac{1}{2} - \tfrac{1}{4}\log\tfrac{1}{4} - \tfrac{1}{8}\log\tfrac{1}{8} - \tfrac{1}{16}\log\tfrac{1}{16} - 4(\tfrac{1}{64}\log\tfrac{1}{64}) \\ &= 2 \text{ bits} \end{aligned}$$

A code that averages 2 bits per race can be built with short encodings for more probable horses, and longer encodings for less probable horses.

For example, we could encode the most likely horse with the code 0, and the remaining horses as 10, then 110, 1110, 111100, 111101, 111110, and 111111.

Concept of entropy in sequences.

computing the entropy of some sequence ofwordsW= {w1,w2, . . . ,wn}.

We could define the entropy rate (we could also think of this as the per-wordentropy) as the entropy of this sequence divided by the number of words:

$$\frac{1}{n}H(w_{1:n}) = -\frac{1}{n}\sum_{w_{1:n} \in L} p(w_{1:n}) \log p(w_{1:n})$$

But to measure the true entropy of a language, we need to consider sequences of infinite length. If we think of a language as a stochastic process L that produces a sequence of words, and allow W to represent the sequence of words w1, . . . ,wn, then L's entropy rate H(L) is defined as:

$$H(L) = \lim_{n\to\infty} \frac{1}{n} H(w_1, w_2, \ldots, w_n)$$
$$= -\lim_{n\to\infty} \frac{1}{n} \sum_{W\in L} p(w_1, \ldots, w_n) \log p(w_1, \ldots, w_n) \quad (3.46)$$

if the language is regular in certain ways (to be exact, if it is both stationary and ergodic),

$$H(L) = \lim_{n\to\infty} -\frac{1}{n} \log p(w_1 w_2 \ldots w_n)$$

That is, we can take a single sequence that is long enough instead of summing over all possible sequences. The intuition of the Shannon-McMillan-Breiman theorem is that a long-enough sequence of words will contain in it many other shorter sequences and that each of these shorter sequences will reoccur in the longer sequence according to their probabilities.

A stochastic process is said to be **stationary** if the probabilities it assigns to a sequence are invariant with respect to shifts in the time index. In other words, the probability distribution for words at time t is the same as the probability distribution at time t +1.

Markov models, and hence n-grams, are stationary. For example, in a bigram, Pi is dependent only on Pi−1. So if we shift our time index by x, Pi+x is still dependent on Pi+x−1.

The **cross-entropy** is useful when we don't know the actual probability distribution p that generated some data. It allows us to use some m, which is a model of p (i.e., an approximation to p). The cross-entropy of m on p is defined by-

$$H(p,m) = \lim_{n\to\infty} -\frac{1}{n} \sum_{W\in L} p(w_1, \ldots, w_n) \log m(w_1, \ldots, w_n) \quad (3.48)$$

$$H(p,m) = \lim_{n\to\infty} -\frac{1}{n} \log m(w_1 w_2 \ldots w_n) \quad (3.49)$$

for entropy, we can estimate the cross-entropy of a model m on some distribution p by taking a single sequence that is long enough instead of summing over all possible sequences.

What makes the cross-entropy useful is that the cross-entropy H(p,m) is an upper bound on the entropy H(p).

$$H(p) \leq H(p,m) \quad (3.50)$$

The more accurate the model m will be, the cross entropy H(p,m) will be closer to the true entropy H(p). Difference between these denotes the accuracy of model.

Lower cross entropy → more accurate the model.

Naïve Bayes and Sentiment Classification

Here we will apply naïve Bayes algo to **text categorization**, the task of assigning a label or sentiment analysis category to an entire text or document.

We focus on one common text categorization task, **sentiment analysis**, the extraction of sentiment.

Spam detection is another important commercial application, the binary classification task of assigning an email to one of the two classes spam or not-spam.

In the **supervised** situation we have a training set of N documents that have each been hand-labeled with a class: (d1,c1), ..., (dN,cN). Our goal is to learn a classifier that is capable of mapping from a new document d to its correct class c ∈C. A **probabilistic classifier** additionally will tell us the probability of the observation being in the class.

Generative classifiers like naive Bayes build a model of how a class could generate some input data.

Discriminative classifiers like LR instead learn what features from the input are most useful to discriminate between the different possible classes.

While discriminative systems are often more accurate and hence more commonly used,

Naïve Bayes Classifiers

multinomial naive Bayes classifier, so called because it is a Bayesian classifier that makes a simplifying (naive) assumption about how the features interact.

bag-of-words is an unordered set of words with their position ignored, keeping only their frequency in the document.

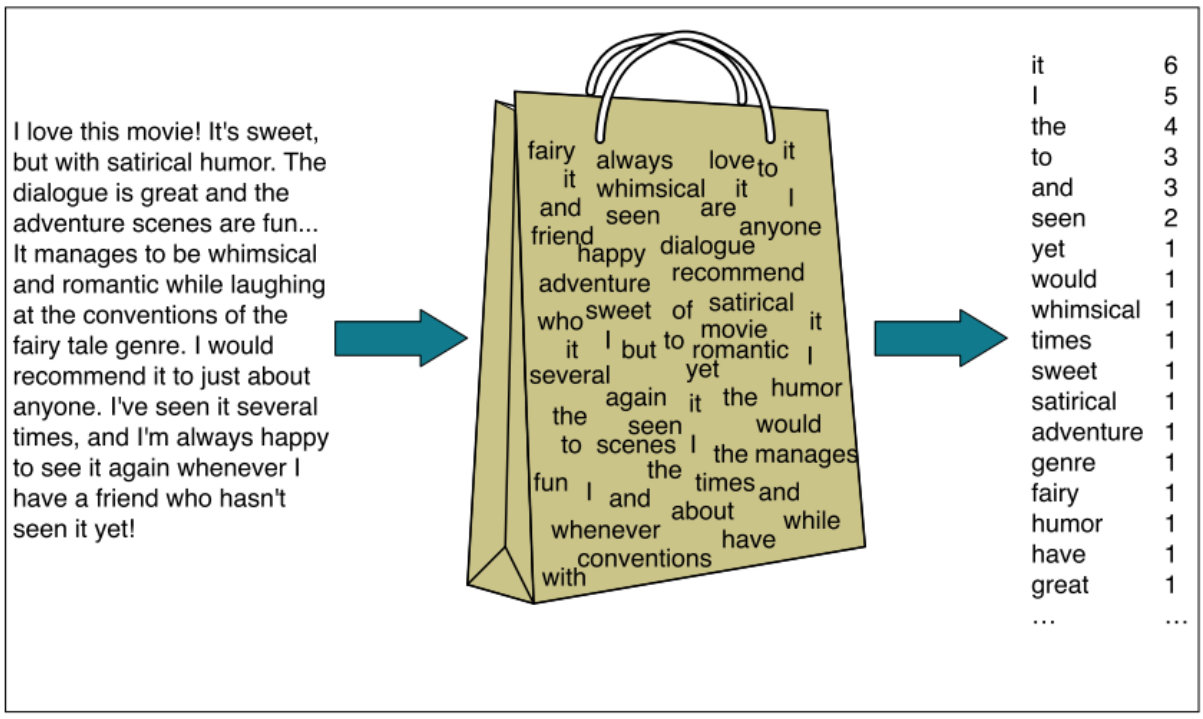


Figure 4.1 Intuition of the multinomial naive Bayes classifier applied to a movie review. The position of the words is ignored (the *bag of words* assumption) and we make use of the frequency of each word.

Naive Bayes is a probabilistic classifier, meaning that for a document d, out of all classes c ∈C the classifier returns the class ˆc which has the maximum posterior probability given the document. In Eq. 4.1 we use the hat notation ˆ to mean "our estimate of the correct class".

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} P(c|d) \qquad (4.1)$$

Substituting the Bayes' rule we get:

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} P(c|d) = \underset{c \in C}{\operatorname{argmax}} \frac{P(d|c)P(c)}{P(d)} \qquad (4.3)$$

In this eq 4.3, since the doc remains same, the probability P(d) remains same, so we can maximise a simpler formula:

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} P(c|d) = \underset{c \in C}{\operatorname{argmax}} P(d|c)P(c) \qquad (4.4)$$

For classification, we compute ^c given doc d by choosing the class with has the highest product of two probabilities:

- the **prior** probability of the class P(c)
- the **likelihood** of the document P(d|c) → representing doc d as set of features f1, f2, f3..fn.

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \overbrace{P(d|c)}^{\text{likelihood}} \overbrace{P(c)}^{\text{prior}} \quad \rightarrow \quad \hat{c} = \underset{c \in C}{\operatorname{argmax}} \overbrace{P(f_1, f_2, \ldots, f_n|c)}^{\text{likelihood}} \overbrace{P(c)}^{\text{prior}}$$

Simplifying the latter eq. as calculating possible combination of features is intensive task, we use two ways:

1. bag of words assumption: position doesn't matter.
2. **naive Bayes assumption**: conditional independence assumption that the probabilities P(fi|c) are independent given the class c.

$$P(f_1, f_2, \ldots, f_n|c) = P(f_1|c) \cdot P(f_2|c) \cdot \ldots \cdot P(f_n|c) \qquad (4.7)$$

The final equation for the class chosen by a naive Bayes classifier is thus:

$$c_{NB} = \underset{c \in C}{\operatorname{argmax}} P(c) \prod_{f \in F} P(f|c) \qquad (4.8)$$

To apply naïve Bayes classifier to text, we consider the word positions, by indexing the word position in document.

$$\text{positions} \leftarrow \text{all word positions in test document}$$

$$c_{NB} = \underset{c \in C}{\operatorname{argmax}} P(c) \prod_{i \in positions} P(w_i|c) \qquad (4.9)$$

$$c_{NB} = \underset{c \in C}{\operatorname{argmax}} \log P(c) + \sum_{i \in positions} \log P(w_i|c) \qquad (4.10)$$

Eq. 4.10 computes the predicted class as a linear function of input features. Classifiers that use a linear combination of the inputs to make a classification decision —like naive Bayes and also LR— are called **linear classifiers**.

Training the Naive Bayes Classifier

How can we learn the probabilities P(c) and P(fi|c)?

For the class prior P(c) we ask what % of the documents in our training set are in each class c.

Nc - number of documents in our training data with class c

Ndoc - total number of documents. Then:

$$\hat{P}(c) = \frac{N_c}{N_{doc}} \qquad (4.11)$$

P(wi|c), which we compute as the fraction of times the word wi appears among all words in all documents of topic c. We first concatenate all documents with category c into one big "category c" text. Then we use the frequency of wi in this concatenated document to give a maximum likelihood estimate of the probability:

$$\hat{P}(w_i|c) = \frac{count(w_i, c)}{\sum_{w \in V} count(w, c)} \qquad (4.12)$$

V consists of the union of all the word types in all classes, not just the words in one class c.

Issues here: What happens if there is no word "finatastic" for a given class +ve in the training set. Instead that word appears in another class -ve. In such case, the probability of this feature becomes 0.

$$\hat{P}(\text{"fantastic"}|\text{positive}) = \frac{count(\text{"fantastic"}, \text{positive})}{\sum_{w \in V} count(w, \text{positive})} = 0 \qquad (4.13)$$

since naive Bayes naively multiplies all the feature likelihoods together, zero probabilities in the likelihood term for any class will cause the probability of the class to be zero, no matter the other evidence!

Solution: add-one smoothening (Laplace)

$$\hat{P}(w_i|c) = \frac{count(w_i, c) + 1}{\sum_{w \in V} (count(w, c) + 1)} = \frac{count(w_i, c) + 1}{\left(\sum_{w \in V} count(w, c)\right) + |V|} \qquad (4.14)$$

Unknown words: Words that occur in test data but not in training data. How to deal with them – ignore them.

Stop words – the, a, etc. Class of words to be completely ignored off.

Example

	Cat	Documents
Training	-	just plain boring
	-	entirely predictable and lacks energy
	-	no surprises and very few laughs
	+	very powerful
	+	the most fun film of the summer
Test	?	predictable with no fun

The prior $P(c)$ for the two classes is computed via Eq. 4.11 as $\frac{N_c}{N_{doc}}$:

$$P(-) = \frac{3}{5} \qquad P(+) = \frac{2}{5}$$

- with – doesn't appear in train → drop it.
- The likelihoods from the training set for the remaining three words "predictable", "no", and "fun", are as follows, from Eq. 4.14

$$P(\text{"predictable"}|-) = \frac{1+1}{14+20} \qquad P(\text{"predictable"}|+) = \frac{0+1}{9+20}$$

$$P(\text{"no"}|-) = \frac{1+1}{14+20} \qquad P(\text{"no"}|+) = \frac{0+1}{9+20}$$

$$P(\text{"fun"}|-) = \frac{0+1}{14+20} \qquad P(\text{"fun"}|+) = \frac{1+1}{9+20}$$

For the test sentence S = "predictable with no fun", after removing the word 'with', the chosen class, via Eq. 4.9, is therefore computed as follows:

$$P(-)P(S|-) = \frac{3}{5} \times \frac{2 \times 2 \times 1}{34^3} = 6.1 \times 10^{-5}$$

$$P(+)P(S|+) = \frac{2}{5} \times \frac{1 \times 1 \times 2}{29^3} = 3.2 \times 10^{-5}$$

- So the model gives negative class to the test sentence.

Optimizing for Sentiment Analysis

Standard naïve Bayes text classification works well for SA, but lets improve performance further.

For sentiment classification and other text classification tasks, seems to matter more than its frequency. Thus, it often improves performance to clip the word counts in each document. This variant is called **Binary Multinomial Naïve Bayes**.

The example is worked without add-1 smoothing to make the differences clearer. Note that the results counts need not be 1; the word great has a count of 2 even for Binary NB, because it appears in multiple documents.

Four original documents:

- − it was pathetic the worst part was the boxing scenes
- − no plot twists or great scenes
- + and satire and great plot twists
- + great scenes great film

After per-document binarization:

- − it was pathetic the worst part boxing scenes
- − no plot twists or great scenes
- + and satire great plot twists
- + great scenes film

	NB Counts		Binary Counts	
	+	−	+	−
and	2	0	1	0
boxing	0	1	0	1
film	1	0	1	0
great	3	1	2	1
it	0	1	0	1
no	0	1	0	1
or	0	1	0	1
part	0	1	0	1
pathetic	0	1	0	1
plot	1	1	1	1
satire	1	0	1	0
scenes	1	2	1	2
the	0	2	0	1
twists	1	1	1	1
was	0	2	0	1
worst	0	1	0	1

Figure 4.3 An example of binarization for the binary naive Bayes algorithm.

2nd addition - dealing with negation.

I really like this movie (positive) and *I didn't like this movie* (negative)

So, a positive word can have negative meaning in different context.

Solution - during text normalization, prepend the prefix NOT to every word after a token of logical negation (n't, not, no, never) until the next punctuation mark.

```
didn't like this movie , but I
```

becomes

```
didn't NOT_like NOT_this NOT_movie , but I
```

3rd situation - insufficient labelled training data to train accurate naive Bayes classifiers using all words in the training set to estimate positive and negative sentiment.

Solution – using sentiment lexicons to derive +ve and -ve word features.

Popular lexicons → General Inquirer, LIWC, The opinion lexicon of Hu and Liu & MPQA

Using lexicons → we might add a feature called 'this word occurs in the positive lexicon', and treat all instances of words in the lexicon as counts for that one feature, instead of counting each word separately. Similarly, we might add as a second feature 'this word occurs in the negative lexicon' of words in the negative lexicon.

If we have lots of training data, and if the test data matches the training data, using just two features won't work as well as using all the words. But when training data is sparse or not representative of the test set, using dense lexicon features instead of sparse individual-word features may generalize better.

Naive Bayes for other text classification tasks

our classifier use all the words in the training data as features.

spam detection solution 1998- rather than using all the words as individual features, is to predefine likely sets of words or phrases as features, combined with features that are not purely linguistic

Naive Bayes as a Language Model

Naive Bayes classifiers can use any sort of feature: dictionaries, URLs, email addresses, network features, phrases, and so on.

Specifically, a naive Bayes model can be viewed as a set of class-specific unigram language models, in which the model for each class instantiates a unigram language model.

Thus consider a naive Bayes model with the classes *positive* (+) and *negative* (-) and the following model parameters:

w	P(w|+)	P(w|-)
I	0.1	0.2
love	0.1	0.001
this	0.01	0.01
fun	0.05	0.005
film	0.1	0.1
...	...	...

$$P(\text{“I love this fun film”}|+) = 0.1 \times 0.1 \times 0.01 \times 0.05 \times 0.1 = 0.0000005$$
$$P(\text{“I love this fun film”}|-) = 0.2 \times 0.001 \times 0.01 \times 0.005 \times 0.1 = .0000000010$$

Means, $P(s|pos) > P(s|neg)$

Note that this is just the likelihood part of the naive Bayes model; once we multiply in the prior a full naive Bayes model might well make a different classification decision

Evaluation of classifiers: Precision, Recall, F-measure

Human labels to documents – **gold labels**

A **confusion matrix** is a table for visualizing how an algorithm performs with respect to the human gold labels, using two dimensions (system output and gold labels), and each cell labeling a set of possible outcomes.

In spam detection, TP – documents are spam. FN – incorrectly labelled as non spam by system.

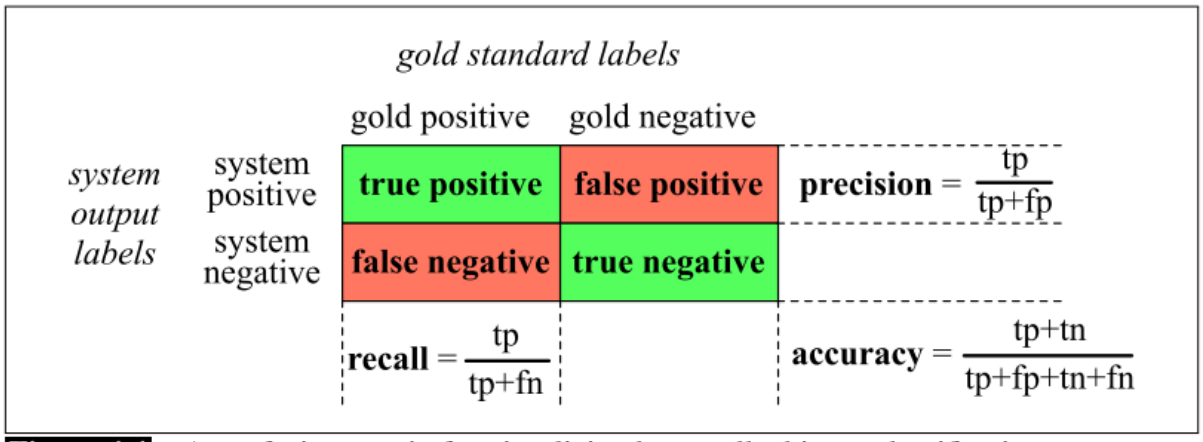


Figure 4.4 A confusion matrix for visualizing how well a binary classification system performs against gold standard labels.

We don't use accuracy for text classification tasks, as it doesn't work when the classes are unbalanced. Imagine a simple classifier that stupidly classified every tweet as "not about pie". This classifier would have 999,900 true negatives and only 100 false negatives for an accuracy of 999,900/1,000,000 or 99.99%! But this wont work at all. Despite having a fabulous accuracy of 99.99%, has a terrible recall of 0 (since there are no true positives, and 100 false negatives, the recall is 0/100)

In other words, accuracy is not a good metric when the goal is to discover something that is rare, or at least not completely balanced in frequency, which is a very common situation in the world.

So we consider **precision** and **recall** instead of accuracy.

$$\textbf{Precision} = \frac{\text{true positives}}{\text{true positives + false positives}} \quad \textbf{Recall} = \frac{\text{true positives}}{\text{true positives + false negatives}}$$

Precision measures the percentage of the items that the system detected (i.e., the system labeled as positive) that are in fact positive (i.e., are positive according to the human gold labels)

Recall measures the percentage of items actually present in the input that were correctly identified by the system.

F – measure : incorporates Accuracy and Precision.

$$F_\beta = \frac{(\beta^2+1)PR}{\beta^2 P + R}$$

The β parameter differentially weights the importance of recall and precision based perhaps on the needs of an application.

When β = 1, precision and recall are equally balanced; F1 score.

Values of β > 1 favour recall,

values of β < 1 favour precision.

To calculate the F measure, we use Harmonic mean, because HM of two values is closer to the minimum of the two values than the arithmetic mean is. Thus it weighs the lower of the two numbers more heavily.

Evaluating with more than two classes

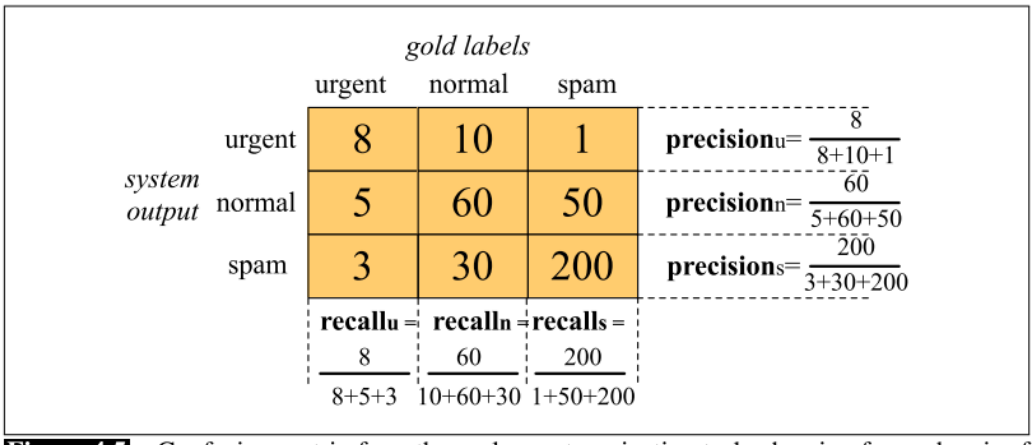


Figure 4.5 Confusion matrix for a three-class categorization task, showing for each pair of classes (c_1, c_2), how many documents from c_1 were (in)correctly assigned to c_2

In **macroaveraging**, we compute the performance for each class, and then average over classes.

In **microaveraging**, we collect the decisions for all classes into a single confusion matrix, and then compute precision and recall from that table.

Fig. 4.6 shows the CM for each class separately, and shows the computation of microaveraged and macroaveraged precision. As the figure shows, a microaverage is dominated by the more frequent class (spam), since the counts are pooled.

The macroaverage better reflects the statistics of the smaller classes, and so is more appropriate when performance on all the classes is equally important.

Test sets and Cross-validation

Classifiers are trained using distinct training, dev, and test sets, including the use of cross-validation in the training set. Once we come up with what we think is the best model, we run it on the (hitherto unseen) test set to report its performance.

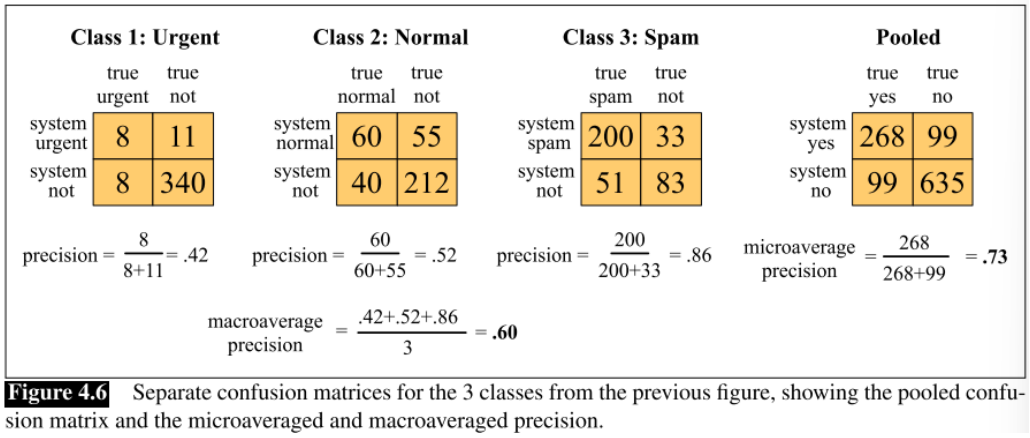


Figure 4.6 Separate confusion matrices for the 3 classes from the previous figure, showing the pooled confusion matrix and the microaveraged and macroaveraged precision.

Devset – avoid the overfitting

Having fixed training set, devset & test set – the data is trimmed a lot. Causes data representation issues. Here **Cross validation** can be helpful.

In cross-validation, we choose a number k, and partition our data into k disjoint subsets called folds. Now we choose one of those k folds as a test set, train our classifier on the remaining k−1 folds, and then compute the error rate on the test set.

E.g., 10-fold CV: train 10 different models (each on 90% of our data), test the model 10 times, and average these 10 values.

Problem with CV: because all the data is used for testing, we need the whole corpus to be blind; we can't examine any of the data to suggest possible features and in general see what's going on, because we'd be peeking at the test set, and such cheating would cause us to overestimate the performance of our system.

But it is important to look at the corpora to see what's going on, So – we create fixed train set & fixed test set and then do 10-fold CV inside training set. But compute the error rate normally on the test set. Fig 4.7

Statistical Significance Testing

Comparing the systems in terms of performance.

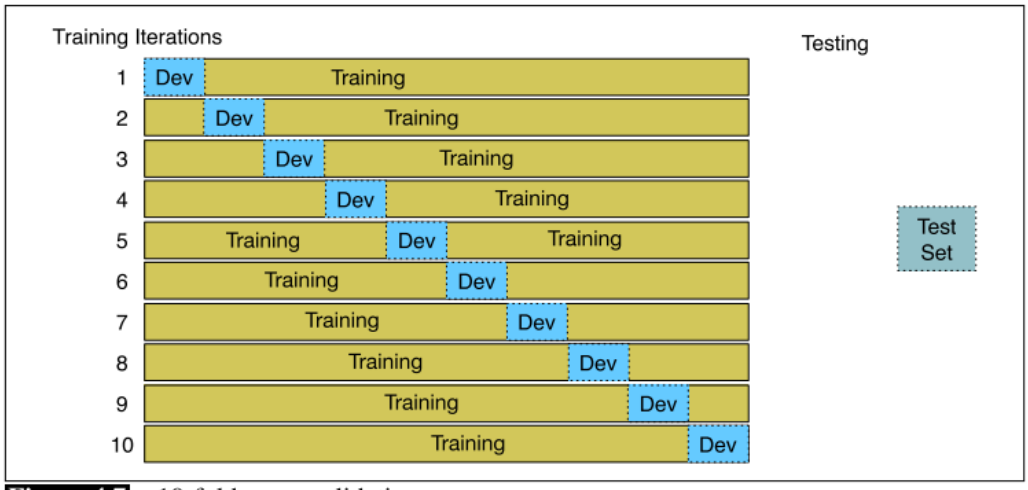


Figure 4.7 10-fold cross-validation

Suppose we are comparing the performance of classifier A & B on metric M.

Perhaps we want to know if our LR sentiment classifier A (Chapter 5) gets a higher F1 score than our naive Bayes sentiment classifier B on a particular test set x. Let's call M(A,x) the score that system A gets on test set x, and δ(x) the performance difference between A and B on x:

$$\delta(x) = M(A,x) - M(B,x) \qquad (4.19)$$

δ(x) is called the **effect size**. Big δ(x) means A is better, if small then B is better.

So just checking δ(x) is not sufficient, because just brcause it is +ve, A might just be accidentally better than B on this particular x. So, we want to know if A's superiority over B is likely to hold again if we checked another test set x', or under some other set of circumstances.

$$H_0 : \delta(x) \leq 0$$
$$H_1 : \delta(x) > 0 \qquad (4.20)$$

H_0, called the **null hypothesis**, supposes that δ(x) is actually negative or zero, meaning that A is not better than B.

creating a random variable X ranging over all test sets. How likely is it, if the null hypothesis H_0 was correct, that among these test sets we would encounter the value of δ(x) that we found. We formalize this likelihood as the **p-value**: the probability, assuming the null hypothesis H0 is true, of seeing the δ(x) that we saw or one even greater

$$P(\delta(X) \geq \delta(x) | H_0 \text{ is true}) \qquad (4.21)$$

A very small p-value (threshold 0.05 or 0.01) means that the difference we observed is very unlikely under the null hypothesis, and we can reject the null hypothesis.

We say that a result (e.g., "A is better than B") is **statistically significant** if the δ we saw has a probability that is below the threshold, and we therefore reject this null hypothesis.

How do we compute this probability we need for the p-value? In NLP we generally don't use simple parametric tests like t-tests or ANOVAs that you might be familiar with.

There are two common non-parametric tests used in NLP: **approximate randomization** (Noreen, 1989) and the **bootstrap test**.

Paired tests are those in which we compare two sets of observations that are aligned: each observation in one set can be paired with an observation in another.ie when comparing the performance of two systems on the same test set; we can pair the performance of system A on an individual observation xi with the performance of system B on the same xi.

The Paired Bootstrap Test

can apply to any metric; from precision, recall, or F1 to the BLEU metric used in machine translation.

bootstrapping refers to repeatedly drawing large numbers of smaller samples with replacement (called bootstrap samples) from an original larger sample.

Consider a set x of 10 documents. Two classifiers A and B on the test set. On first doc AB (Green is right. Red is false) & on second doc AB

Suppose, A has an accuracy of .70 and B of .50, so δ(x) is .20.

Now we create a large number b (perhaps 105) of virtual test sets x(i), each of size n = 10.

	1	2	3	4	5	6	7	8	9	10	A%	B%	$\delta()$
x	AB	A~~B~~	AB	~~A~~B	A~~B~~	~~A~~B	A~~B~~	AB	~~A~~~~B~~	A~~B~~	.70	.50	.20
$x^{(1)}$	A~~B~~	AB	A~~B~~	~~A~~B	~~A~~B	A~~B~~	~~A~~B	AB	~~A~~~~B~~	AB	.60	.60	.00
$x^{(2)}$	A~~B~~	AB	~~A~~~~B~~	~~A~~B	~~A~~B	AB	~~A~~B	A~~B~~	AB	AB	.60	.70	-.10
...													
$x^{(b)}$													

Figure 4.8 The paired bootstrap test: Examples of b pseudo test sets $x^{(i)}$ being created from an initial true test set x. Each pseudo test set is created by sampling $n = 10$ times with replacement; thus an individual sample is a single cell, a document with its gold label and the correct or incorrect performance of classifiers A and B. Of course real test sets don't have only 10 examples, and b needs to be large as well.

In the b test sets we want to know how often A has unfair advantage. How to calculate?

Berg-Kirkpatrick et al. -- Assuming H_0 (A isn't better than B), we would expect that δ(X), estimated over many test sets, would be zero;

a higher value would be surprising, since H0 specifically assumes A isn't better than B.

To measure exactly how surprising is our observed δ(x) we would in other circumstances compute the p-value by counting over many test sets how often δ(x(i)) exceeds the expected zero value by δ(x) or more:

$$\text{p-value}(x) = \frac{1}{b}\sum_{i=1}^{b} \mathbb{1}\left(\delta(x^{(i)}) - \delta(x) \geq 0\right)$$

$\mathbb{1}(x)$ mean "1 if x is true, and 0 otherwise"

However, although it's generally true that the expected value of δ(X) over many test sets, (again assuming A isn't better than B) is 0, this isn't true for the bootstrapped test sets we created.

That's because we didn't draw these samples from a distribution with 0 mean; we created them from the original test set x, which is biased (by .20) in favor of A. So to measure how surprising is our observed δ(x), we actually compute the p-value by counting over many test sets how often δ(x(i)) exceeds the expected value of δ(x) by δ(x) or more:

$$\begin{aligned}\text{p-value}(x) &= \frac{1}{b}\sum_{i=1}^{b} \mathbb{1}\left(\delta(x^{(i)}) - \delta(x) \geq \delta(x)\right) \\ &= \frac{1}{b}\sum_{i=1}^{b} \mathbb{1}\left(\delta(x^{(i)}) \geq 2\delta(x)\right) \qquad (4.22)\end{aligned}$$

So, for ex. 10000 test set, threshold of 0.01, & in only 47 of the test sets $x^{(i)}$ we find that $\delta(x^{(i)}) \geq 2\delta(x)$, the resulting p-value of .0047 is smaller than .01, indicating δ(x) is indeed sufficiently surprising, and we can reject the null hypothesis and conclude A is better than B.

Avoiding harms in classification

Representational harms - caused by a system that demeans a social group.

example Kiritchenko and Mohammad (2018) examined the performance of 200 sentiment analysis systems on pair of sentences containing African and European American names. Findings - most systems assigned lower sentiment and more negative emotion to sentences with African American names, reflecting and perpetuating stereotypes that associate African Americans with negative emotions.

the important text classification task of **toxicity detection** is the task of detecting hate speech, abuse, harassment, or other kinds of toxic language.

These problems are caused by the biases in the training data.

Model cards – contains the information about the training algo, parameters, data sources, motivation and pre-processing, intended use, model performance across multiple domains.

LR

This chapter introduced the **logistic regression** model of **classification**.

- Logistic regression is a supervised machine learning classifier that extracts real-valued features from the input, multiplies each by a weight, sums them, and passes the sum through a **sigmoid** function to generate a probability. A threshold is used to make a decision.
- Logistic regression can be used with two classes (e.g., positive and negative sentiment) or with multiple classes (**multinomial logistic regression**, for example for n-ary text classification, part-of-speech labeling, etc.).
- Multinomial logistic regression uses the **softmax** function to compute probabilities.
- The weights (vector w and bias b) are learned from a labeled training set via a loss function, such as the **cross-entropy loss**, that must be minimized.
- Minimizing this loss function is a **convex optimization** problem, and iterative algorithms like **gradient descent** are used to find the optimal weights.
- **Regularization** is used to avoid overfitting.
- Logistic regression is also one of the most useful analytic tools, because of its ability to transparently study the importance of individual features.

In NLP, LR is the baseline supervised ML algorithm for classification, and has a very close relationship with NN.

Multinomial LR: used for more than two classes.

Generative and Discriminative Classifiers

Difference Naïve Bayes and LR – LR is discriminative while NB is generative classifier.

While differentiating the dogs and cats, the generative models will observe many features while discriminative models will only observe certain patterns like collars. So when asked the models will only tell that cats don't wear collars. While the generative models can provide more details.

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \overbrace{P(d|c)}^{\text{likelihood}} \overbrace{P(c)}^{\text{prior}} \qquad (5.1)$$

A **generative model** like naive Bayes makes use of this **likelihood** term, which expresses how to generate the features of a document *if we knew it was of class c*.

discriminative model in this text categorization scenario attempts to directly compute P(c|d). Perhaps it will learn to assign a high weight to document features that directly improve its ability to *discriminate* between possible classes, even if it couldn't generate an example of one of the classes.

Components of a probabilistic machine learning classifier:

- A **feature representation** of the input.
- A classification function that computes ˆy, the estimated class, via p(y|x). [**sigmoid** and **softmax** tools for classification.]
- Objective Function for learning, [**cross-entropy loss function**].
- A f() for optimizing the Objective Function. [stochastic GD]

Logistic regression has two phases:

training: we train the system (specifically the weights w and b) using stochastic gradient descent and the cross-entropy loss.

test: Given a test example x we compute $p(y|x)$ and return the higher probability label $y = 1$ or $y = 0$.

Classification: the sigmoid

When given the task to perform binary decision between +ve and -ve sentiment, LR solves this task by learning, from a training set, a vector of weights and a bias term.

$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b \qquad (5.2)$$

$$z = w \cdot x + b \qquad (5.3)$$

Each weight w_i is a real number, and is associated with one of the input features xi.

The weight w_i represents how important that input feature is to the classification decision, can be +ve or -ve.

The bias term, also called the intercept, real number that's added to the weighted inputs.

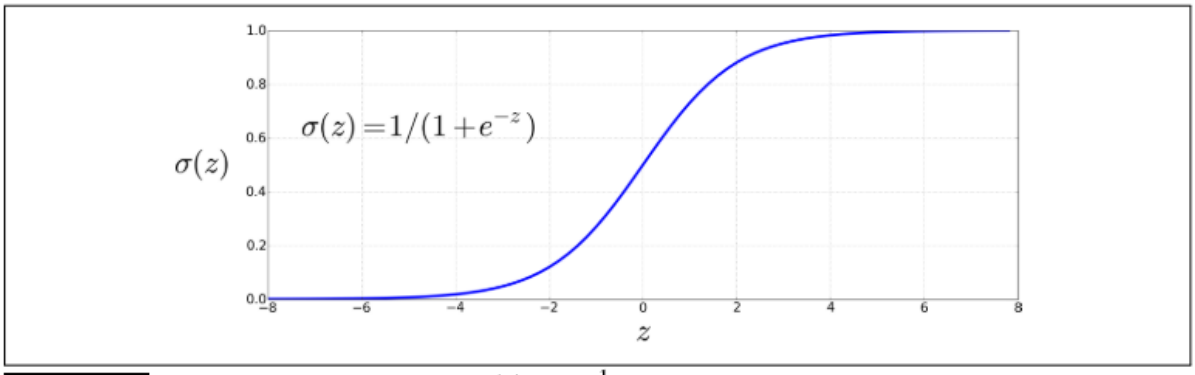


Figure 5.1 The sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ takes a real value and maps it to the range $[0, 1]$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

Since nothing forces z to be a legal probability as it has range -infi to +infi, we pass it through sigmoid function. **Sigmoid f()** also called as **logistic function**.

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+\exp(-z)} \tag{5.4}$$

The threshold acts as a decision boundary, here DB - 0.5

$$\text{decision}(x) = \begin{cases} 1 & \text{if } P(y=1|x) > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

Var	Definition	Value in Fig. 5.2
x_1	count(positive lexicon words $\in$ doc)	3
x_2	count(negative lexicon words $\in$ doc)	2
x_3	$\begin{cases} 1 & \text{if "no"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	1
x_4	count(1st and 2nd pronouns $\in$ doc)	3
x_5	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	0
x_6	log(word count of doc)	$\ln(66) = 4.19$

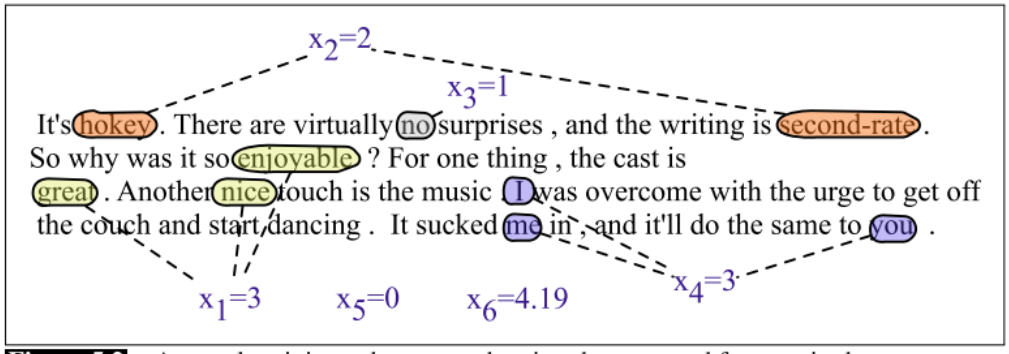


Figure 5.2 A sample mini test document showing the extracted features in the vector x.

$$
\begin{aligned}
p(+|x) = P(y=1|x) &= \sigma(w \cdot x + b) \\
&= \sigma([2.5, -5.0, -1.2, 0.5, 2.0, 0.7] \cdot [3, 2, 1, 3, 0, 4.19] + 0.1) \\
&= \sigma(.833) \\
&= 0.70 \qquad (5.7) \\
p(-|x) = P(y=0|x) &= 1 - \sigma(w \cdot x + b) \\
&= 0.30
\end{aligned}
$$

period disambiguation: deciding if a period is the end of a sentence or part of a word, by classifying each period into one of two classes EOS (end-of-sentence) and not-EOS(eg. Prof. / St. for street).

Designing features:

- We saw such a feature for period disambiguation above, where a period on the word St. was less likely to be the end of the sentence if the previous word was capitalized.
- For LR and NB these combination features or feature interactions have to be designed by hand.
- On large scale, these are created automatically via **feature templates**, abstract specifications of features.
- Considering the n-grams the features should be contained in the n-gram range. So feature space is sparse. Generally the features are created as a hash from the string descriptions.
    - A user description of a feature as, “bigram(American breakfast)” is hashed into a unique integer i that becomes the feature number fi.
- to avoid the extensive human effort of feature design, recent research in NLP has focused on **representation learning**: ways to learn features automatically in an unsupervised way from the input.

Choosing a classifier:

- Advantage of LR over NB
    - NB has overly strong conditional independence assumptions. Ie if two features are strongly correlated, imagine that we just add the same feature f1 twice. Naive Bayes will treat both copies of f1 as if they were separate, multiplying them both in, overestimating the evidence. NB works well on smaller datasets.
    - LR is much more robust to correlated features; if two features f1 and f2 are perfectly correlated, regression will simply assign part of the weight to w1 and part to w2. Thus, when there are many correlated features, LR will assign a more accurate probability than NB. So LR generally works better on larger documents or datasets and is a common default.

Learning in LR

Two components:

- A metric for how close the current label (ˆy) is to the true gold label y. [**Loss** or **cost funciton**] eg. **cross-entropy loss**.
- An optimization algorithm for iteratively updating the weights so as to minimize this loss function. **GD/ stochastic GD**.

The Cross-entropy Loss Function

$$L(\hat{y}, y) = \text{How much } \hat{y} \text{ differs from the true } y \qquad (5.8)$$

loss function that prefers the correct class labels of the training examples to be *more likely*. This is called **conditional maximum likelihood estimation**: we choose the parameters *w,b* that **maximize the log probability of the true *y* labels in the training data** given the observations x. The resulting loss function is the negative log likelihood loss, generally called the **cross-entropy loss**.

$$\begin{aligned} \log p(y|x) &= \log\left[\hat{y}^y (1-\hat{y})^{1-y}\right] \\ &= y\log\hat{y} + (1-y)\log(1-\hat{y}) \qquad (5.10) \end{aligned}$$

The negative log of this probability is a convenient loss metric since it goes from 0 (negative log of 1, no loss) to infinity (negative log of 0, infinite loss). This loss function also ensures that as the probability of the correct answer is maximized, the probability of the incorrect answer is minimized; since the two sum to one, any increase in the probability of the correct answer is coming at the expense of the incorrect answer. It's called the cross-entropy loss, because Eq. 5.10 is also the formula for the cross-entropy between the true probability distribution y and our estimated distribution ˆy.

Gradient Descent

Gradient descent is a method that finds a minimum of a function by figuring out in which direction(in the space of the parameters θ) the function's slope is rising the most steeply, and moving in the opposite direction.

For logistic regression, this loss function is conveniently **convex**. A convex function has just one minimum; there are no local minima to get stuck in, so gradient descent starting from any point is guaranteed to find the minimum.

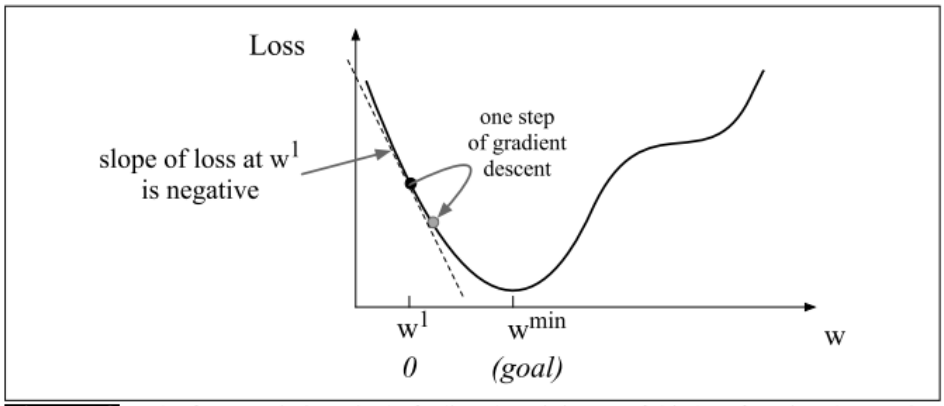


Figure 5.3 The first step in iteratively finding the minimum of this loss function, by moving *w* in the reverse direction from the slope of the function. Since the slope is negative, we need to move *w* in a positive direction, to the right. Here superscripts are used for learning steps, so w^1 means the initial value of *w* (which is 0), w^2 at the second step, and so on.

The gradient descent algorithm answers this question by finding the **gradient** of the loss function at the current point and moving in the opposite direction. The gradient of a function of many variables is a vector pointing in the direction of the greatest increase in a function. The gradient is a multi-variable generalization of the slope, so for a function of one variable like the one in Fig. 5.3, we can informally think of the gradient as the slope. The dotted line in Fig. 5.3 shows the slope of this hypothetical loss function at point w = w1. You can see that the slope of this dotted line is negative. Thus to find the minimum, gradient descent tells us to go in the opposite direction: moving w in a positive direction.

The magnitude of the amount to move in gradient descent is the value of the slope $\frac{d}{dw}L(f(x;w),y)$ weighted by a **learning rate** η. A higher (faster) learning rate means that we should move w more on each step.

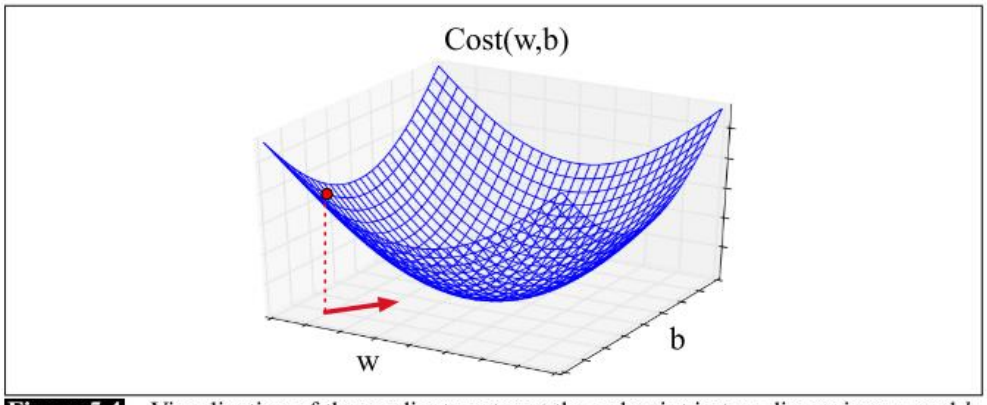


Figure 5.4 Visualization of the gradient vector at the red point in two dimensions w and b, showing the gradient as a red arrow in the x-y plane.

The Gradient for Logistic Regression

$$\frac{\partial L_{CE}(\hat{y},y)}{\partial w_j} = [\sigma(w\cdot x+b)-y]x_j \tag{5.18}$$

Note in Eq. 5.18 that the gradient with respect to a single weight w_j represents a very intuitive value: the difference between the true y and our estimated $\hat{y} = \sigma(w\cdot x+b)$ for that observation, multiplied by the corresponding input value x_j.

The Stochastic Gradient Descent Algorithm

online algorithm that minimizes the loss function by computing its gradient after each training example, and nudging θ in the right direction (the opposite direction of the gradient)

The learning rate η is a hyperparameter that must be adjusted.

Mini-batch training

Stochastic gradient descent is called stochastic because it chooses a single random example at a time, moving the weights so as to improve performance on that single example. That can result in very choppy movements, so it's common to compute the gradient over batches of training instances rather than a single instance.

in **batch training** we compute the gradient over the entire dataset.

A **compromise is mini-batch training**: we train on a group of m examples (perhaps 512, or 1024) that is less than the whole dataset. (If m is the size of the dataset, then we are doing batch gradient descent; if m = 1, we are back to doing stochastic gradient descent). Mini-batch training also has the advantage of computational efficiency. The mini-batches can easily be vectorized, choosing the size of the minibatch based on the computational resources. This allows us to process all the examples in one mini-batch in parallel and then accumulate the loss, something that's not possible with individual or batch training.

The mini-batch gradient is the average of the individual gradients from Eq. 5.18:

$$\frac{\partial Cost(\hat{y},y)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}\left[\sigma(w\cdot x^{(i)}+b)-y^{(i)}\right]x_j^{(i)} \qquad (5.21)$$

Regularization

When the model fits perfectly to the details of the training set, **overfitting** occurs. A good model should be able to **generalize** well from the training data to the unseen test set, but a model that overfits will have poor generalization.

To avoid overfitting, a new regularization term R(θ) is added to the objective function in Eq. 5.13, resulting in the following objective for a batch of m examples (slightly rewritten from Eq. 5.13 to be maximizing log probability rather than minimizing loss, and removing the 1

m term which doesn't affect the argmax):

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}}\sum_{i=1}^{m}\log P(y^{(i)}|x^{(i)}) - \alpha R(\theta) \qquad (5.22)$$

R(θ) is used to penalize large weights.

- Thus a setting of the weights that matches the training data perfectly— but uses many weights with high values to do so—will be penalized more than a setting that matches the data a little less well, but does so using smaller weights.
- Two common ways to compute regularization term R(θ):
  - **L2 Regularization (ridge regression) :** quadratic function of the weight values, named because it uses the (square of the) L2 norm of the weight values. The L2 norm, $||\theta||_2$, is the same as the Euclidean distance of the vector θ from the origin. If θ consists of *n* weights, then:

$$R(\theta) = ||\theta||_2^2 = \sum_{j=1}^{n}\theta_j^2 \qquad (5.23)$$

The L2 regularized objective function becomes:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}}\left[\sum_{i=1}^{m}\log P(y^{(i)}|x^{(i)})\right] - \alpha\sum_{j=1}^{n}\theta_j^2 \qquad (5.24)$$

  - **L1 regularization (lasso regression)**: linear function of the weight values, named after the L1 norm $||W||_1$, the sum of the absolute values of the weights, or Manhattan distance.

The L1 regularized objective function becomes:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\sum_{1=i}^{m} \log P(y^{(i)}|x^{(i)}) \right] - \alpha \sum_{j=1}^{n} |\theta_j| \qquad (5.26)$$

- Comparison.
  - L2 is easier to optimize, coz of derivative.
  - L2 prefers weight vectors with many small weights, L1 prefers sparse solutions with some larger weights but many more weights set to zero. Thus, L1 regularization leads to much sparser weight vectors, ie. far fewer features.
  - Both L1 and L2 regularization have Bayesian interpretations as constraints on the prior of how weights should look.
    - L1-reg can be viewed as a Laplace prior on the weights.
    - L2-reg corresponds to assuming that weights are distributed according to a Gaussian distribution with mean μ = 0.

Multinomial logistic regression (MLR)

Also called SoftMax regression. Here, the target y is a variable that ranges over more than two classes; we want to know the probability of y being in each potential class c ∈C, p(y = c|x).

Multinomial logistic Classifier uses a generalization of the sigmoid, called the softmax function, to compute the probability p(y = c|x). For a vector of z of dimensionality k, the SoftMax can be defined as:

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{k} \exp(z_j)} \quad 1 \le i \le k \qquad (5.30)$$

Again like the sigmoid, the input to the softmax will be the dot product between a weight vector w and an input vector x (plus a bias). But now we'll need separate weight vectors (and bias) for each of the K classes.

$$p(y = c|x) = \frac{\exp(w_c \cdot x + b_c)}{\sum_{j=1}^{K} \exp(w_j \cdot x + b_j)} \qquad (5.32)$$

Like the sigmoid, the softmax has the property of squashing values toward 0 or 1.

Features in Multinomial Logistic Regression

Like binary logistic regression, with one difference that we'll need separate weight vectors (and biases) for each of the K classes.

In binary classification a positive weight w_5 on a feature influences the classifier toward y = 1 (+ve sentiment) and a -ve weight influences it toward y = 0 (-ve sentiment) with the absolute value indicating how important the feature is.

For multinominal logistic regression, by contrast, with separate weights for each class, a feature can be evidence for or against each individual class.

Learning in Multinomial Logistic Regression

- The loss function for multinomial logistic regression generalizes the loss function for binary logistic regression from 2 to K classes. cross-entropy loss for binary logistic regression:

$$L_{CE}(\hat{y},y) = -\log p(y|x) = -[y\log\hat{y} + (1-y)\log(1-\hat{y})] \quad (5.33)$$

- The loss function for a single example x is thus the sum of the logs of the K output classes, each weighted by yk, the probability of the true class:

$$\begin{aligned} L_{CE}(\hat{y},y) &= -\sum_{k=1}^{K} y_k \log \hat{y}_k \\ &= -\sum_{k=1}^{K} y_k \log \hat{p}(y=k|x) \end{aligned} \quad (5.34)$$

- Because only one class (let's call it i) is the correct one, the vector y takes the value 1 only for this value of k, i.e., has yi = 1 and y j = 0 ∀ j ・= i. A vector like this, with one value=1 and the rest 0, is called a one-hot vector. The terms in the sum in Eq. 5.34 will thus be 0 except for the term corresponding to the true class, i.e.:

$$\begin{aligned} L_{CE}(\hat{y},y) &= -\sum_{k=1}^{K} \mathbb{1}\{y=k\} \log \hat{p}(y=k|x) \\ &= -\sum_{k=1}^{K} \mathbb{1}\{y=k\} \log \frac{\exp(w_k \cdot x + b_k)}{\sum_{j=1}^{K} \exp(w_j \cdot x + b_j)} \end{aligned} \quad (5.35)$$

  - w_k to mean the vector of weights from each input xi to the output node k,
  - indicator function $\mathbb{1}\{\}$ evaluates to 1 if the condition in the brackets is true and to 0 otherwise.

Cross-entropy loss is simply the log of the output probability corresponding to the correct class, and we therefore also call this the **negative log likelihood loss**:

$$\begin{aligned} L_{CE}(\hat{y},y) &= -\log \hat{y}_k, \quad \text{(where } k \text{ is the correct class)} \\ &= -\log \frac{\exp(w_k \cdot x + b_k)}{\sum_{j=1}^{K} \exp(w_j \cdot x + b_j)} \quad \text{(where } k \text{ is the correct class)} \end{aligned} \quad (5.36)$$

Interpreting models

Interpretability can be hard to define strictly, but the core idea is that as humans we should know why our algorithms reach the conclusions they do. Because the features to logistic regression are often human-designed, one way to understand a classifier's decision is to understand the role each feature plays in the decision. Logistic regression can be combined with statistical tests (the likelihood ratio test, or the Wald test); investigating whether a particular feature is significant by one of these tests, or inspecting its magnitude (how large is the weight w associated with the feature?) can help us interpret why the classifier made the decision it makes. This is enormously important for building transparent models.

logistic regression in NLP and many other fields is widely used as an analytic tool for testing hypotheses about the effect of various explanatory variables (features).

Vector Semantics and Embeddings

Words that occur in similar contexts tend to have similar meanings. This link between similarity in how words are distributed and similarity in what they mean is called the **distributional hypothesis**.

Vector semantics, which instantiates this linguistic hypothesis by learning representations of the meaning of words, called **embeddings**, directly from their distributions in texts. These representations are used in every natural language processing application that makes use of meaning, and the **static embeddings** we introduce here underlie the more powerful dynamic or **contextualized embeddings** like **BERT.**

Representation learning, automatically learning useful representations of the input text.

Finding such **self-supervised** ways to learn representations of the input, instead of creating representations by hand via **feature engineering**, is an important focus of NLP research.

Lexical Semantics

It is linguistic study of word meaning.

Generally, a model of word meaning should allow us to draw inferences to address meaning-related tasks like question-answering or dialogue.

Lemmas and Senses:

```
mouse (N)
1.  any of numerous small rodents...
2.  a hand-operated device that controls a cursor...
```

- Here the form *mouse* is the **lemma**, also called the **citation form**.
- *mouse* is lemma for word *mice*, *sing -> sing, sang, sung.*

each lemma can have multiple meanings; the lemma *mouse* can refer to *the rodent* or *the cursor control device*. These sense of meaning of *mouse* is called **word sense**.
Lemmas can be **polysemous** (have multiple senses) can make interpretation difficult.

To discuss the problem of polysemy, and introduce **word sense disambiguation**, the task of determining which sense of a word is being used in a particular context.

Synonymy:

One important component of word meaning is the relationship between word senses.

couch/sofa *vomit/throw up* *filbert/hazelnut* *car/automobile*

here two words have the same **propositional meaning**.

principle of contrast, difference in linguistic form is always associated with some difference in meaning. For example, the word H2O → scientific contexts and would be inappropriate in a hiking guide—water would be more appropriate— and this genre difference is part of the meaning of the word.

Word Similarity

While words don't have many synonyms, most words do have lots of similar words.

vanish	disappear	9.8
belief	impression	5.95
muscle	bone	3.65
modest	flexible	0.98
hole	agreement	0.3

In moving from synonymy to similarity, it will be useful to shift from talking about relations between word senses (like synonymy) to relations between words (like similarity). *Cats* and *dogs* are not synonyms but similar.

The notion of word **similarity** is very useful in larger semantic tasks. It can help in computing how similar the meaning of two phrases or sentences are, a very important component of tasks like question answering, paraphrasing, and summarization.

Word Relatedness

also traditionally called word **association** in psychology.

Coffee is not similar to *cup*; but *coffee* and *cup* are clearly related; they are associated by co-participating in an everyday event.

Relatedness between words – when they belong to same **semantic field**.

E.g. - hospitals (surgeon, scalpel, nurse, anesthetic, hospital), restaurants (waiter, menu, plate, food, chef), etc.

Semantic fields are also related to **topic models**, like **Latent Dirichlet Allocation, LDA**, which apply unsupervised learning on large sets of texts to induce sets of associated words from text. Semantic fields and topic models are very useful tools for discovering topical structure in documents.

Later we will introduce more relations between senses like **hypernymy** or **IS-A, antonymy** (opposites) and **meronymy** (part-whole relations).

Semantic Frames and Roles

A **semantic frame** is a set of words that denote perspectives or participants in a particular type of event. A commercial transaction, for example, is a kind of event in which one entity trades money to another entity in return for some good or service, after which the good changes hands or perhaps the service is performed.

Frames have semantic roles (like buyer, seller, goods, money), and words in a sentence can take on these roles.

Knowing that *buy* and *sell* have this relation makes it possible for a system to know that a sentence like *Sam bought the book from Ling* could be paraphrased as *Ling sold the book to Sam*, and that Sam has the role of the *buyer* in the frame and Ling the *seller*.

Connotation

Finally, words have *affective meanings* or **connotations**. The word *connotation* has different meanings in different fields, but here we use it to mean the aspects of a word's meaning that are related to a writer or reader's emotions, sentiment, opinions, or evaluations.

Early work on affective meaning (Osgood et al., 1957) found that words varied along three important dimensions of affective meaning:

valence: the pleasantness of the stimulus
arousal: the intensity of emotion provoked by the stimulus
dominance: the degree of control exerted by the stimulus

	Valence	Arousal	Dominance
courageous	8.05	5.5	7.38
music	7.67	5.57	6.5
heartbreak	2.45	5.65	3.58
cub	6.71	3.95	4.24

This revolutionary idea that word meaning could be represented as a point in space.

Vector Semantics

To define the meaning of a word by its distribution in language use, meaning its neighbouring words or grammatical environments. The idea was that two words that occur in very similar **distributions** (whose neighbouring words are similar) have similar meanings.

The idea of vector semantics is to represent a word as a point in a multidimensional semantic space that is derived (in ways we'll see) from the distributions of word neighbours. Vectors for representing words are called **embeddings** (although the term is sometimes more strictly applied only to dense vectors like word2vec (Section 6.8), rather than sparse tf-idf or PPMI vectors (Section 6.3-Section 6.6)). The word "embedding" derives from its mathematical sense as a mapping from one space or structure to another, although the meaning has shifted.

Figure 6.1 A two-dimensional (t-SNE) projection of embeddings for some words and phrases, showing that words with similar meanings are nearby in space. The original 60-dimensional embeddings were trained for sentiment analysis. Simplified from Li et al. (2015) with colors added for explanation.

The fine-grained model of word similarity of vector semantics offers enormous power to NLP applications.

- Sentiment analysis: where classifiers depend on same words appearing in train and test sets. But by representing words as embedding, classifiers can assign sentiment as long as it sees some words with *similar meanings*.
- Semantic models can be learned automatically from text without supervision.
- two most commonly used models:
    - In the **tf-idf model**, an important baseline, the meaning of a word is defined by a simple function of the counts of nearby words. We will see that this method results in very long vectors that are sparse, i.e. mostly zeros (since most words simply never occur in the context of others).
    - the **word2vec model** family for constructing short, **dense** vectors that have useful semantic properties. We'll also introduce the **cosine**, the standard way to use embeddings to compute *semantic similarity*, between two words, two sentences, or two documents, an important tool in practical applications like question answering, summarization, or automatic essay grading.

Words and Vectors

Vector or distributional models ofmeaning are generally based on a **co-occurrence matrix**, a way of representing how often words co-occur. We'll look at two popular matrices: the term-document matrix and the term-term matrix.

Vectors and documents

In a term-document matrix, each row represents a word in the vocabulary and each column represents a document from some collection of documents.

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Figure 6.2 The term-document matrix for four words in four Shakespeare plays. Each cell contains the number of times the (row) word occurs in the (column) document.

The term-document matrix of Fig. 6.2 was first defined as part of the vector space model of information retrieval.

To review some basic linear algebra, a **vector** is, at heart, just a list or array of numbers. So, *As You Like It* is represented as the list [1,114,36,20] (the first column vector in Fig. 6.3) and *Julius Caesar* is represented as the list [7,62,1,2] (the third column vector).

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Figure 6.3 The term-document matrix for four words in four Shakespeare plays. The red boxes show that each document is represented as a column vector of length four.

A vector space is a collection of vectors, characterized by their **dimension**.

Here, the document vectors are of dimension 4, in real term-document matrices, the vectors representing each document would have dimensionality |V|, the vocabulary size.

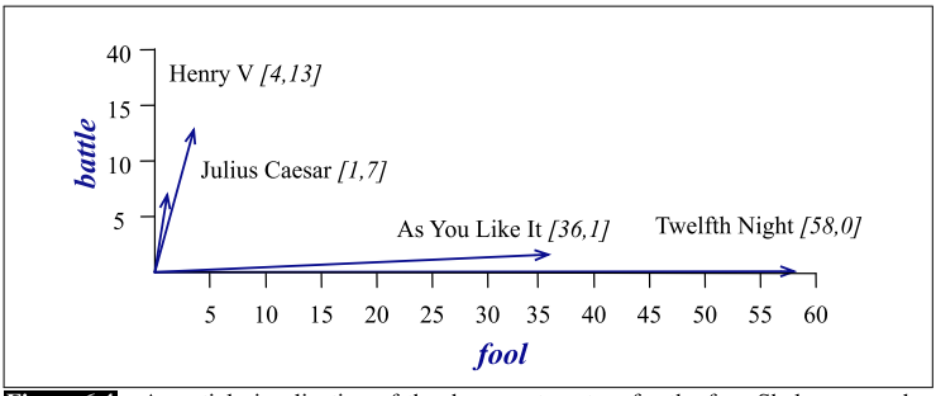


Figure 6.4 A spatial visualization of the document vectors for the four Shakespeare play documents, showing just two of the dimensions, corresponding to the words *battle* and *fool*. The comedies have high values for the *fool* dimension and low values for the *battle* dimension.

Term-document matrices were originally defined as a means of finding similar documents for the task of document **information retrieval**. Two documents that are similar will tend to have similar words, and if two documents have similar words their column vectors will tend to be similar.

More generally, the term-document matrix has |V| rows (one for each word type in the vocabulary) and D columns (one for each document in the collection); as we'll see, vocabulary sizes are generally in the tens of thousands, and the number of documents can be enormous.

Words as vectors: document dimensions

documents → represented as vectors in a vector space.

vector semantics →represent the meaning of *words*. We do this by associating each word with a word vector— a **row vector** rather than a column vector, hence with different dimensions, as shown in Fig. 6.5.

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Figure 6.5 The term-document matrix for four words in four Shakespeare plays. The red boxes show that each word is represented as a row vector of length four.

For documents, we saw that similar documents had similar vectors, because similar documents tend to have similar words.

This same principle applies to words: similar words have similar vectors because they tend to occur in similar documents. The term-document matrix thus lets us represent the meaning of a word by the documents it tends to occur in.

Words as vectors: word dimensions

The **term-term matrix**, also called the **word-word matrix** or the **term-context matrix**, in which the columns are labelled by words rather than documents.

- This matrix is thus of dimensionality |V|×|V| and each cell records the number of times the row (target) word and the column (context) word co-occur in some context in some training corpus.
- The context could be the document, in which case the cell represents the number of times the two words appear in the same document.
- It is most common, however, to use smaller contexts, generally a window around the word, for example of 4 words to the left and 4 words to the right.

is traditionally followed by	**cherry**	pie, a traditional dessert
often mixed, such as	**strawberry**	rhubarb pie. Apple pie
computer peripherals and personal	**digital**	assistants. These devices usually
a computer. This includes	**information**	available on the internet

- If we then take every occurrence of each word (say strawberry) and count the context words around it, we get a word-word co-occurrence matrix. Fig. 6.6 shows a simplified subset of the word-word co-occurrence matrix for these four words computed from the Wikipedia corpus (Davies, 2015)

	aardvark	...	computer	data	result	pie	sugar	...
cherry	0	...	2	8	9	442	25	...
strawberry	0	...	0	0	1	60	19	...
digital	0	...	1670	1683	85	5	4	...
information	0	...	3325	3982	378	5	13	...

Figure 6.6 Co-occurrence vectors for four words in the Wikipedia corpus, showing six of the dimensions (hand-picked for pedagogical purposes). The vector for *digital* is outlined in red. Note that a real vector would have vastly more dimensions and thus be much sparser.

- *cherry* and *strawberry* are more similar to each other (both *pie* and *sugar* tend to occur in their window) than they are to other words like *digital*; conversely, *digital* and *information* are more similar to each other than, say, to *strawberry*.

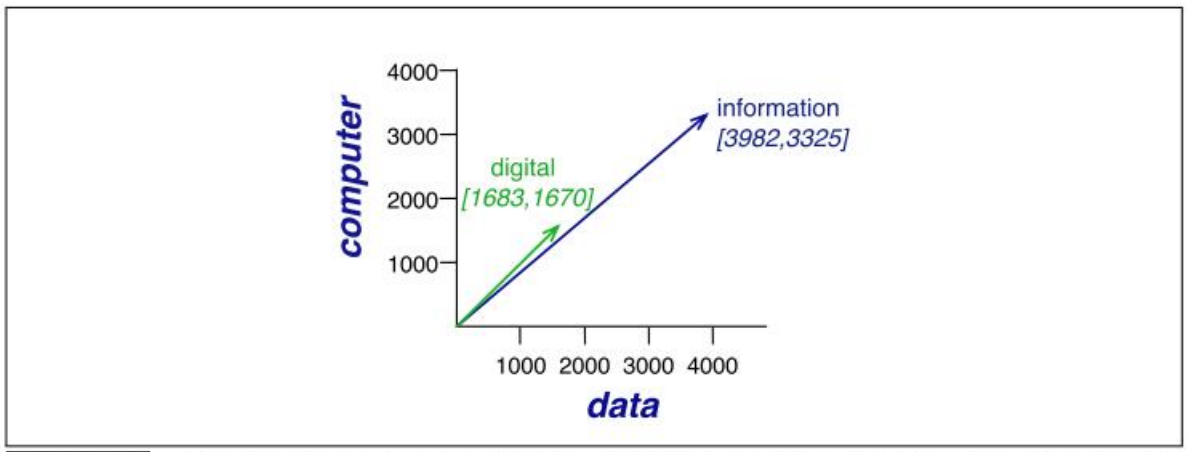


Figure 6.7 A spatial visualization of word vectors for *digital* and *information*, showing just two of the dimensions, corresponding to the words *data* and *computer*.

- Generally the words are humongous so, most of these numbers are sparse vector representations. There are efficient alternate algorithms.

Cosine for measuring similarity

The **cosine**—like most measures for vector similarity used in NLP—is based on the **dot product** operator from linear algebra, also called the **inner product**:

$$\text{dot product}(\mathbf{v},\mathbf{w}) = \mathbf{v}\cdot\mathbf{w} = \sum_{i=1}^{N} v_i w_i = v_1 w_1 + v_2 w_2 + \ldots + v_N w_N \qquad (6.7)$$

- The dot product acts as a similarity metric because it will tend to be high just when the two vectors have large values in the same dimensions. Alternatively, vectors that have zeros in different dimensions—orthogonal vectors—will have a dot product of 0, representing their strong dissimilarity.
- It favours the long vectors. Vector length:

$$|\mathbf{v}| = \sqrt{\sum_{i=1}^{N} v_i^2} \qquad (6.8)$$

- Problem: More frequent words have longer vectors, since they tend to co-occur with more words and have higher co-occurrence values with each of them. we'd like a similarity metric that tells us how similar two words are regardless of their frequency.
  Solution:
  This normalized dot product turns out to be the same as the cosine of the angle between the two vectors

$$\begin{aligned} \mathbf{a}\cdot\mathbf{b} &= |\mathbf{a}||\mathbf{b}|\cos\theta \\ \frac{\mathbf{a}\cdot\mathbf{b}}{|\mathbf{a}||\mathbf{b}|} &= \cos\theta \end{aligned} \qquad (6.9)$$

- The **cosine** similarity metric between two vectors v and w thus can be computed as:

$$\text{cosine}(\mathbf{v},\mathbf{w}) = \frac{\mathbf{v}\cdot\mathbf{w}}{|\mathbf{v}||\mathbf{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}} \qquad (6.10)$$

- In few cases, we pre-normalize each vectors by dividing it by its length, creating a unit vector of length 1, the dot product is the same as the cosine.
- Range: 1 for vectors pointing in the same direction,
  0 for orthogonal vectors
  -1 for vectors pointing in opposite directions.
- But since raw frequency values are non-negative, the cosine for these vectors ranges from 0–1.

	pie	data	computer
cherry	442	8	2
digital	5	1683	1670
information	5	3982	3325

$$\cos(\text{cherry},\text{information}) = \frac{442*5+8*3982+2*3325}{\sqrt{442^2+8^2+2^2}\sqrt{5^2+3982^2+3325^2}} = .017$$

$$\cos(\text{digital},\text{information}) = \frac{5*5+1683*3982+1670*3325}{\sqrt{5^2+1683^2+1670^2}\sqrt{5^2+3982^2+3325^2}} = .996$$

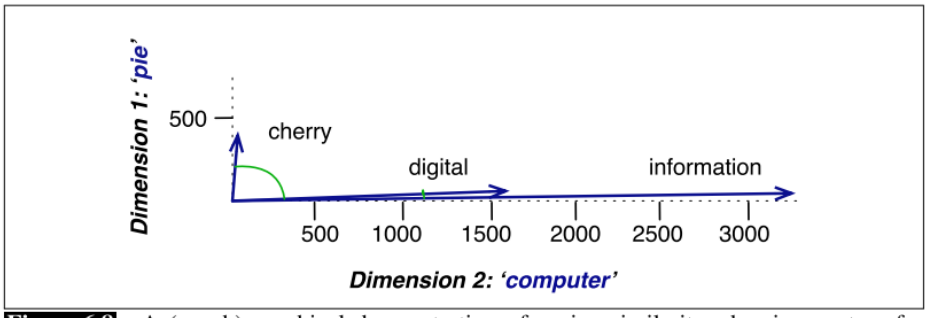


Figure 6.8 A (rough) graphical demonstration of cosine similarity, showing vectors for three words (*cherry*, *digital*, and *information*) in the two dimensional space defined by counts of the words *computer* and *pie* nearby. The figure doesn't show the cosine, but it highlights the angles; note that the angle between *digital* and *information* is smaller than the angle between *cherry* and *information*. When two vectors are more similar, the cosine is larger but the angle is smaller; the cosine has its maximum (1) when the angle between two vectors is smallest (0°); the cosine of all other angles is less than 1.

TF-IDF: Weighing terms in the vector

- Raw frequency is not the best measure of association between words. Raw frequency is very skewed and not very discriminative.
- It's a bit of a paradox. Words that occur nearby frequently (maybe pie nearby cherry) are more important than words that only appear once or twice. Yet words that are too frequent—ubiquitous, like the or good— are unimportant. How can we balance these two conflicting constraints?
  Solution:
  - **tf-idf** weighting, usually used when the dimensions are documents.
  - **PPMI** algorithm usually used when the dimensions are words.
- tf-idf weighting: is product of two terms, each term capturing one of the 2 intutions.
  - 1st is **Term frequency**: the frequency of the word t in the document d.
    - We squash the raw frequency by using the log10 of the frequency instead.
    - The intuition is that a word appearing 100 times in a document doesn't make that word 100 times more likely to be relevant to the meaning of the document. Because we can't take the log of 0, we normally add 1 to the count:

$$\text{tf}_{t,d} = \log_{10}(\text{count}(t,d)+1) \qquad (6.12)$$

    - If we use log weighting, terms which occur
      0 times in a document would have tf = log10(1) = 0,
      10 times in a document tf = log10(11) = 1.04,
      100 times tf = log10(101) = 2.004,
      1000 es tf = 3.00044, and so on.
  - 2nd factor in tf-idf is used to give a higher weight to words that occur only in a few documents.
    - Terms that are limited to a few documents are useful for discriminating those documents from the rest of the collection; terms that occur frequently across the entire collection aren't as helpful
    - **document frequency** df_t of a term *t* is the number of documents it occurs in. Document frequency is not the same as the **collection frequency** of a term,

which is the total number of times the word appears in the whole collection in any document.

- E.g. If considering Shakespeare's 37 plays – two words *Romeo* and *action*. The words have identical collection frequencies (they both occur 113 times in all the plays) but very different document frequencies, since Romeo only occurs in a single play. If our goal is to find documents about the romantic tribulations of Romeo, the word Romeo should be highly weighted, but not action:

	Collection Frequency	Document Frequency
Romeo	113	1
action	113	31

- We emphasize discriminative words like *Romeo* via the **inverse document frequency** or **idf** term weight.
  idf = (no. of doc in collection)/(no. of doc in which term *t* occurs)
- Because of the large number of documents in many collections, this measure too is usually squashed with a log function. The resulting definition for inverse document frequency (idf) is thus

$$\text{idf}_t = \log_{10}\left(\frac{N}{\text{df}_t}\right) \qquad (6.13)$$

- tf-idf weighted value:

$$w_{t,d} = \text{tf}_{t,d} \times \text{idf}_t \qquad (6.14)$$

- Note that the tf-idf values for the dimension corresponding to the word good have now all become 0; since this word appears in every document, the tf-idf weighting leads it to be ignored. Similarly, the word fool, which appears in 36 out of the 37 plays, has a much lower weight.

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	0.074	0	0.22	0.28
good	0	0	0	0
fool	0.019	0.021	0.0036	0.0083
wit	0.049	0.044	0.018	0.022

Figure 6.9 A tf-idf weighted term-document matrix for four words in four Shakespeare plays, using the counts in Fig. 6.2. For example the 0.049 value for *wit* in *As You Like It* is the product of $\text{tf} = \log_{10}(20+1) = 1.322$ and idf = .037. Note that the idf weighting has eliminated the importance of the ubiquitous word *good* and vastly reduced the impact of the almost-ubiquitous word *fool*.

- Tf-idf is also a great baseline, the simple thing to try first.

Pointwise Mutual Information (PMI)

An alternative weighting function to tf-idf, PPMI (positive pointwise mutual information), is used for term-term-matrices, when the vector dimensions correspond to words rather than documents.

The **more** the two words co-occur in our corpus than we would have a priori expected them to appear by chance.

Pointwise mutual information important concepts in NLP. It is a measure of how often two events x and y occur, compared with what we would expect if they were independent:

$$I(x,y) = \log_2 \frac{P(x,y)}{P(x)P(y)} \qquad (6.16)$$

x– target word y – context

Numerator – how often two words appear together.

Denominator - tells us how often we would expect the two words to co-occur assuming they each occurred independently.

- PMI values range from negative to positive infinity.
- -ve PMI → things are co-occurring less often → generally with large corpora.
- Generally we use Positive PMI (called **PPMI**) which replaces all -ve PMI values with zero.

$$\text{PPMI}(w,c) = \max(\log_2 \frac{P(w,c)}{P(w)P(c)}, 0) \qquad (6.18)$$

	computer	data	result	pie	sugar	count(w)
cherry	2	8	9	442	25	486
strawberry	0	0	1	60	19	80
digital	1670	1683	85	5	4	3447
information	3325	3982	378	5	13	7703
count(context)	4997	5673	473	512	61	11716

Figure 6.10 Co-occurrence counts for four words in 5 contexts in the Wikipedia corpus, together with the marginals, pretending for the purpose of this calculation that no other words/contexts matter.

$$\begin{aligned} P(\text{w=information, c=data}) &= \frac{3982}{11716} = .3399 \\ P(\text{w=information}) &= \frac{7703}{11716} = .6575 \\ P(\text{c=data}) &= \frac{5673}{11716} = .4842 \\ \text{ppmi(information,data)} &= \log 2(.3399/(.6575 * .4842)) = .0944 \end{aligned}$$

- PMI has the problem of being biased toward infrequent events; very rare words tend to have very high PMI values.
- **Solution 1**: One way to reduce this bias toward low frequency events is to slightly change the computation for P(c), using a different function Pα(c) that raises the probability of the context word to the power of α:

$$\text{PPMI}_\alpha(w,c) = \max(\log_2 \frac{P(w,c)}{P(w)P_\alpha(c)}, 0) \qquad (6.21)$$

$$P_\alpha(c) = \frac{count(c)^\alpha}{\sum_c count(c)^\alpha} \qquad (6.22)$$

  - α = 0.75 improved performance of embeddings on a wide range of tasks (drawing on a similar weighting used for skipgrams described below in Eq. 6.32). This works because raising the count to α = 0.75 increases the probability assigned to rare contexts, and hence lowers their PMI (Pα(c) > P(c) when c is rare).
- **Solution 2**: Using Laplace smoothing; Before computing PMI, a small constant k (values of 0.1-3 are common) is added to each of the counts, shrinking (discounting) all the non-zero values. The larger the k, the more the non-zero counts are discounted.

Applications of the tf-idf or PPMI vector models

Vector semantics model we've described so far represents a target word as a vector with dimensions corresponding either to the documents in a large collection (the term-document matrix) or to the counts ofwords in some neighboring window (the term-term matrix).

Tf-idf is used for document f() like deciding if two doc are similar.

We represent a document by taking the vectors of all the words in the document, and computing the centroid of all those vectors.

Uses of document similarity: information retrieval, plagiarism detection, news recommender systems etc.

PPMI model or the tf-idf model can be used to compute word similarity, for tasks like finding word paraphrases, tracking changes in word meaning, or automatically discovering meanings of words in different corpora.

> For example, we can find the 10 most similar words to any target word w by computing the cosines between w and each of the V−1 other words, sorting, and looking at the top 10.

Word2vec

After vector representation, lets use more powerful representation: **embeddings**,

> **short** & **dense vectors**: instead of vector entries being sparse, mostly-zero counts or functions of counts, the values will be real-valued numbers that can be negative.
>
> with number of dimensions *d* ranging from 50-1000, rather than the much larger vocabulary size |V| or number of documents D we've seen. d dimensions don't have a clear interpretation.
>
> dense vectors
>
> - work better in every NLP task than sparse vectors, but but we don't have any understanding of reason.
> - Needs small parameter space.
> - Classifiers have to learn fewer weights.
> - may also do a better job of capturing synonymy.

one method for computing embeddings:

skip-gram with negative sampling, sometimes called **SGNS**. The skip-gram algorithm is one of two algorithms in a software package called **word2vec**.

> The word2vec methods are fast, efficient to train, and easily available online with code and pretrained embeddings.
>
> Word2vec embeddings are static embeddings, →method learns one fixed embedding for each word in the vocabulary.

In later chapters, we'll introduce methods for learning dynamic **contextual embeddings** like the popular family of **BERT** representations, in which the vector for each word is different in different contexts.

Intuition of word2vec → instead of counting how often each word *w* occurs near, say, *apricot*, we'll instead train a classifier on a binary prediction task: "Is word w likely to show up near apricot?" We don't actually care about this prediction task; instead, we'll take the learned classifier weights as the word embeddings.

- The revolutionary intuition here is that we can just use running text as implicitly supervised training data for such a classifier.
- word *c* that occurs near the target word *apricot* acts as gold 'correct answer' to the question "Is word *c* likely to show up near *apricot*?" This method, often called **self-supervision**, avoids the need for any sort of hand-labelled supervision signal.

Word2vec is simpler model than NN language model in 2 ways:

- word2vec simplifies the task (making it binary classification instead of word prediction)
- word2vec simplifies the architecture (training a logistic regression classifier instead of a multi-layer neural network with hidden layers that demand more sophisticated training algorithms)

The intuition of skip-gram is:

1. Treat the target word and a neighbouring context word as positive examples.
2. Randomly sample other words in the lexicon to get negative samples.
3. Use logistic regression to train a classifier to distinguish those two cases.
4. Use the learned weights as the embeddings.

The classifier

target word - *apricot*, assume a window of ±2 context words:

```
... lemon,  a [tablespoon of apricot jam,      a] pinch ...
              c1         c2    w     c3       c4
```

Goal: train a classifier such that, given a tuple (*w,c*) of a target word *w* paired with a candidate context word c (for example (*apricot, jam*), or perhaps (*apricot, aardvark*)) it will return the probability that c is a real context word (true for *jam*, false for *aardvark*):

$$P(+|w,c) \qquad (6.24)$$

How does the classifier compute the probability P?

Solution: To compute similarity between these dense embeddings, we rely on the intuition that two vectors are similar if they have a high **dot product** (after all, cosine is just a normalized dot product). In other words:

$$Similarity(w,c) \approx \mathbf{c} \cdot \mathbf{w} \qquad (6.26)$$

The range is $-\infty$ to ∞ so, we convert dot product in **logistic** or **sigmoid** function σ(x)

$$P(+|w,c) = \sigma(\mathbf{c}\cdot\mathbf{w}) = \frac{1}{1+\exp(-\mathbf{c}\cdot\mathbf{w})} \quad (6.28)$$

For -ve, $$P(-|w,c) = 1 - P(+|w,c)$$
$$= \sigma(-\mathbf{c}\cdot\mathbf{w}) = \frac{1}{1+\exp(\mathbf{c}\cdot\mathbf{w})} \quad (6.29)$$

These were for one word, but there are many context words in the window.

Skip-gram makes the simplifying assumption that all context words are independent, allowing us to just multiply their probabilities:

$$P(+|w,c_{1:L}) = \prod_{i=1}^{L} \sigma(\mathbf{c_i}\cdot\mathbf{w}) \quad (6.30)$$

$$\log P(+|w,c_{1:L}) = \sum_{i=1}^{L} \log \sigma(\mathbf{c_i}\cdot\mathbf{w}) \quad (6.31)$$

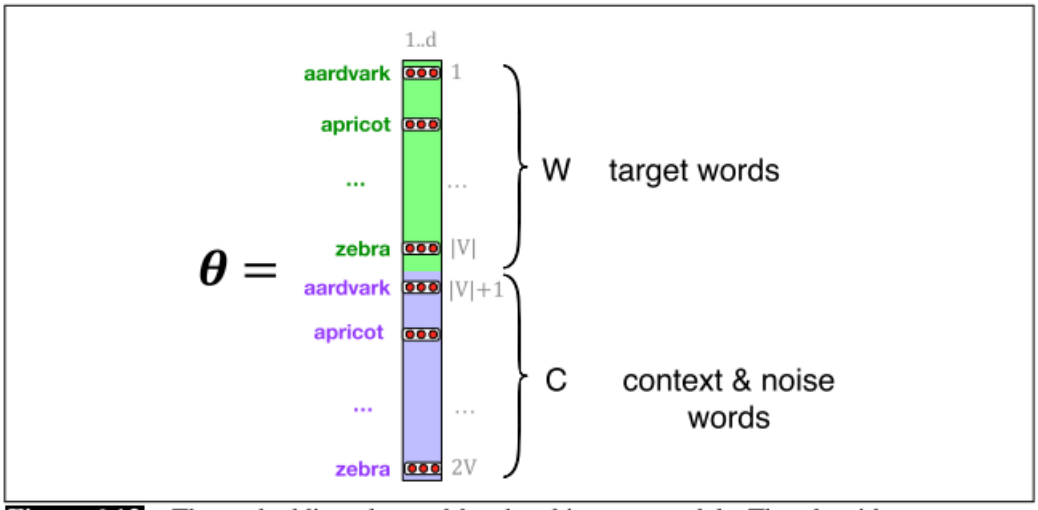


Figure 6.13 The embeddings learned by the skipgram model. The algorithm stores two embeddings for each word, the target embedding (sometimes called the input embedding) and the context embedding (sometimes called the output embedding). The parameter θ that the algorithm learns is thus a matrix of $2|V|$ vectors, each of dimension d, formed by concatenating two matrices, the target embeddings **W** and the context+noise embeddings **C**.

Skip gram store two embeddings for each word – one for target and another for context, so we need to learn 2 matrices **W** & **C**, each containing an embedding for every one of the |V| words in the vocabulary V.

Learning skip-gram embeddings

The learning algorithm for skip-gram embeddings takes as input a corpus of text, and a chosen vocabulary size N. It begins by assigning a random embedding vector for each of the N vocabulary words, and then proceeds to iteratively shift the embedding of each word *w* to be more like the

embeddings of words that occur nearby in texts, and less like the embeddings of words that don't occur nearby.

Binary classifier→ needs -ve examples as well.

Skip-gram with negative sampling (SGNS) uses more negative examples than positive examples (with the ratio between them set by a parameter k)

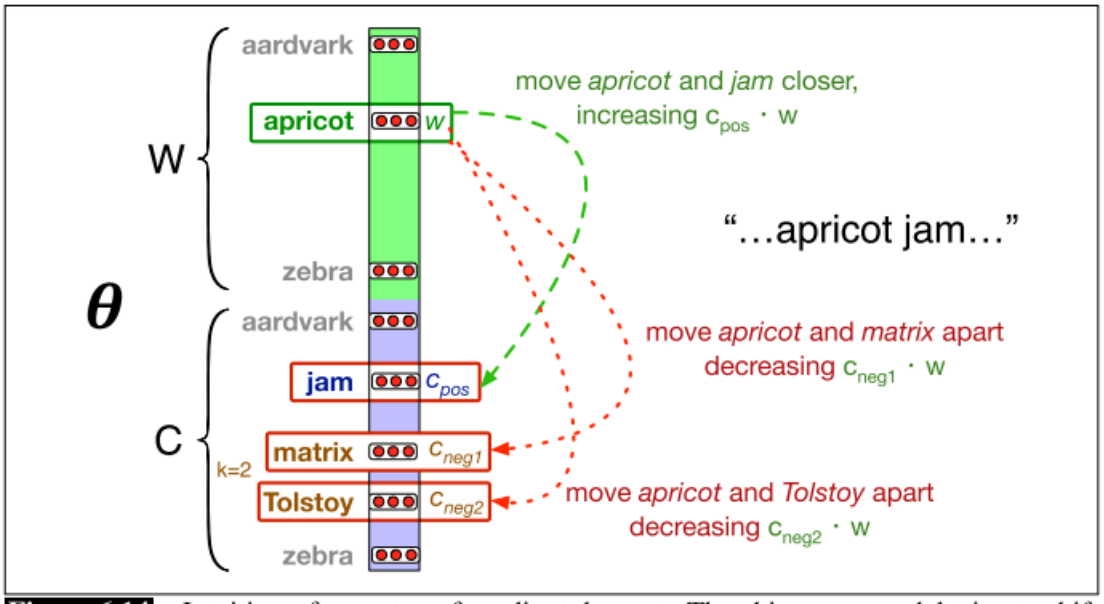


Figure 6.14 Intuition of one step of gradient descent. The skip-gram model tries to shift embeddings so the target embeddings (here for *apricot*) are closer to (have a higher dot product with) context embeddings for nearby words (here *jam*) and further from (lower dot product with) context embeddings for noise words that don't occur nearby (here *Tolstoy* and *matrix*).

If we consider one word/context pair (w,c_{pos}) with its k noise words $c_{neg1}...c_{neg\,k}$, we can express these two goals as the following

- loss function L to be minimized (hence the −); here the first term expresses that we want the classifier to assign the real context word c_{pos} a high probability of being a neighbor,
- the second term expresses that we want to assign each of the noise words c_{negi} a high probability of being a non-neighbor, all multiplied because we assume independence:

$$
\begin{aligned}
L_{CE} &= -\log\left[P(+|w,c_{pos})\prod_{i=1}^{k}P(-|w,c_{neg_i})\right] \\
&= -\left[\log P(+|w,c_{pos})+\sum_{i=1}^{k}\log P(-|w,c_{neg_i})\right] \\
&= -\left[\log P(+|w,c_{pos})+\sum_{i=1}^{k}\log\left(1-P(+|w,c_{neg_i})\right)\right] \\
&= -\left[\log\sigma(c_{pos}\cdot w)+\sum_{i=1}^{k}\log\sigma(-c_{neg_i}\cdot w)\right] \qquad (6.34)
\end{aligned}
$$

We minimize this loss using stochastic gradient descent.

$$\frac{\partial L_{CE}}{\partial c_{pos}} = [\sigma(\mathbf{c}_{pos} \cdot \mathbf{w}) - 1]\mathbf{w} \quad (6.35)$$

$$\frac{\partial L_{CE}}{\partial c_{neg}} = [\sigma(\mathbf{c}_{neg} \cdot \mathbf{w})]\mathbf{w} \quad (6.36)$$

$$\frac{\partial L_{CE}}{\partial w} = [\sigma(\mathbf{c}_{pos} \cdot \mathbf{w}) - 1]\mathbf{c}_{pos} + \sum_{i=1}^{k}[\sigma(\mathbf{c}_{neg_i} \cdot \mathbf{w})]\mathbf{c}_{neg_i} \quad (6.37)$$

We update the equations from time step *t* to *t+1* in stochastic GD.

$$\mathbf{c}_{pos}^{t+1} = \mathbf{c}_{pos}^{t} - \eta[\sigma(\mathbf{c}_{pos}^{t} \cdot \mathbf{w}^{t}) - 1]\mathbf{w}^{t} \quad (6.38)$$

$$\mathbf{c}_{neg}^{t+1} = \mathbf{c}_{neg}^{t} - \eta[\sigma(\mathbf{c}_{neg}^{t} \cdot \mathbf{w}^{t})]\mathbf{w}^{t} \quad (6.39)$$

$$\mathbf{w}^{t+1} = \mathbf{w}^{t} - \eta\left[[\sigma(\mathbf{c}_{pos} \cdot \mathbf{w}^{t}) - 1]\mathbf{c}_{pos} + \sum_{i=1}^{k}[\sigma(\mathbf{c}_{neg_i} \cdot \mathbf{w}^{t})]\mathbf{c}_{neg_i}\right] \quad (6.40)$$

Just as in logistic regression, then, the learning algorithm starts with randomly initializedWand C matrices, and then walks through the training corpus using gradient descent to move W and C so as to maximize the objective in Eq. 6.34 by making the updates in (Eq. 6.39)-(Eq. 6.40).

Recall that the skip-gram model **learns two separate embeddings** for each word i: **the target embedding $\mathbf{w}_i$** and the **context embedding $\mathbf{c}_i$**, stored in two matrices, the **target matrix W** and the **context matrix C**. It's common to just add them together, representing word i with the vector **$\mathbf{w}_i + \mathbf{c}_i$**. Alternatively, we can throw away the C matrix and just represent each word i by the vector wi.

As with the simple count-based methods like tf-idf, the context window size L affects the performance of skip-gram embeddings, and experiments often tune the parameter L on a devset.

Other kind of static embeddings

An extension of word2vec – **fasttext** addresses a problem with word2vec as we have presented it so far: it has no good way to deal with **unknown words**.

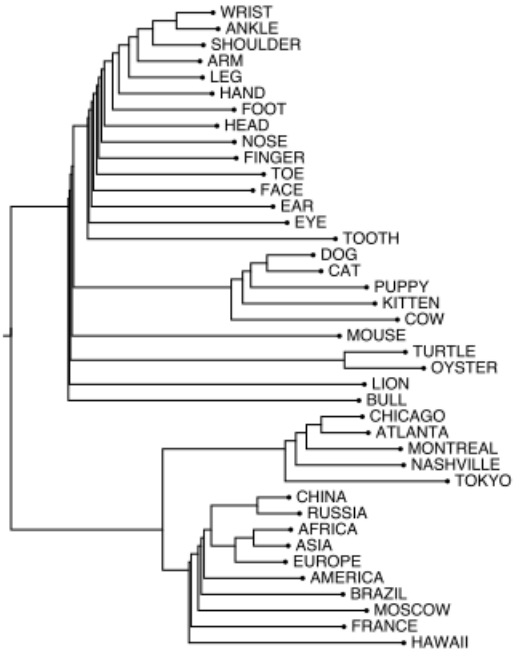
A related problem is word sparsity, such as in languages with rich morphology, where some of the many forms for each noun and verb may only occur rarely.

> Fasttext deals with these problems by using subword models, representing each word as itself plus a bag of constituent n-grams, with special boundary symbols < and > added to each word.
>
> with n = 3 the word where would be represented by the sequence <where> plus the character n-grams:
>
> *<wh, whe, her, ere, re>*
>
> Then a skipgram embedding is learned for each constituent n-gram, and the word *where* is represented by the sum of all of the embeddings of its constituent n-grams.
>
> Unknown words can then be presented only by the sum of the constituent n-grams.

GloVe model – Global Vectors

based on capturing global corpus statistics.

GloVe is based on ratios of probabilities from the word-word cooccurrence matrix, combining the intuitions of count-based models like PPMI while also capturing the linear structures used by methods like word2vec.



Visualizing Embeddings

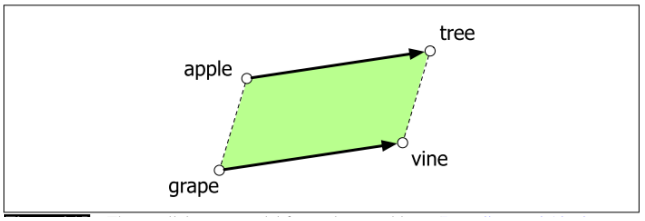
The simplest way to visualize the meaning of a word *w* embedded in a space - list most similar words to *w* by sorting the vectors for all words in the vocabulary by their cosine with the vector for *w*.

For example, the 7 closest words to *frog* using the GloVe embeddings are: *frogs, toad, litoria*, *leptodactylidae, rana, lizard, and eleutherodactylus.*

Another visualization – using clustering algo – shows hierarchical representation of similar words in embedding space ----------→

Most common visualization: project the 100 dimensions of a word down into 2 dimensions.

Another similar method is t-SNE:



Semantic properties of embeddings

Different types of similarity or association:

One parameter of vector semantic models that is relevant to both sparse tf-idf vectors and dense word2vec vectors is the size of the context window used to collect counts, generally between 1 -10 words on each side of the target word (for a total context of 2-20 words).

> Shorter context windows tend to lead to representations that are a bit more syntactic, since the information is coming from immediately nearby words. When the vectors are computed from short context windows, the most similar words to a target word w tend to be semantically similar words with the same parts of speech.
>
> When vectors are computed from long context windows, the highest cosine words to a target word w tend to be words that are topically related but not similar.

It's also often useful to distinguish two kinds of similarity or association between words.

Two words have **first-order co-occurrence** (sometimes called **syntagmatic association**) if they are typically nearby each other. Thus, *wrote* is a first-order associate of *book* or *poem*.

Two words have **second-order co-occurrence** (sometimes called **paradigmatic association**) if they have similar neighbours. Thus, *wrote* is a second-order associate of words like *said* or *remarked*.

Analogy/Relational Similarity

Another semantic property of embeddings is their ability to capture relational meanings.

parallelogram model - *a is to b as a* is to what?* *apple: tree :: grape : ?*

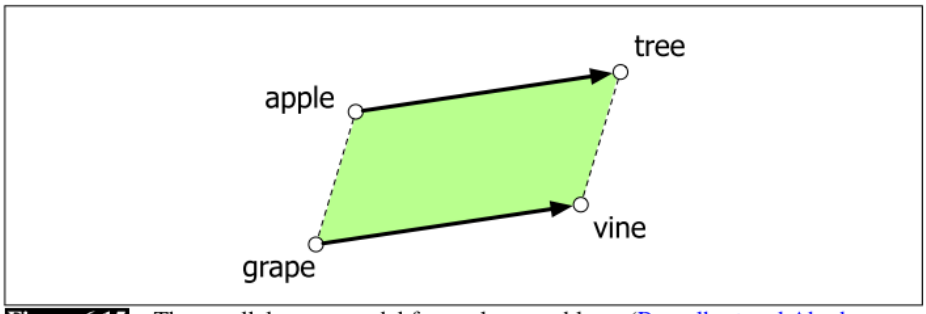


Figure 6.15 The parallelogram model for analogy problems (Rumelhart and Abrahamson, 1973): the location of $\overrightarrow{\text{vine}}$ can be found by subtracting $\overrightarrow{\text{apple}}$ from $\overrightarrow{\text{tree}}$ and adding $\overrightarrow{\text{grape}}$.

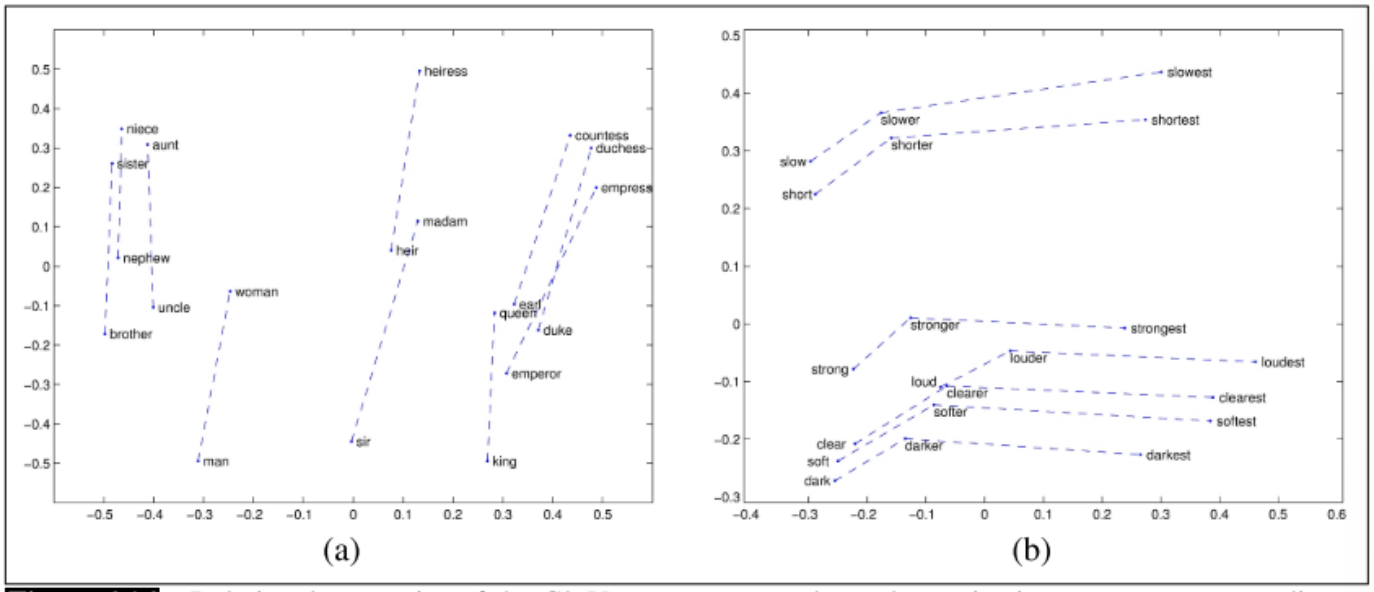


Figure 6.16 Relational properties of the GloVe vector space, shown by projecting vectors onto two dimensions. (a) $\overrightarrow{\text{king}} - \overrightarrow{\text{man}} + \overrightarrow{\text{woman}}$ is close to $\overrightarrow{\text{queen}}$. (b) offsets seem to capture comparative and superlative morphology (Pennington et al., 2014).

But it has some exceptions as well, *cherry: red*:: *potato*: ? → returns *potato* instead of *brown*.

Embeddings and Historical Semantics

Embeddings can also be a useful tool for studying how meaning changes over time, by computing multiple embedding spaces, each from texts written in a particular time period.

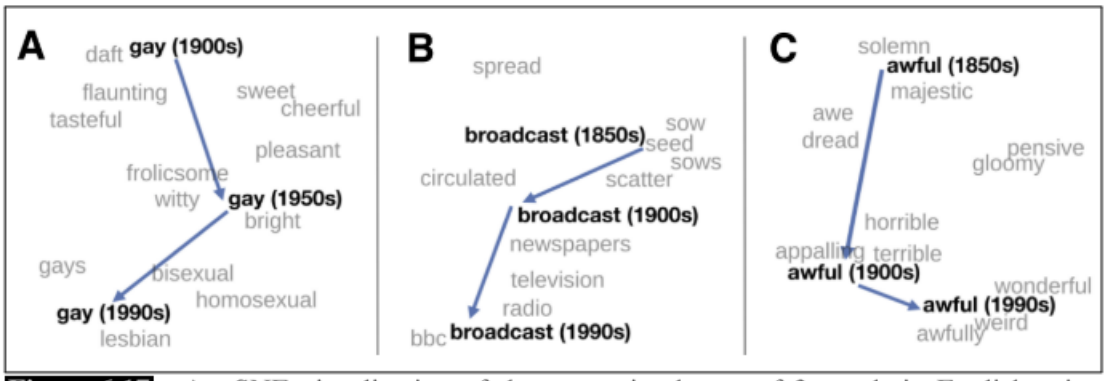


Figure 6.17 A t-SNE visualization of the semantic change of 3 words in English using word2vec vectors. The modern sense of each word, and the grey context words, are computed from the most recent (modern) time-point embedding space. Earlier points are computed from earlier historical embedding spaces. The visualizations show the changes in the word *gay* from meanings related to "cheerful" or "frolicsome" to referring to homosexuality, the development of the modern "transmission" sense of *broadcast* from its original sense of sowing seeds, and the pejoration of the word *awful* as it shifted from meaning "full of awe" to meaning "terrible or appalling" (Hamilton et al., 2016b).

Bias and Embeddings

In addition to their ability to learn word meaning from text, embeddings, alas, also reproduce the implicit biases and stereotypes that were latent in the text.

'man' - 'computer programmer' + 'woman' in word2vec embeddings trained on news text is 'homemaker', and that the embeddings similarly suggest the analogy 'father' is to 'doctor' as 'mother' is to 'nurse'

allocational harm – when the system allocates the resources unfairly to different groups.

Embeddings – reflect the statics of their input, also amplify bias, gendered terms becomes more gendered in embedding spaces as they were in the input text statics and biases are more exaggerated than in actual labor employment statistics.

African-American names like 'Leroy' and 'Shaniqua' had a higher GloVe cosine with unpleasant words while European-American names ('Brad', 'Greg', 'Courtney') had a higher cosine with pleasant words. These problems with embeddings are an example of a **representational harm** - caused by a system demeaning or even ignoring some social groups. Any embedding-aware algorithm that made use of word sentiment could thus exacerbate bias against African Americans.

> Solution – removal of such biases, by developing transformation of embedding space that removes gender stereotypes but preserves the definitional gender. However, although these sorts of debiasing may reduce bias in embeddings, they do not eliminate it

Evaluating Vector Models

Most important eval metric: extrinsic evaluation on tasks, i.e., using vectors in an NLP task and seeing whether this improves performance over some other model.

Nonetheless it is useful to have intrinsic evaluations. The most common metric is to test their performance on **similarity**, computing the correlation between an algorithm's word similarity scores and word similarity ratings assigned by humans.

WordSim-353 – used set of ratings from 0 to 10 for 353 noun pairs. E.g., (*plane, car*) had an average score of 5.77.

SimLex-999 - more difficult dataset that quantifies similarity (*cup, mug*) rather than relatedness (*cup, coffee*), and including both concrete and abstract adjective, noun and verb pairs.

TOEFL dataset is a set of 80 questions, each consisting of a target word with 4 additional word choices; the task is to choose which is the correct synonym, as in the example: *Levied* is closest in meaning to: *imposed, believed, requested, correlated*.

SUMMARY

- In vector semantics, a word is modeled as a vector—a point in high-dimensional space, also called an **embedding**. In this chapter we focus on **static embeddings**, in each each word is mapped to a fixed embedding.
- Vector semantic models fall into two classes: **sparse** and **dense**. In sparse models each dimension corresponds to a word in the vocabulary V and cells are functions of **co-occurrence counts**. The **term-document** matrix has a row for each word (**term**) in the vocabulary and a column for each document. The **word-context** or **term-term** matrix has a row for each (target) word in the vocabulary and a column for each context term in the vocabulary. Two sparse weightings are common: the **tf-idf** weighting which weights each cell by its **term frequency** and **inverse document frequency**, and **PPMI** (pointwise positive mutual information) most common for for word-context matrices.
- Dense vector models have dimensionality 50–1000. **Word2vec** algorithms like **skip-gram** are a popular way to compute dense embeddings. Skip-gram trains a logistic regression classifier to compute the probability that two words are 'likely to occur nearby in text'. This probability is computed from the dot product between the embeddings for the two words.
- Skip-gram uses stochastic gradient descent to train the classifier, by learning embeddings that have a high dot product with embeddings of words that occur nearby and a low dot product with noise words.
- Other important embedding algorithms include **GloVe**, a method based on ratios of word co-occurrence probabilities.
- Whether using sparse or dense vectors, word and document similarities are computed by some function of the **dot product** between vectors. The cosine of two vectors—a normalized dot product—is the most popular such metric.

7. Neural Networks and Neural Language Models

A modern neural network is a network of small computing units, each of which takes a vector of input values and produces a single output value.

Feedforward network – the computation proceeds iteratively from one layer of units to the next. The use of modern neural nets is often called **deep learning** because modern networks are often **deep** (have many layers).

Neural networks share much of the same mathematics as logistic regression. But neural networks are a more powerful classifier than logistic regression, and indeed a minimal neural network (technically one with a single 'hidden layer') can be shown to learn any function.

Difference between NN and LR:

- LR - we applied the regression classifier to many different tasks by developing many rich kinds of feature templates based on domain knowledge.
- NN- common to avoid rich hand-derived features, instead building NN that take raw words as inputs and learn to induce features as part of the process of learning to classify.

Units

A building block of NN that takes a set of real valued numbers as input, performs some computation on them, and produces an output.

It takes the weighted sum of its input and additional term – bias. w -> weights and x -> observations.

$$z = b + \sum_i w_i x_i \tag{7.1}$$

Considering the input as vector, we consider z in terms of weight vector w.

$$z = \mathbf{w} \cdot \mathbf{x} + b \tag{7.2}$$

Finally, instead of using z, a linear function of x, as the output, neural units apply a non-linear function f to z. We will refer to the output of this function as the **activation** value for the unit, a. Since we are just modelling a single unit, the activation for the node is in fact the final output of the network, which we'll generally call y. So the value y is defined as: $y = a = f(z)$

Sigmoid f()

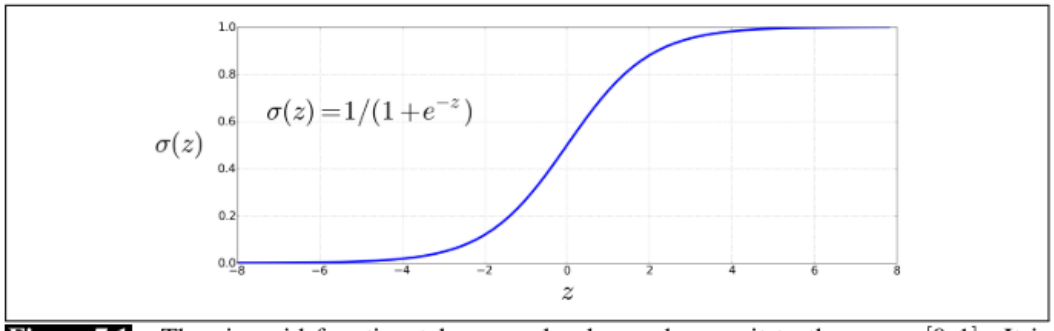


Figure 7.1 The sigmoid function takes a real value and maps it to the range $[0, 1]$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

Substituting Eq. 7.2 into Eq. 7.3 gives us the output of a neural unit:

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1 + \exp(-(\mathbf{w} \cdot \mathbf{x} + b))} \tag{7.4}$$

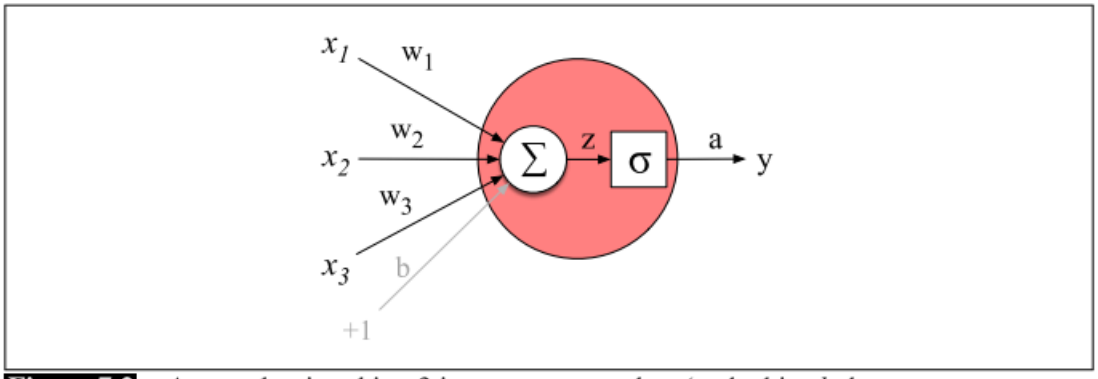


Figure 7.2 A neural unit, taking 3 inputs x_1, x_2, and x_3 (and a bias b that we represent as a weight for an input clamped at +1) and producing an output y. We include some convenient intermediate variables: the output of the summation, z, and the output of the sigmoid, a. In this case the output of the unit y is the same as a, but in deeper networks we'll reserve y to mean the final output of the entire network, leaving a as the activation of an individual node.

The sigmoid is not commonly used as an activation function.

A function that is very similar but almost always better is the **tanh function**, which is variant of sigmoid f() with range -1 to 1.

$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{7.5}$$

The simplest activation function, and perhaps the most commonly used, is the rectified linear unit, also called the **ReLU**, it's just the same as z when z is positive, and 0 otherwise:

$$y = max(z, 0) \tag{7.6}$$

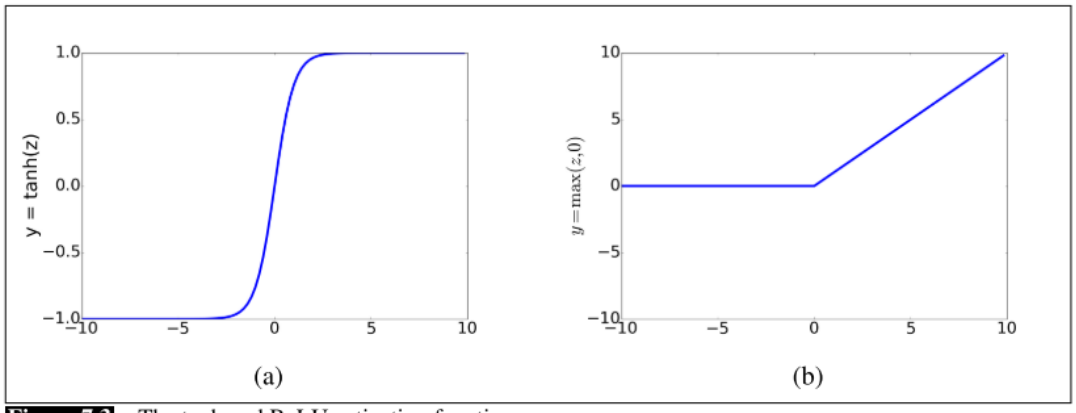


Figure 7.3 The tanh and ReLU activation functions.

Activation functions have different properties.

- tanh function has the nice properties of being smoothly differentiable and mapping outlier values toward the mean.
- rectifier function has nice properties that result from it being very close to linear.
- sigmoid or tanh functions, very high values of z result in values of y that are saturated, i.e., extremely close to 1, and have derivatives very close to 0.
    - Zero derivatives cause problems for learning, because we'll train networks by propagating an error signal backwards, multiplying gradients (partial derivatives) from each layer of the network; gradients that are almost 0 cause the error signal to get

smaller and smaller until it is too small to be used for training, a problem called the **vanishing gradient problem**.

- Rectifiers don't have this problem, since the derivative of ReLU for high values of z is 1 rather than very close to 0.

The XOR problem

that a single neural unit cannot compute some very simple functions of its input.

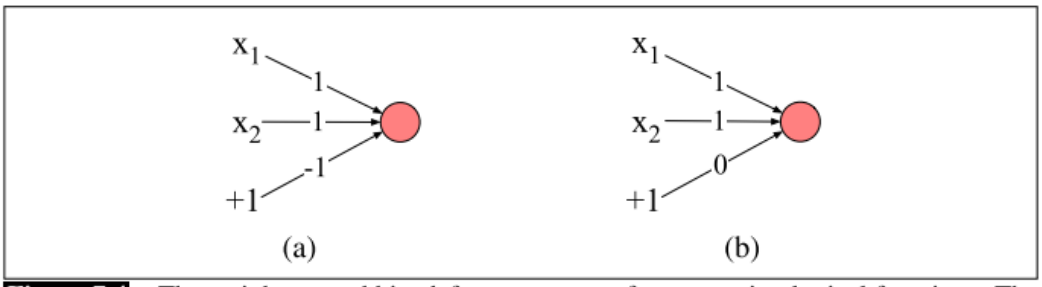


Figure 7.4 The weights w and bias b for perceptrons for computing logical functions. The inputs are shown as x_1 and x_2 and the bias as a special node with value $+1$ which is multiplied with the bias weight b. (a) logical AND, showing weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = -1$. (b) logical OR, showing weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = 0$. These weights/biases are just one from an infinite number of possible sets of weights and biases that would implement the functions.

AND				XOR			
x1	x2	y		x1	x2	y	
0	0	0		0	0	0	
0	1	0		0	1	1	
1	0	0		1	0	1	
1	1	1		1	1	0	

OR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

We say that **XOR is not a linearly separable function**. Of course, we could draw a boundary with a curve, or some other function, but not a single line.

The solution: neural networks

XOR function cannot be calculated by a single perceptron, it can be calculated by a layered network of units.

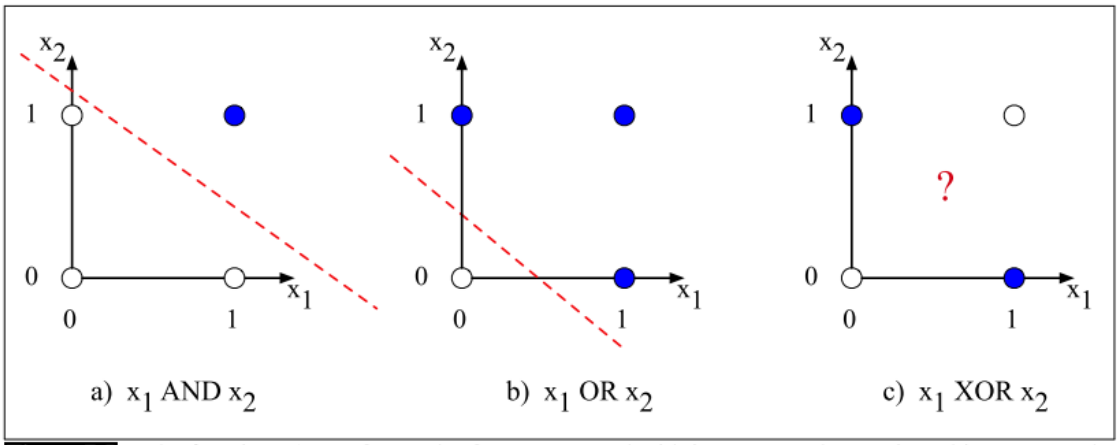


Figure 7.5 The functions AND, OR, and XOR, represented with input x_1 on the x-axis and input x_2 on the y axis. Filled circles represent perceptron outputs of 1, and white circles perceptron outputs of 0. There is no way to draw a line that correctly separates the two categories for XOR. Figure styled after Russell and Norvig (2002).

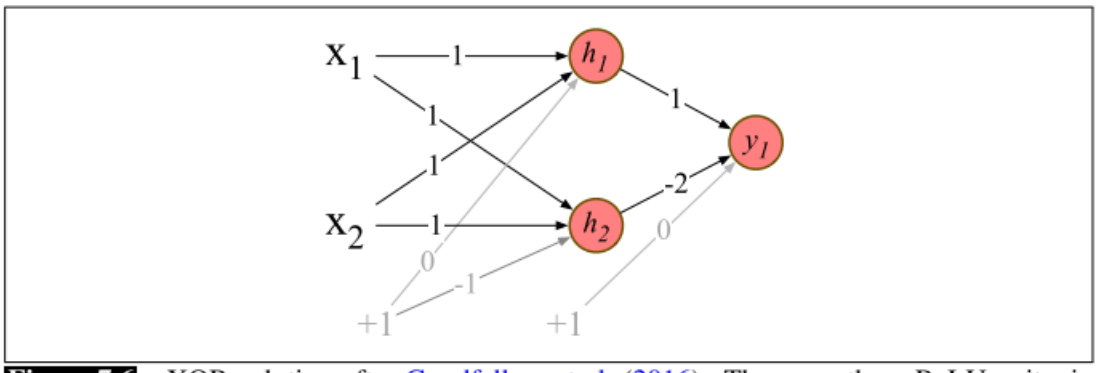


Figure 7.6 XOR solution after Goodfellow et al. (2016). There are three ReLU units, in two layers; we've called them h_1, h_2 (h for "hidden layer") and y_1. As before, the numbers on the arrows represent the weights w for each unit, and we represent the bias b as a weight on a unit clamped to +1, with the bias weights/units in gray.

The weights for NN are learned automatically using the error backpropagation algorithm → hidden layers will learn to form useful representations.

Feedforward NNs

A multilayer network in which the units are connected with **no cycles**.

the outputs from units in each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers.

Also called as **multi-layer perceptrons**. (or MLPs) this is a technical misnomer, purely linear, but modern networks are made up of units with non-linearities like sigmoids), but at some point the name stuck.

FFN have three kind of nodes:

- **Input units**, the input layer x is a vector of simple scalar values.
- **Hidden units**, core of the neural network is the hidden layer h formed of hidden units h_i, each of which is a neural unit, taking a weighted sum of its inputs and then applying a non-linearity. In standard architecture, each layer is fully connected. [each unit in each layer takes as input the outputs from all the units in the previous layer, and there is a link between every pair of units from two adjacent layers. Thus each hidden unit sums over all the input units]
- **Output units** – h, This output could be a real valued number, but in many cases the goal of the network is to make some sort of classification decision. For sentiment it could be +ve or -ve, for multinomial classification (e.g., POS) where the output is in the form of probability, sum should be 1.

  We can perform normalization of the output using softmax function.

More details on feedforward networks

The algorithm for computing the forward step in an n-layer feedforward network, given the input vector $a^{[0]}$ is thus simply:

$$\begin{aligned}&\textbf{for } i \textbf{ in } 1..n\\ &\quad \mathbf{z}^{[i]} = \mathbf{W}^{[i]}\,\mathbf{a}^{[i-1]} + \mathbf{b}^{[i]}\\ &\quad \mathbf{a}^{[i]} = g^{[i]}(\mathbf{z}^{[i]})\\ &\hat{\mathbf{y}} = \mathbf{a}^{[n]}\end{aligned}$$

The activation functions g(·) are generally different at the final layer. Thus $g^{[2]}$ might be SoftMax for multinomial classification or sigmoid for binary classification, while ReLU or tanh might be the activation function g(·) at the internal layers.

More on the need for non-linear activation functions

one of the reasons we use non-linear activation functions for each layer in a neural network is that if we did not, the resulting network is exactly equivalent to a single-layer network.

$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}$$
$$\mathbf{z}^{[2]} = \mathbf{W}^{[2]}\mathbf{z}^{[1]} + \mathbf{b}^{[2]}$$

So without non-linear activation functions, a multilayer network is just a notational variant of a single layer network with a different set of weights, and we lose all the representational power of multilayer networks.

Replacing the bias unit

b is replaced by a dummy node – a_0 to each layer whose value will be 1. This dummy node still the associated weights, and weight represents the bias value *b*. So the eq. 7.15

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \qquad (7.15)$$

becomes

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x}) \qquad (7.16)$$

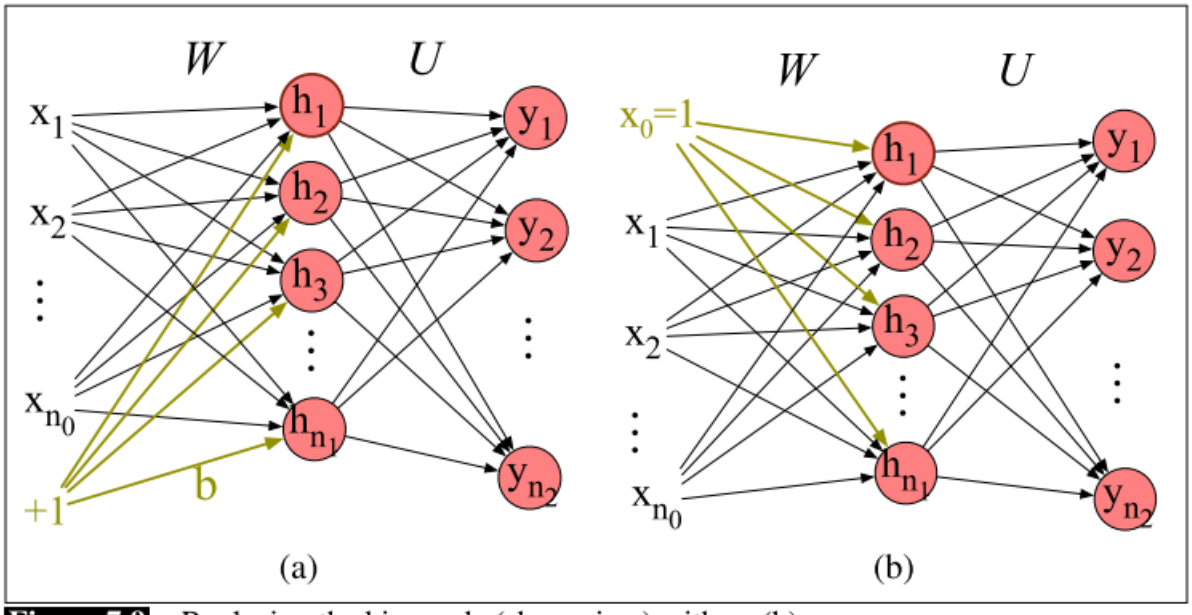


Figure 7.9 Replacing the bias node (shown in a) with x_0 (b).

Feedforward networks for NLP: Classification

Adding a hidden layer to our logistic regression regression classifier allows the network to represent the non-linear interactions between features. This alone might give us a better sentiment classifier.

Instead of using hand-built human-engineered features as the input to our classifier, we draw on deep learning's ability to learn features from the data by representing words as word2vec or GloVe embeddings.

The idea of using word2vec or GloVe embeddings as our input representation and more generally the idea of relying on another algorithm to have already learned an embedding representation for our input words — is called **pretraining**.

Feedforward Neural Language Modeling

language modelling: predicting upcoming words from prior word context.

Modern neural language models use more powerful architectures like – RRNs, Transformers; the feedforward language model introduces many of the important concepts of neural language modelling.

Advantage of neural language model (LM) over n-gram LM.

- neural language models can handle much longer histories.
- Generalize better over context of similar words
- More accurate at word-prediction.

Disadvantages:

- Complex
- Slower to train.
- Less interpretable than n-gram.

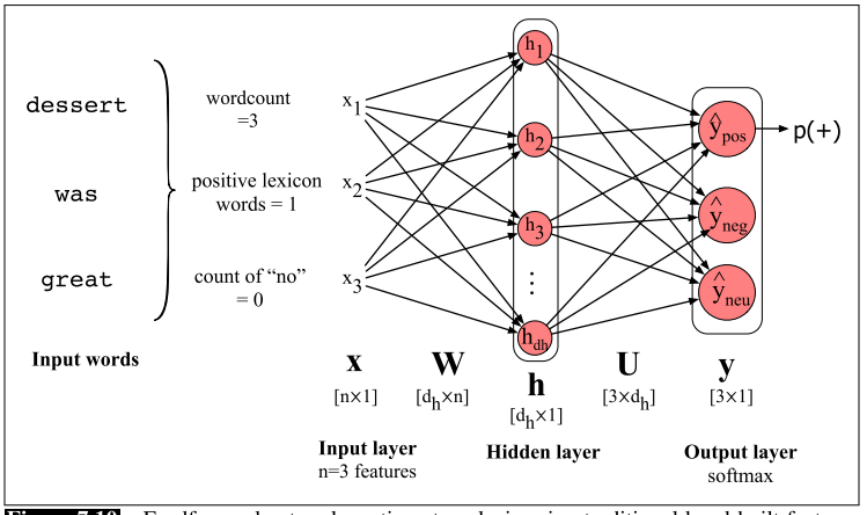


Figure 7.10 Feedforward network sentiment analysis using traditional hand-built features of the input text.

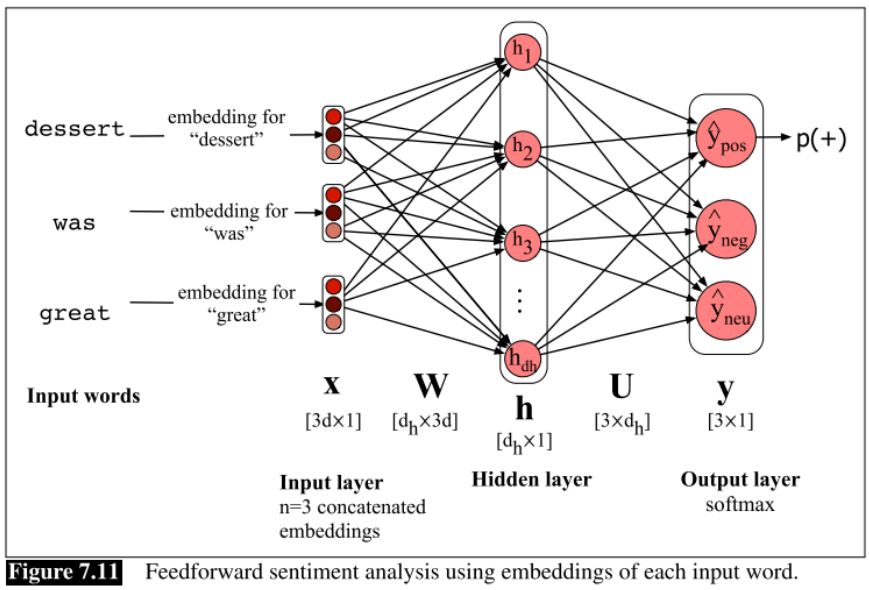


Figure 7.11 Feedforward sentiment analysis using embeddings of each input word.

A feedforward neural LM is a feedforward network that takes as input at time t a representation of some number of previous words (wt−1,wt−2, etc.) and outputs a probability distribution over possible next words. So similar to n-gram LM, the FFNLM approximates the probability of word given the entire prior context by approximating based on N previous words:

$$P(w_t|w_1,\ldots,w_{t-1}) \approx P(w_t|w_{t-N+1},\ldots,w_{t-1}) \quad (7.21)$$

NNLM represent words in this prior context by their **embeddings**, rather than just by their word identity as used on n-gram LM. Using embeddings allows neural language models to generalize better to unseen data.

E.g. Suppose we have a training sentence → "I have to make sure that the cat gets fed."but have never seen the words "gets fed" after the word "dog".

Our test set has the prefix "I forgot to make sure that the dog gets". What's the next word?

- An n-gram language model will predict "fed" after "that the cat gets", but not after "that the dog gets".
- But a neural LM, knowing that "cat" and "dog" have similar embeddings, will be able to generalize from the "cat" context to assign a high enough probability to "fed" even after seeing "dog".

Forward inference in the neural language model

forward inference or **decoding** for neural language models – is the task, given an input, of running a forward pass on the network to produce a probability distribution over possible output, in this case next words.

We first represent each of the N previous words as a one-hot vector of length |V|, i.e., with one dimension for each word in the vocabulary.

- A **one-hot vector** is a vector that has one element equal to 1—in the dimension corresponding to that word's index in the vocabulary— while all the other elements are set to zero.

The FFNLM has a moving window that can see N words into the past.

We'll let N-3, so the 3 words wt−1, wt−2, and wt−3 are each represented as a one-hot vector. We then multiply these one-hot vectors by the embedding matrix E. The embedding weight matrix E has a column for each word, each a column vector of d dimensions, and hence has dimensionality d×|V|. Multiplying by a one-hot vector that has only one non-zero element xi = 1 simply selects out the relevant column vector for word i, resulting in the embedding for word i, as shown in Fig. 7.12.

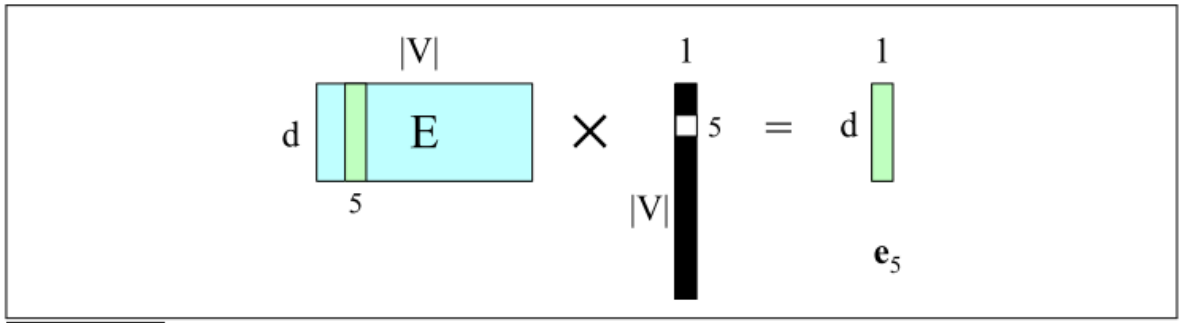


Figure 7.12 Selecting the embedding vector for word V_5 by multiplying the embedding matrix **E** with a one-hot vector with a 1 in index 5.

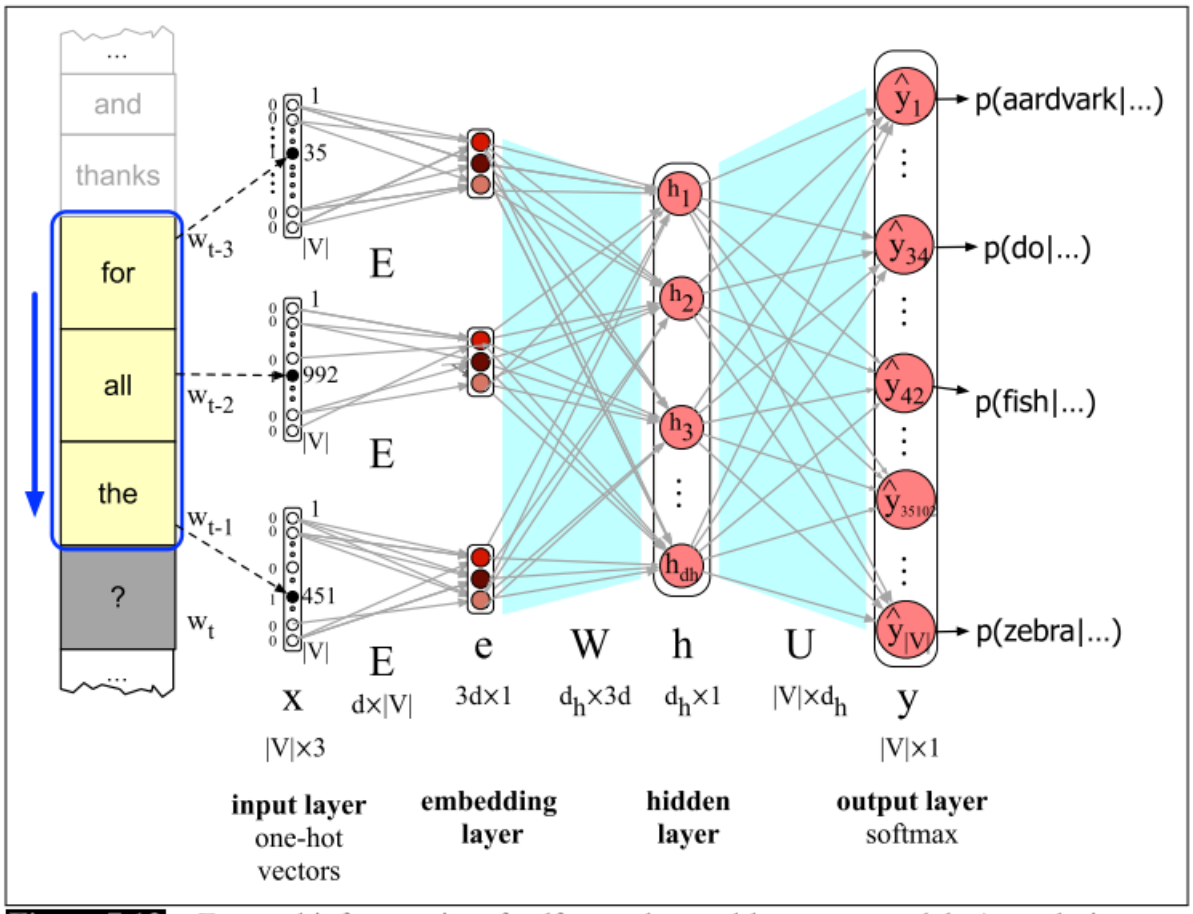


Figure 7.13 Forward inference in a feedforward neural language model. At each timestep t the network computes a d-dimensional embedding for each context word (by multiplying a one-hot vector by the embedding matrix **E**), and concatenates the 3 resulting embeddings to get the embedding layer **e**. The embedding vector **e** is multiplied by a weight matrix **W** and then an activation function is applied element-wise to produce the hidden layer **h**, which is then multiplied by another weight matrix **U**. Finally, a softmax output layer predicts at each node i the probability that the next word w_t will be vocabulary word V_i.

Algorithm for example shown above:

1. **Select three embeddings from E**: Given the three previous words, we look up their indices, create 3 one-hot vectors, and then multiply each by the embedding matrix **E**. Consider wt−3. The one-hot vector for 'for' (index 35) is multiplied by the embedding matrix E, to give the first part of the first hidden layer, the embedding layer. Since each column of the input matrix **E** is an embedding for a word, and the input is a one-hot column vector xi for word Vi, the embedding layer for input w will be $\mathbf{Ex}_i = \mathbf{e}_i$, the embedding for word i. We now concatenate the three embeddings for the three context words to produce the embedding layer **e**.
2. **Multiply by W**: We multiply by W (and add *b*) and pass through the ReLU (or other) activation function to get the hidden layer *h*.
3. **Multiply by U**: h is now multiplied by U.
4. **Apply softmax**: After the softmax, each node i in the output layer estimates the probability $P(w_{t=i}|w_{t-1},w_{t-2},w_{t-3})$

In summary, the eq for NNLM for window size = 3 given 1-hot input vectors for each input context word, are:

$$\begin{aligned}\mathbf{e} &= [\mathbf{E}\mathbf{x}_{t-3}; \mathbf{E}\mathbf{x}_{t-2}; \mathbf{E}\mathbf{x}_{t-1}] \\ \mathbf{h} &= \sigma(\mathbf{W}\mathbf{e} + \mathbf{b}) \\ \mathbf{z} &= \mathbf{U}\mathbf{h} \\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{z}) \end{aligned} \tag{7.22}$$

The embedding layer **e** is formed by concatenating the 3 embeddings for the three context vectors; we'll often use semicolons to mean concatenation of vectors.

Training Neural Nets

A FFNN is an instance of Supervised ML where we know the correct output for each obs x.

The goal of the training procedure is to learn parameters $\mathbf{W}_{[i]}$ and $\mathbf{b}_{[i]}$ for each layer *i* that make ˆ*y* for each training observation as close as possible to the true *y*.

- First, we'll need a **loss function** that models the distance between the system output and the gold output, and it's common to use the loss function used for logistic regression, the **cross-entropy loss**.
- Second, to find the parameters to minimize the loss functions, we will use the gradient descent optimization algo.
- Third, gradient descent requires knowing the gradient of the loss function, the vector that contains the partial derivative of the loss function with respect to each of the parameters.

- In logistic regression, for each observation we could directly compute the derivative of the loss function with respect to an individual w or b. But for neural networks, with millions of parameters in many layers, it's much harder to see how to compute the partial derivative of some weight in layer 1 when the loss is attached to some much later layer. How do we partial out the loss over all those intermediate layers?
  - The answer is the algorithm called error **backpropagation** or **backward differentiation**.

Loss function

The cross-entropy loss that is used in NN is the same one we saw for LR. When using NN is used as binary classifier, with sigmoid at final layer, the loss function is exactly same as LR—

$$L_{CE}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})] \tag{7.23}$$

But for multinomial classifier, let y be vector over C clases representing true output probability f(). The cross-entropy loss here is:

$$L_{CE}(\hat{y}, y) = -\sum_{i=1}^{C} y_i \log \hat{y}_i \tag{7.24}$$

Simplifying the eq, Assume this is a **hard classification** task, meaning that only one class is the correct one, and that there is one output unit in y for each class. If the true class is i, then y is a vector where yi = 1 and y j = 0 ∀ j ≠ i. A vector like this, with one value=1 and the rest 0, is called a one-hot vector. The terms in the sum in Eq. 7.24 will be 0 except for the term corresponding to the true class, i.e. →

$$\begin{aligned} L_{CE}(\hat{y},y) &= -\sum_{k=1}^{K} \mathbb{1}\{y=k\}\log \hat{y}_i \\ &= -\sum_{k=1}^{K} \mathbb{1}\{y=k\}\log \hat{p}(y=k|x) \\ &= -\sum_{k=1}^{K} \mathbb{1}\{y=k\}\log \frac{\exp(\mathbf{z}_k)}{\sum_{j=1}^{K}\exp(\mathbf{z}_j)} \end{aligned} \tag{7.25}$$

The cross-entropy loss is simply the log of the output probability corresponding to the correct class, and we therefore also call this the **negative log likelihood loss**:

$$L_{CE}(\hat{y},y) = -\log \hat{y}_i, \quad \text{(where } i \text{ is the correct class)} \tag{7.26}$$

Plugging in the softmax formula from Eq. 7.9, and with K the number of classes:

$$L_{CE}(\hat{y},y) = -\log \frac{\exp(\mathbf{z}_i)}{\sum_{j=1}^{K}\exp(\mathbf{z}_j)} \quad \text{(where } i \text{ is the correct class)} \tag{7.27}$$

Computing the Gradient

Requires partial derivative of the loss function with respect to each parameter. For a network with one weight layer and sigmoid output (which is what logistic regression is), we could simply use the derivative of the loss that we used for logistic regression in Eq. 7.28.

$$\begin{aligned} \frac{\partial L_{CE}(w,b)}{\partial w_j} &= (\hat{y}-y)\,\mathbf{x}_j \\ &= (\sigma(\mathbf{w}\cdot\mathbf{x}+b)-y)\,\mathbf{x}_j \end{aligned} \tag{7.28}$$

for a network with one hidden layer and softmax output, we could use the derivative of the softmax loss:

$$\begin{aligned} \frac{\partial L_{CE}}{\partial w_k} &= -(\mathbb{1}\{y=k\}-p(y=k|x))\mathbf{x}_k \\ &= -\left(\mathbb{1}\{y=k\}-\frac{\exp(\mathbf{w}_k\cdot\mathbf{x}+\mathbf{b}_k)}{\sum_{j=1}^{K}\exp(\mathbf{w}_j\cdot\mathbf{x}+\mathbf{b}_j)}\right)\mathbf{x}_k \end{aligned} \tag{7.29}$$

But these derivatives only give correct updates for one weight layer: the last one!

For deep networks, computing the gradient for each weight is much more complex, since we are computing the derivative wrt weight parameters that appear all the way back in the early layers of the network, even though the loss is computed at the very end of n/w.

Solution- **Error Backpropagation** or **Backprop**.

> While backprop was invented specially for neural networks, it turns out to be the same as a more general procedure called **backward differentiation**, which depends on the notion of computation graphs.

Computational Graphs:

Representation of the process of computing a mathematical expression, in which the computation is broken down into separate operations, each of which is modelled as a node in a graph.

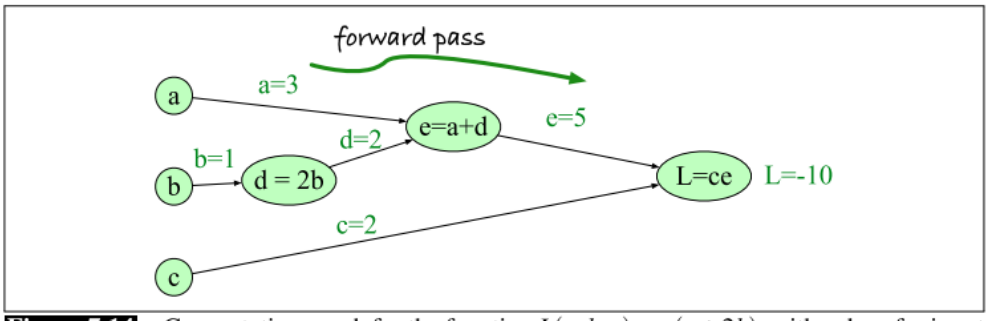


Figure 7.14 Computation graph for the function $L(a,b,c) = c(a+2b)$, with values for input nodes $a = 3$, $b = 1$, $c = -2$, showing the forward pass computation of L.

Consider the f() -- L(a,b,c) = c(a+2b) this can also be broken as (adding extra states – d & e)→

$$d = 2 * b$$
$$e = a + d$$
$$L = c * e$$

Backward differentiation on computation graphs (CG)

The importance of the CG comes from the backward pass, which is used to compute the derivatives that we'll need for the weight update.
Goal – compute derivative of output function L, wrt each input variables $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$.

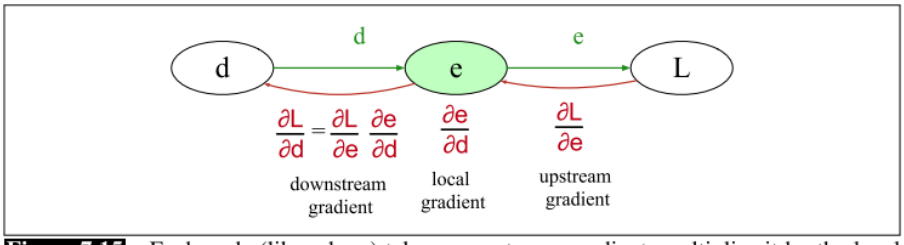


Figure 7.15 Each node (like e here) takes an upstream gradient, multiplies it by the local gradient (the gradient of its output with respect to its input), and uses the chain rule to compute a downstream gradient to be passed on to a prior node. A node may have multiple local gradients if it has multiple inputs.

Backwards differentiation makes use of the **chain rule** in calculus.

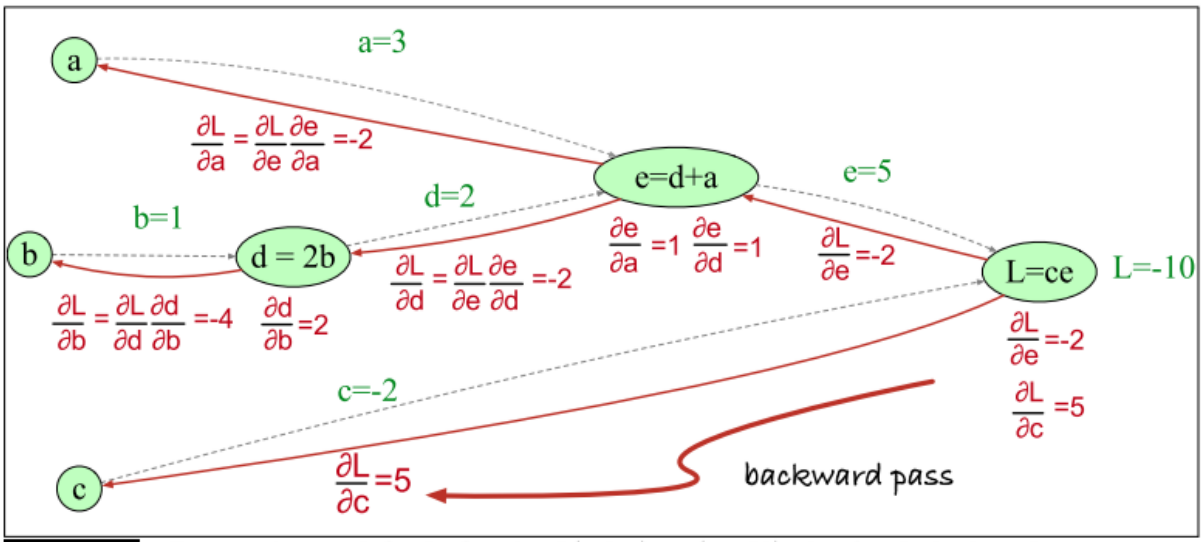


Figure 7.16 Computation graph for the function $L(a,b,c) = c(a+2b)$, showing the backward pass computation of $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$.

Backward differentiation for a neural network

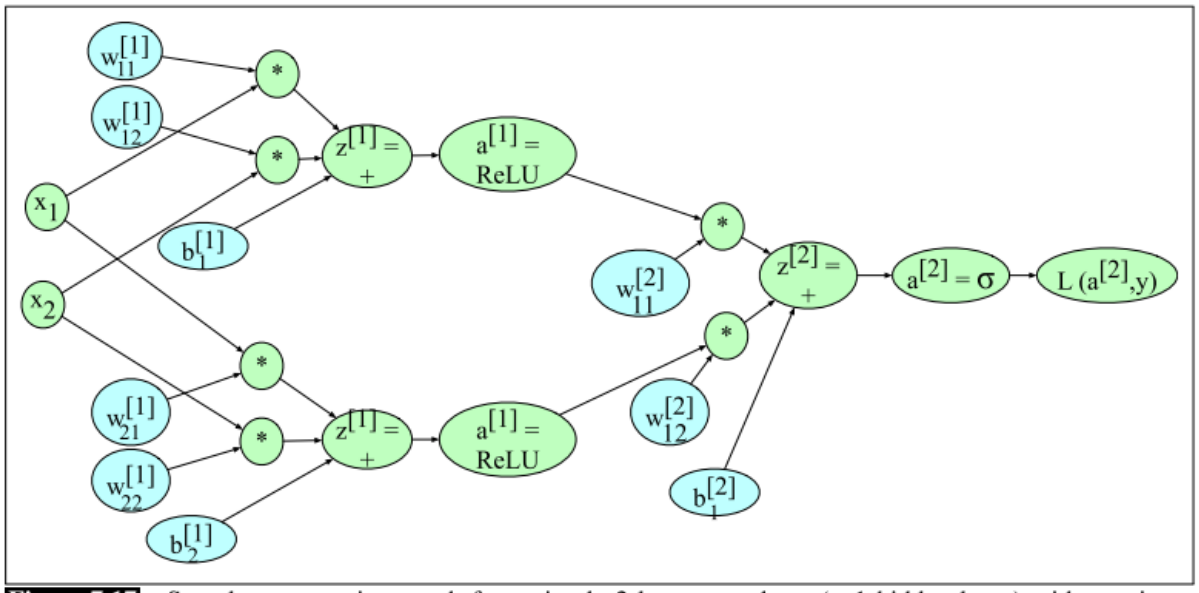


Figure 7.17 Sample computation graph for a simple 2-layer neural net (= 1 hidden layer) with two input dimensions and 2 hidden dimensions.

The weights that need updating (those for which we need to know the partial derivative of the loss function) are shown in teal. In order to do the backward pass, we'll need to know the derivatives of all the functions in the graph.

More details on learning

Optimization in neural networks is a non-convex optimization problem, more complex than for logistic regression, and for that and other reasons there are many best practices for successful learning.

For logistic regression we can initialize gradient descent with all the weights and biases having the value 0. In neural networks, by contrast, we need to initialize the weights with small random numbers. It's also helpful to normalize the input values to have 0 mean and unit variance.

Regularizations used for overfitting:

- Dropout: randomly dropping some units and their connections from the network during training
- Hyperparameter tuning: Hyperparameters of NN (W,b) are learnt by Gradient Descent. The hyperparameters are things that are chosen by the algorithm designer; optimal values are tuned on a devset rather than by gradient descent learning on the training set.
    - Hyperparameters:
        - Learning rate η,
        - the mini-batch size,
        - the model architecture (the number of layers, the number of hidden nodes per layer, the choice of activation functions),
        - how to regularize, and so on.

[*Gradient descent itself also has many architectural variants such as Adam*]

Sandeep Kumar Kushwaha

Training the neural language model

let's talk about the architecture for training a neural language model, setting the parameters θ = **E,W,U,b**.

For some tasks, it's ok to **freeze** the embedding layer E with initial word2vec values.

- Freezing means we use word2vec or some other pretraining algorithm to compute the initial embedding matrix **E**, and then hold it constant while we only modify **W, U**, and **b**, i.e., we don't update E during language model training.

However, often we'd like to learn the embeddings simultaneously with training the network. This is useful when the task the network is designed for (sentiment classification, or translation, or parsing) places strong constraints on what makes a good representation for words.

Let's train the entire model including E, i.e. to set all the parameters **θ = E,W,U,b**. We do this using GD, using BP on computational graph to compute the gradient.

Training thus not only sets the weights **W** and **U** of the network, but also as we're predicting upcoming words, we're learning the embeddings **E** for each word that best predict upcoming words.

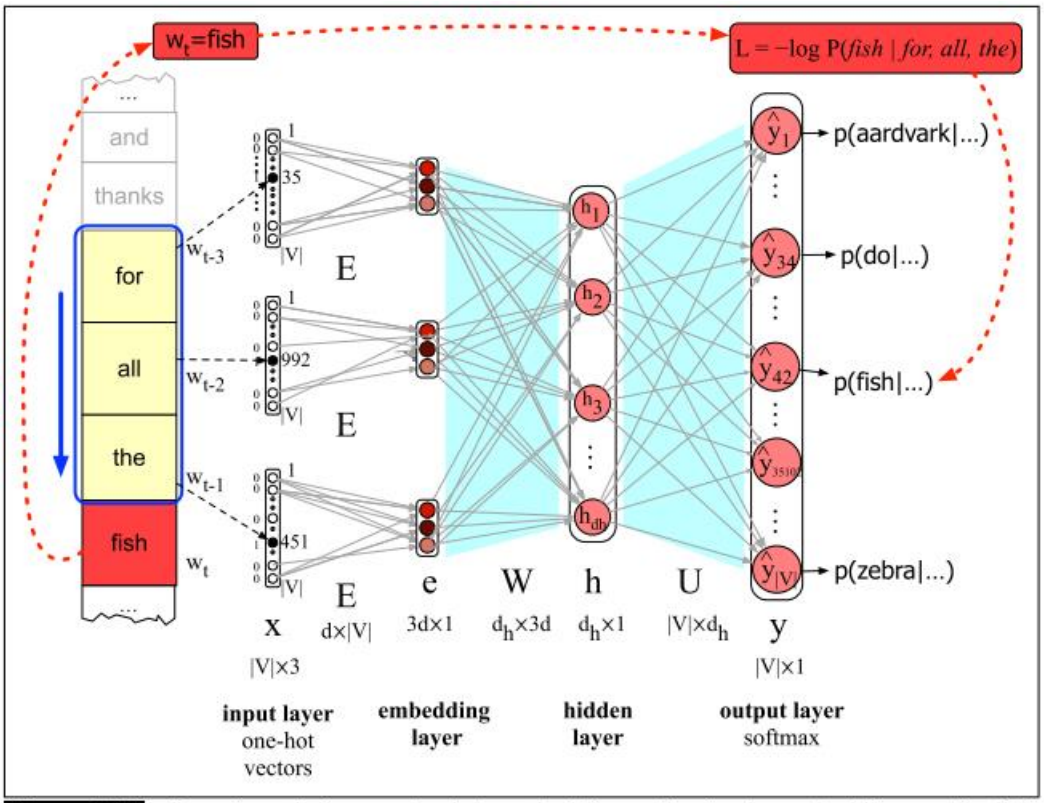


Figure 7.18 Learning all the way back to embeddings. Again, the embedding matrix *E* is shared among the 3 context words.

Fig 7.18 – window size N = 3 context words.

Input x consists of 3 one-hot vectors, fully connected to embedding layer via 3 instantiations of embedding matrix E.

We don't want to learn separate weight matrices for mapping each of the 3 previous words to the projection layer. We want one single embedding dictionary E that's shared among these three. That's because over time, many different words will appear as wt−2 or wt−1, and we'd like to just represent each word with one vector, whichever context position it appears in. Recall that the embedding weight matrix E has a column for each word, each a column vector of d dimensions, and hence has dimensionality d×|V|.

Generally training proceeds by taking as input a very long text, concatenating all the sentences, starting with random weights, and then iteratively moving through the text predicting each word wt . At each word wt , we use the cross-entropy (negative log likelihood) loss.

$$L_{CE}(\hat{y}, y) = -\log \hat{y}_i, \quad \text{(where } i \text{ is the correct class)} \tag{7.43}$$

For language modeling, the classes are the words in the vocabulary, so ˆyi here means the probability that the model assigns to the correct next word wt :

$$L_{\text{CE}} = -\log p(w_t | w_{t-1}, \ldots, w_{t-n+1}) \tag{7.44}$$

The parameter update for stochastic gradient descent for this loss from step s to s+1 is then:

$$\theta^{s+1} = \theta^s - \eta \frac{\partial \left[-\log p(w_t | w_{t-1}, \ldots, w_{t-n+1}) \right]}{\partial \theta} \tag{7.45}$$

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

Training the parameters to minimize loss will result both in an algorithm for language modeling (a word predictor) but also a new set of embeddings E that can be used as word representations for other tasks.

Summary:

- Neural networks are built out of **neural units**, originally inspired by human neurons but now simply an abstract computational device.
- Each neural unit multiplies input values by a weight vector, adds a bias, and then applies a non-linear activation function like sigmoid, tanh, or rectified linear unit.
- In a **fully-connected**, **feedforward** network, each unit in layer i is connected to each unit in layer $i+1$, and there are no cycles.
- The power of neural networks comes from the ability of early layers to learn representations that can be utilized by later layers in the network.
- Neural networks are trained by optimization algorithms like **gradient descent**.
- **Error backpropagation**, backward differentiation on a **computation graph**, is used to compute the gradients of the loss function for a network.
- **Neural language models** use a neural network as a probabilistic classifier, to compute the probability of the next word given the previous n words.
- Neural language models can use pretrained **embeddings**, or can learn embeddings from scratch in the process of language modeling.

8. Sequence Labelling for Parts of Speech and Names Entities

Eight **parts of speech**: noun, verb, pronoun, preposition, adverb, conjunction, participle, and article.

POS and named entities are useful clues to sentence structure and meaning. Knowing whether a word is a noun or a verb tells us about likely neighbouring words (nouns in English are preceded by determiners and adjectives, verbs by nouns) and syntactic structure (verbs have dependency links to nouns), making part-of-speech tagging a key aspect of parsing.

POS tagging - taking a sequence of words and assigning each word a part of speech like NOUN or VERB,

named entity recognition (NER) - assigning words or phrases tags like PERSON, LOCATION, or ORGANIZATION.

sequence labeling tasks - we assign, to each word x_i in an input word sequence, a label y_i, so that the output sequence Y has the same length as the input sequence X.

classic sequence labeling algorithms,

- one generative— the **Hidden Markov Model (HMM)**
- one discriminative— the **Conditional Random Field (CRF)**

(Mostly) English Word Classes

	Tag	Description	Example
Open Class	**ADJ**	Adjective: noun modifiers describing properties	*red, young, awesome*
	ADV	Adverb: verb modifiers of time, place, manner	*very, slowly, home, yesterday*
	NOUN	words for persons, places, things, etc.	*algorithm, cat, mango, beauty*
	VERB	words for actions and processes	*draw, provide, go*
	PROPN	Proper noun: name of a person, organization, place, etc..	*Regina, IBM, Colorado*
	INTJ	Interjection: exclamation, greeting, yes/no response, etc.	*oh, um, yes, hello*
Closed Class Words	**ADP**	Adposition (Preposition/Postposition): marks a noun's spacial, temporal, or other relation	*in, on, by under*
	AUX	Auxiliary: helping verb marking tense, aspect, mood, etc.,	*can, may, should, are*
	CCONJ	Coordinating Conjunction: joins two phrases/clauses	*and, or, but*
	DET	Determiner: marks noun phrase properties	*a, an, the, this*
	NUM	Numeral	*one, two, first, second*
	PART	Particle: a preposition-like form used together with a verb	*up, down, on, off, in, out, at, by*
	PRON	Pronoun: a shorthand for referring to an entity or event	*she, who, I, others*
	SCONJ	Subordinating Conjunction: joins a main clause with a subordinate clause such as a sentential complement	*that, which*
Other	**PUNCT**	Punctuation	; , ()
	SYM	Symbols like $ or emoji	$, %
	X	Other	asdf, qwfg

Figure 8.1 The 17 parts of speech in the Universal Dependencies tagset (Nivre et al., 2016a). Features can be added to make finer-grained distinctions (with properties like number, case, definiteness, and so on).

Closed classes are those with relatively fixed membership, such as prepositions— new prepositions are rarely coined.

- Closed class words are generally function words like *of, it, and*, or *you*, which tend to be very short, occur frequently, and often have structuring uses in grammar.

By contrast, nouns and verbs are **open classes**— new nouns and verbs like *iPhone* or to *fax* are continually being created or borrowed.

- Four major open classes occur in the languages of the world: nouns (including proper nouns), verbs, adjectives, and adverbs, as well as the smaller open class of interjections.

Nouns are words for people, places, or things, but include others as well.

- Common nouns include concrete terms like *cat* and *mango*, abstractions like *algorithm* and *beauty.*
- Nouns in English can occur with determiners (*a goat, its bandwidth*)
- Count nouns – *goat*
- Mass nouns – *goats*
- Proper nouns, like *Regina*, *Colorado*, and *IBM*, are names of specific persons or entities.

Verb: refer to actions and processes, including main verbs like *draw*, *provide* and *go.*

- They have inflections (non-third-person-singular (*eat*), third-person-singular (*eats*), progressive (*eating*), past participle (*eaten*))

Adjectives often describe properties or qualities of nouns, like color (*white, black*), age (*old, young*), and value (*good, bad),* but there are languages without adjectives.

- In Korean, for example, the words corresponding to English adjectives act as a subclass of verbs, so what is in English an adjective "beautiful" acts in Korean like a verb meaning "to be beautiful".

Adverbs are a hodge-podge. All the italicized words in this example are adverbs:

- *Actually*, I ran *home extremely quickly yesterday.*
- Adverbs generally modify something (often verbs, hence the name "adverb", but also other adverbs and entire verb phrases).
- **Directional adverbs** or **locative adverbs** (home, here, downhill) specify the direction or location ofsome action;
- **Degree adverbs** (extremely, very, somewhat) specify the extent of some action, process, or property;
- **manner adverbs** (slowly, slinkily, delicately) describe the manner of some action or process; and
- **temporal adverbs** describe the time that some action or event took place (yesterday, Monday).

Interjections (oh, hey, alas, uh, um), are a smaller open class, that also includes greetings (hello, goodbye), and question responses (yes, no, uh-huh).

English adpositions occur before nouns, hence, are called **prepositions**. They can indicate spatial or temporal relations, whether literal (on it, before then, by the house) or metaphorical (on time, with gusto, beside herself), and relations like marking the agent in Hamlet was written by Shakespeare.

Particle resembles a preposition or an adverb and is used in combination with a verb. Particles often have extended meanings that aren't quite the same as the prepositions they resemble, as in the particle *over* in *she turned the paper over*.

Phrasal verb – Verb and particle acting together. The meaning of phrasal verbs is often **non-compositional**—not predictable from the individual meanings of the verb and the particle. Thus, *turn down* means 'reject', *rule out* 'eliminate', and *go on* 'continue'.

Determiners like *this* and *that* can mark the start of an English noun phrase.

- Articles like *a, an,* and *the*, are a type of determiner that mark discourse properties of the noun and are quite frequent; *the* is the most common word in written English, with *a* and *an* right behind.

Conjunctions join two phrases, clauses, or sentences. Coordinating conjunctions like *and*, *or*, and *but* join two elements of equal status. Subordinating conjunctions are used when one of the elements has some embedded status. For example, the subordinating conjunction *that* in *"I thought that you might like some milk"* links the main clause *I thought* with the subordinate clause *you might like some milk*. This clause is called subordinate because this entire clause is the "content" of the main verb *thought*. Subordinating conjunctions like *that* which link a verb to its argument in this way are also called **complementizers**.

Pronouns act as a shorthand for referring to an entity or event. Personal **pronouns** refer to persons or entities (*you*, *she*, *I*, *it*, *me*, etc.). Possessive **pronouns** are forms of personal pronouns that indicate either actual possession or more often just an abstract relation between the person and some object (*my, your, his, her, its, one's, our, their*). **Wh-pronouns** (*what, who, whom, whoever*) are used in certain question forms, or act as complementizers (*Frida, who married Diego...*).

Auxiliary verbs mark semantic features of a main verb such as its tense, whether it is completed (aspect), whether it is negated (polarity), and whether an action is necessary, possible, suggested, or desired (mood). English auxiliaries include the **copula** verb *be*, the two verbs *do* and *have*, forms, as well as **modal verbs** used to mark the mood associated with the event depicted by the main verb: *can* indicates ability or possibility, *may* permission or possibility, *must* necessity.

Part-of-Speech Tagging

Tagging is a **disambiguation task**; words are **ambiguous** —have more than one possible part-of-speech—and the goal is to find the correct tag for the situation.

E.g., "hand me the *book*" & "let's *book* the flight".

The goal of POS-tagging is to resolve these ambiguities, choosing the proper tag for the context.

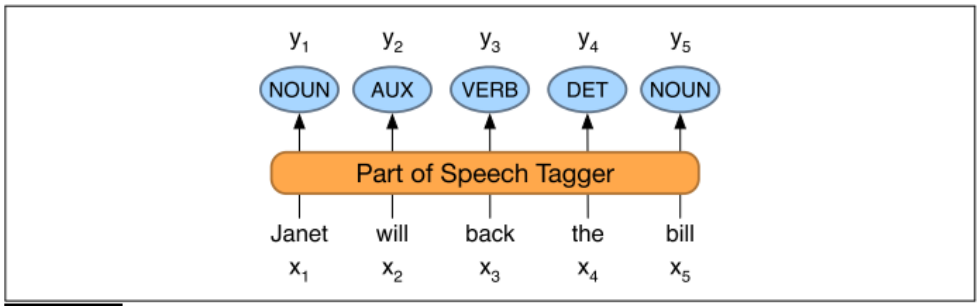


Figure 8.3 The task of part-of-speech tagging: mapping from input words $x_1, x_2, \ldots, x_n$ to output POS tags $y_1, y_2, \ldots, y_n$.

The accuracy of POS tagging algorithms (the % of test set tags that match human gold labels) is extremely high ~ 97%.

Types:		WSJ	Brown
Unambiguous	(1 tag)	44,432 (86%)	45,799 (85%)
Ambiguous	(2+ tags)	7,025 (14%)	8,050 (15%)
Tokens:			
Unambiguous	(1 tag)	577,421 (45%)	384,349 (33%)
Ambiguous	(2+ tags)	711,780 (55%)	786,646 (67%)

Figure 8.4 Tag ambiguity in the Brown and WSJ corpora (Treebank-3 45-tag tagset).

earnings growth took a **back/JJ** seat
a small building in the **back/NN**
a clear majority of senators **back/VBP** the bill
Dave began to **back/VB** toward the door
enable the country to buy **back/RP** debt
I was twenty-one **back/RB** then

Nonetheless, many words are easy to disambiguate, because their different tags aren't equally likely. For example, *a* can be a determiner or the letter *a*, but the determiner sense is much more likely.

This concept works as a baseline: :

Most Frequent Class Baseline: Always compare a classifier against a baseline at least as good as the most frequent class baseline (assigning each token to the class it occurred in most often in the training set).

Named Entities and Named Entity Tagging

A **named entity** is, roughly speaking, anything that can be referred to with a proper name: a person, a location, an organization.

The task of **named entity recognition (NER)** is to find spans of text that constitute proper names and tag the type of the entity. Four entity tags are most common:

- PER (person),
- LOC (location),
- ORG (organization),
- GPE (geo-political entity).

Example:

Citing high fuel prices, [ORG **United Airlines**] said [TIME **Friday**] it has increased fares by [MONEY **$6**] per round trip on flights to some cities also served by lower-cost carriers. [ORG **American Airlines**], a unit of [ORG **AMR Corp.**], immediately matched the move, spokesman [PER **Tim Wagner**] said. [ORG **United**], a unit of [ORG **UAL Corp.**], said the increase took effect [TIME **Thursday**] and applies to most routes where it competes against discount carriers, such as [LOC **Chicago**] to [LOC **Dallas**] and [LOC **Denver**] to [LOC **San Francisco**].

Type	Tag	Sample Categories	Example sentences
People	PER	people, characters	**Turing** is a giant of computer science.
Organization	ORG	companies, sports teams	The **IPCC** warned about the cyclone.
Location	LOC	regions, mountains, seas	**Mt. Sanitas** is in **Sunshine Canyon**.
Geo-Political Entity	GPE	countries, states	**Palo Alto** is raising the fees for parking.

Figure 8.5 A list of generic named entity types with the kinds of entities they refer to.

Name Entity Tagging is a useful first step in lots of NLP tasks.

- In sentiment analysis we might want to know a consumer's sentiment toward a particular entity.
- Entities are a useful first stage in question answering, or for linking text to information in structured knowledge sources like Wikipedia.
- NET is also central to tasks involving building semantic representations, like extracting events and the relationship between participants.

Unlike part-of-speech tagging, where there is no segmentation problem since each word gets one tag, the task of NER is to find and label spans of text, and is difficult partly because of the ambiguity of segmentation; we need to decide what's an entity and what isn't, and where the boundaries are.

Another difficulty is caused by type ambiguity. The mention JFK can refer to a person, the airport in New York, or any number of schools, bridges, and streets around the United States.

[PER Washington] was born into slavery on the farm of James Burroughs.
[ORG Washington] went up 2 games to 1 in the four-game series.
Blair arrived in [LOC Washington] for what may well be his last state visit.
In June, [GPE Washington] passed a primary seatbelt law.

Figure 8.6 Examples of type ambiguities in the use of the name *Washington*.

The standard approach to sequence labelling for a span-recognition problem like NER is **BIO tagging** (Ramshaw and Marcus, 1995). Treats NER like a word-by-word sequence labelling task, via tags that capture both the boundary and the NE type. Consider the following sentence:

[PER **Jane Villanueva**] of [ORG **United**] , a unit of [ORG **United Airlines Holding**] , said the fare applies to the [LOC **Chicago**] route.

In BIO tagging we label any token that begins a span of interest with the label B, tokens that occur inside a span are tagged with an I, and any tokens outside of any span of interest are labeled O. While there is only one O tag, we'll have distinct B and I tags for each named entity class. The number of tags is thus 2n+1 tags, where n is the number of entity types. BIO tagging can represent exactly the same information as the bracketed notation, but has the advantage that we can represent the task in the same simple sequence modeling way as part-of-speech tagging: assigning a single label yi to each input word xi:

Words	IO Label	BIO Label	BIOES Label
Jane	I-PER	B-PER	B-PER
Villanueva	I-PER	I-PER	E-PER
of	O	O	O
United	I-ORG	B-ORG	B-ORG
Airlines	I-ORG	I-ORG	I-ORG
Holding	I-ORG	I-ORG	E-ORG
discussed	O	O	O
the	O	O	O
Chicago	I-LOC	B-LOC	S-LOC
route	O	O	O
.	O	O	O

Figure 8.7 NER as a sequence model, showing IO, BIO, and BIOES taggings.

two variant tagging schemes:

- IO tagging, which loses some information by eliminating the B tag,

- BIOES tagging, which adds an end tag E for the end of a span, and a span tag S for a span consisting of only one word.

A sequence labeler (HMM, CRF, RNN, Transformer, etc.) is trained to label each token in a text with tags that indicate the presence (or absence) of particular kinds of named entities.

HMM Part-of-Speech Tagging

- sequence labelling algorithm, the Hidden Markov Model.
- probabilistic sequence model: given a sequence of units (words, letters, morphemes, sentences, whatever), it computes a probability distribution over possible sequences of labels and chooses the best label sequence.

Markov Chains

HMM is based on augmenting the Markov chain.

A **Markov chain** is a model that tells us something about the probabilities of sequences of random variables, states, each of which can take on values from some set. These sets can be words, or tags, or symbols representing anything.

A Markov chain makes a very strong assumption that if we want to predict the future in the sequence, all that matters is the current state.

- All the states before the current state have no impact on the future except via the current state. It's as if to predict tomorrow's weather you could examine today's weather but you weren't allowed to look at yesterday's weather.

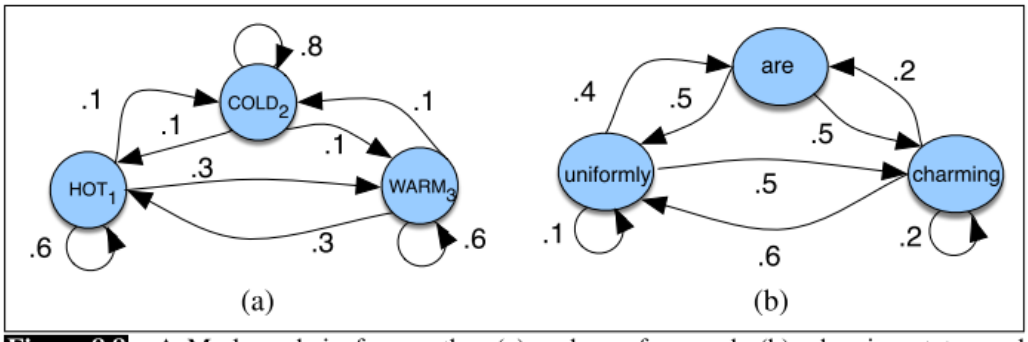


Figure 8.8 A Markov chain for weather (a) and one for words (b), showing states and transitions. A start distribution π is required; setting $\pi = [0.1, 0.7, 0.2]$ for (a) would mean a probability 0.7 of starting in state 2 (cold), probability 0.1 of starting in state 1 (hot), etc.

- **Markov Assumption:** $P(q_i = a|q_1 \ldots q_{i-1}) = P(q_i = a|q_{i-1})$ (8.3)
- Sum of arcs leaving the state = 1.

Formally, a Markov chain is specified by the following components:

$Q = q_1 q_2 \ldots q_N$	a set of N **states**
$A = a_{11} a_{12} \ldots a_{N1} \ldots a_{NN}$	a **transition probability matrix** A, each a_{ij} representing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{n} a_{ij} = 1 \quad \forall i$
$\pi = \pi_1, \pi_2, \ldots, \pi_N$	an **initial probability distribution** over states. π_i is the probability that the Markov chain will start in state i. Some states j may have $\pi_j = 0$, meaning that they cannot be initial states. Also, $\sum_{i=1}^{n} \pi_i = 1$

-

The Hidden Markov Model

- What if the events we are interested are hidden?
- For example we don't normally observe part-of-speech tags in a text. Rather, we see words, and must infer the tags from the word sequence.
- We call the tags hidden because they are not observed.
- HMM) allows us to talk about both *observed* events (like words that we see in the input) and *hidden* events (like part-of-speech tags) that we think of as causal factors in our probabilistic model.
- An HMM is specified by the following components:

$Q = q_1 q_2 \ldots q_N$	a set of N **states**
$A = a_{11} \ldots a_{ij} \ldots a_{NN}$	a **transition probability matrix** A, each a_{ij} representing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{N} a_{ij} = 1 \quad \forall i$
$O = o_1 o_2 \ldots o_T$	a sequence of T **observations**, each one drawn from a vocabulary $V = v_1, v_2, ..., v_V$
$B = b_i(o_t)$	a sequence of **observation likelihoods**, also called **emission probabilities**, each expressing the probability of an observation o_t being generated from a state q_i
$\pi = \pi_1, \pi_2, ..., \pi_N$	an **initial probability distribution** over states. π_i is the probability that the Markov chain will start in state i. Some states j may have $\pi_j = 0$, meaning that they cannot be initial states. Also, $\sum_{i=1}^{n} \pi_i = 1$

-
- HMM has 2 simplifying assumptions:
  - 1st - a first-order Markov chain, the probability of a particular state depends only on the previous state:

    Markov Assumption: $P(q_i|q_1, ..., q_{i-1}) = P(q_i|q_{i-1})$ (8.6)
  - 2nd - probability of an output observation oi depends only on the state that produced the observation qi and not on any other states or any other observations:

    Output Independence: $P(o_i|q_1, \ldots q_i, \ldots, q_T, o_1, \ldots, o_i, \ldots, o_T) = P(o_i|q_i)$ (8

HMM tagging as decoding

For any model, such as an HMM, that contains hidden variables, the task of determining the hidden variables sequence corresponding to the sequence of observations is called **decoding**.

> **Decoding**: Given as input an HMM $\lambda = (A, B)$ and a sequence of observations $O = o_1, o_2, ..., o_T$, find the most probable sequence of states $Q = q_1 q_2 q_3 \ldots q_T$.

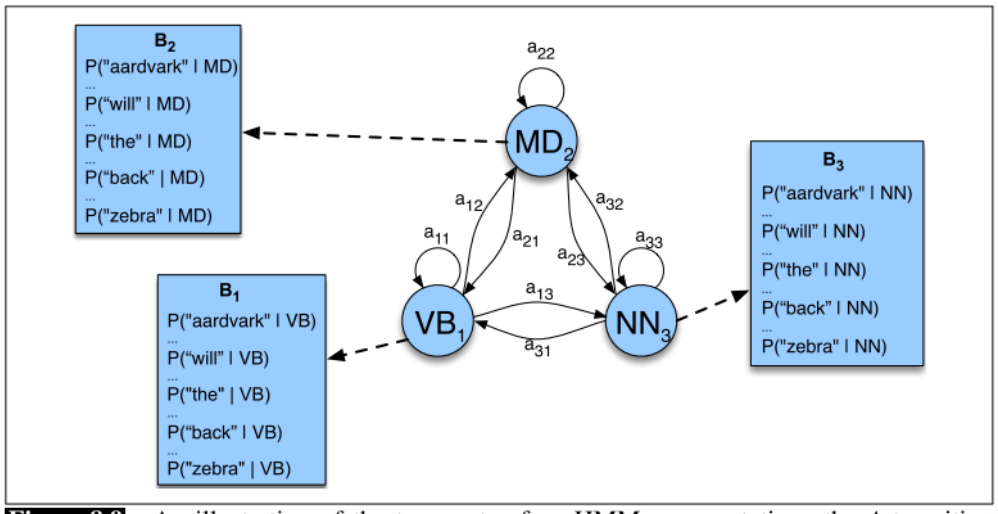


Figure 8.9 An illustration of the two parts of an HMM representation: the A transition probabilities used to compute the prior probability, and the B observation likelihoods that are associated with each state, one likelihood for each possible observation word.

For part-of-speech tagging, the goal of HMM decoding is to choose the tag sequence $t_1 \dots t_n$ that is most probable given the observation sequence of n words $w_1 \dots w_n$:

$$\hat{t}_{1:n} = \underset{t_1 \dots t_n}{\operatorname{argmax}} P(t_1 \dots t_n | w_1 \dots w_n) \tag{8.12}$$

The way we'll do this in the HMM is to use Bayes' rule to instead compute:

$$\hat{t}_{1:n} = \underset{t_1 \dots t_n}{\operatorname{argmax}} \frac{P(w_1 \dots w_n | t_1 \dots t_n) P(t_1 \dots t_n)}{P(w_1 \dots w_n)} \tag{8.13}$$

Furthermore, we simplify Eq. 8.13 by dropping the denominator $P(w_1^n)$:

$$\hat{t}_{1:n} = \underset{t_1 \dots t_n}{\operatorname{argmax}} P(w_1 \dots w_n | t_1 \dots t_n) P(t_1 \dots t_n) \tag{8.14}$$

- HMM taggers make two further simplifying assumptions.
  - first is that the probability of a word appearing depends only on its own tag and is independent of neighboring words and tags:

$$P(w_1 \dots w_n | t_1 \dots t_n) \approx \prod_{i=1}^{n} P(w_i | t_i) \tag{8.15}$$

  - Second assumption, the bigram assumption, is that the probability of a tag is dependent only on the previous tag, rather than the entire tag sequence:

$$P(t_1 \dots t_n) \approx \prod_{i=1}^{n} P(t_i | t_{i-1}) \tag{8.16}$$

$$\hat{t}_{1:n} = \underset{t_1 \dots t_n}{\operatorname{argmax}} P(t_1 \dots t_n | w_1 \dots w_n) \approx \underset{t_1 \dots t_n}{\operatorname{argmax}} \prod_{i=1}^{n} \overbrace{P(w_i | t_i)}^{\text{emission}} \overbrace{P(t_i | t_{i-1})}^{\text{transition}} \tag{8.17}$$

The two parts of Eq. 8.17 correspond neatly to the B **emission probability** and A **transition probability** that we just defined above!

Viterbi Algorithm

- decoding algorithm for HMM is Viterbi Algo.
- instance of **dynamic programming**, Viterbi resembles the dynamic programming **minimum edit distance.**

function VITERBI(*observations* of len T,*state-graph* of len N) **returns** *best-path*, *path-prob*

create a path probability matrix *viterbi*$[N,T]$
for each state s **from** 1 **to** N **do** ; initialization step
  $viterbi[s,1] \leftarrow \pi_s * b_s(o_1)$
  $backpointer[s,1] \leftarrow 0$
for each time step t **from** 2 **to** T **do** ; recursion step
  for each state s **from** 1 **to** N **do**
    $viterbi[s,t] \leftarrow \max_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
    $backpointer[s,t] \leftarrow \operatorname{argmax}_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
$bestpathprob \leftarrow \max_{s=1}^{N} viterbi[s,T]$; termination step
$bestpathpointer \leftarrow \operatorname{argmax}_{s=1}^{N} viterbi[s,T]$; termination step
$bestpath \leftarrow$ the path starting at state *bestpathpointer*, that follows backpointer[] to states back in time
return *bestpath*, *bestpathprob*

Figure 8.10 Viterbi algorithm for finding the optimal sequence of tags. Given an observation sequence and an HMM $\lambda = (A, B)$, the algorithm returns the state path through the HMM that assigns maximum likelihood to the observation sequence.

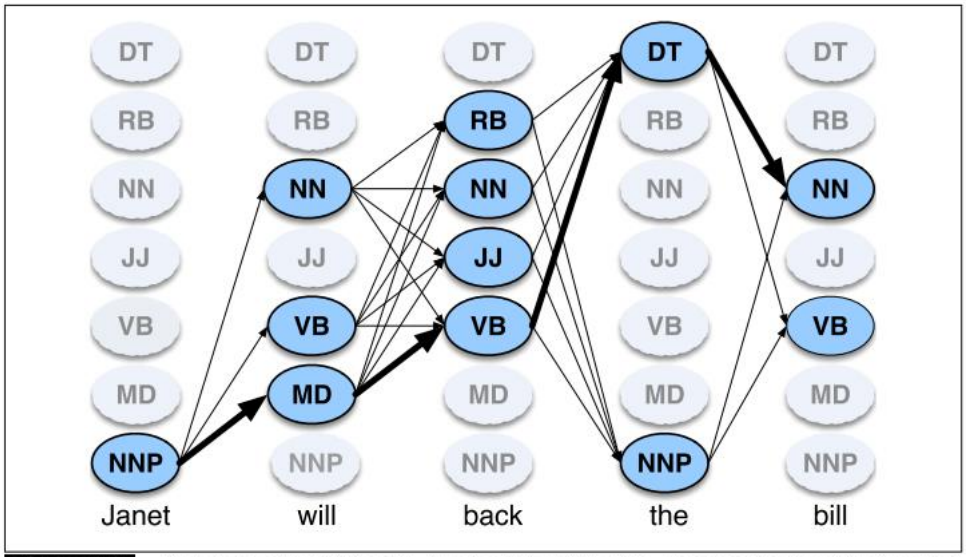


Figure 8.11 A sketch of the lattice for *Janet will back the bill*, showing the possible tags (q_i) for each word and highlighting the path corresponding to the correct tag sequence through the hidden states. States (parts of speech) which have a zero probability of generating a particular word according to the B matrix (such as the probability that a determiner DT will be realized as *Janet*) are greyed out.

$v_{t-1}(i)$	the **previous Viterbi path probability** from the previous time step
a_{ij}	the **transition probability** from previous state q_i to current state q_j
$b_j(o_t)$	the **state observation likelihood** of the observation symbol o_t given the current state j

Working through an example

Let's tag the sentence Janet will back the bill; the goal is the correct series of tags

(8.20) Janet/NNP will/MD back/VB the/DT bill/NN

	NNP	MD	VB	JJ	NN	RB	DT
<s>	0.2767	0.0006	0.0031	0.0453	0.0449	0.0510	0.2026
NNP	0.3777	0.0110	0.0009	0.0084	0.0584	0.0090	0.0025
MD	0.0008	0.0002	0.7968	0.0005	0.0008	0.1698	0.0041
VB	0.0322	0.0005	0.0050	0.0837	0.0615	0.0514	0.2231
JJ	0.0366	0.0004	0.0001	0.0733	0.4509	0.0036	0.0036
NN	0.0096	0.0176	0.0014	0.0086	0.1216	0.0177	0.0068
RB	0.0068	0.0102	0.1011	0.1012	0.0120	0.0728	0.0479
DT	0.1147	0.0021	0.0002	0.2157	0.4744	0.0102	0.0017

Figure 8.12 The A transition probabilities $P(t_i|t_{i-1})$ computed from the WSJ corpus without smoothing. Rows are labeled with the conditioning event; thus $P(VB|MD)$ is 0.7968.

	Janet	will	back	the	bill
NNP	0.000032	0	0	0.000048	0
MD	0	0.308431	0	0	0
VB	0	0.000028	0.000672	0	0.000028
JJ	0	0	0.000340	0	0
NN	0	0.000200	0.000223	0	0.002337
RB	0	0	0.010446	0	0
DT	0	0	0	0.506099	0

Figure 8.13 Observation likelihoods B computed from the WSJ corpus without smoothing, simplified slightly.

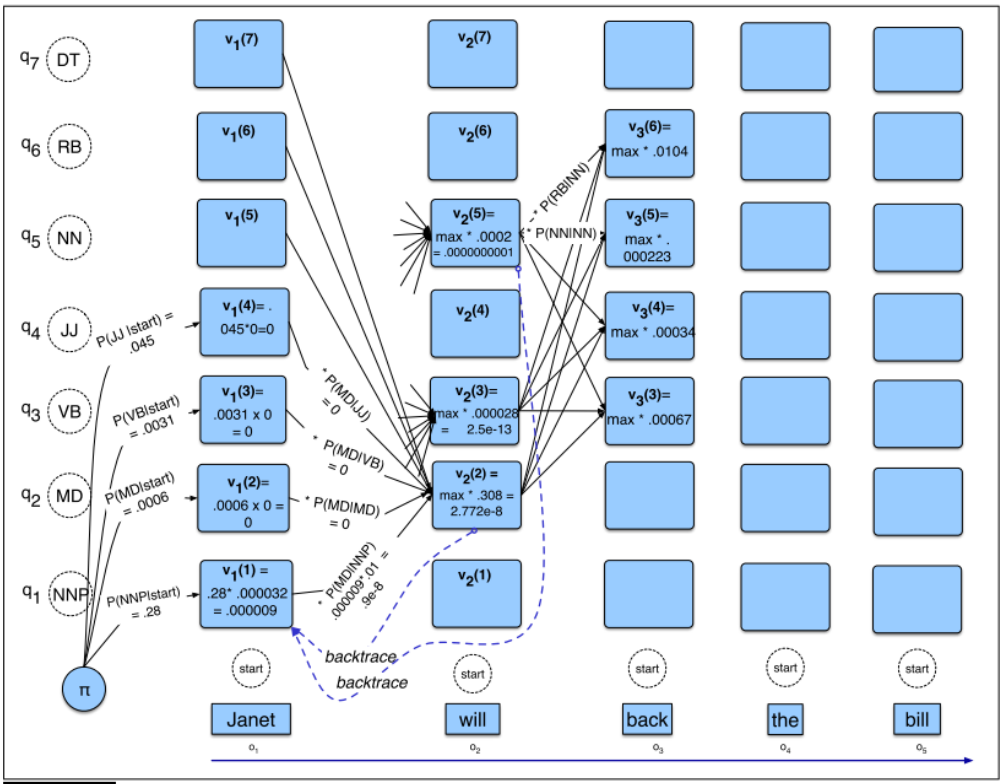


Figure 8.14 The first few entries in the individual state columns for the Viterbi algorithm. Each cell keeps the probability of the best path so far and a pointer to the previous cell along that path. We have only filled out columns 1 and 2; to avoid clutter most cells with value 0 are left empty. The rest is left as an exercise for the reader. After the cells are filled in, backtracing from the *end* state, we should be able to reconstruct the correct state sequence NNP MD VB DT NN.

Conditional Random Fields (CRFs)

Challenges with HMM:

- HMMs need a number of augmentations to achieve high accuracy. For example, in POS tagging as in other tasks, we often run into unknown words: proper names and acronyms are created very often, and even new common nouns and verbs enter the language at a surprising rate.

It would be great to have ways to add arbitrary features to help with this, perhaps based on capitalization or morphology. (words starting with capital letters are likely to be proper nouns, words ending with -ed tend to be past tense (VBD or VBN), etc.) Or knowing the previous or following words might be a useful feature (if the previous word is the, the current tag is unlikely to be a verb).

There is a discriminative sequence model based on log-linear models: **CRF**.

Linear chain CRF, the version of the CRF most commonly used for language processing, and the one whose conditioning closely matches the HMM.

Assuming we have a sequence of input words X = x1...xn and want to compute a sequence of output tags Y = y1...yn.

- In an HMM to compute the best tag sequence that maximizes P(Y|X) we rely on Bayes' rule and the likelihood P(X|Y):

$$\begin{aligned}\hat{Y} &= \operatorname*{argmax}_Y p(Y|X) \\ &= \operatorname*{argmax}_Y p(X|Y)p(Y) \\ &= \operatorname*{argmax}_Y \prod_i p(x_i|y_i) \prod_i p(y_i|y_{i-1}) \qquad (8.21)\end{aligned}$$

- In a CRF, by contrast, we compute the posterior p(Y|X) directly, training the CRF to discriminate among the possible tag sequences:

$$\hat{Y} = \operatorname*{argmax}_{Y \in \mathcal{Y}} P(Y|X) \qquad (8.22)$$

- at each time step the CRF computes log-linear functions over a set of relevant features, and these local features are aggregated and normalized to produce a global probability for the whole sequence.

  We'll call these K functions $F_k(X,Y)$ **global features**, since each one is a property of the entire input sequence X and output sequence Y. We compute them by decomposing into a sum of **local** features for each position i in Y:

$$F_k(X,Y) = \sum_{i=1}^{n} f_k(y_{i-1}, y_i, X, i) \qquad (8.26)$$

- constraint to only depend on the current and previous output tokens yi and yi−1 are what characterizes **a linear chain CRF**. this limitation makes it possible to use versions of the efficient Viterbi and Forward-Backwards algorithms from the HMM.
- A general CRF, allows a feature to make use of any output tokem, and are this necessary for tasks in which decision depends on the distant output tokens, like y_{i-4}, General CRFs require more complex inference, and are less commonly used for language processing.

Features in a CRG POS Tagger

the reason to use a discriminative sequence model is that it's easier to incorporate a lot of features.

In a linear-chain CRF, each local feature f_k at position *i* can depend on any information from: (*yi−1,yi,X, i*). So some legal features representing common situations might be the following:

$$\mathbb{1}\{x_i = the,\ y_i = \text{DET}\}$$
$$\mathbb{1}\{y_i = \text{PROPN}, x_{i+1} = Street, y_{i-1} = \text{NUM}\}$$
$$\mathbb{1}\{y_i = \text{VERB}, y_{i-1} = \text{AUX}\}$$

For simplicity, we assume CRF takes value 1/0.

Feature template: $\langle y_i, x_i\rangle, \langle y_i, y_{i-1}\rangle, \langle y_i, x_{i-1}, x_{i+2}\rangle$

These templates automatically populate the set of features from every instance in the training and test set. Thus for our example Janet/NNP will/MD back/VB the/DT bill/NN, when xi is the word back, the following features would be generated and have the value 1 (we've assigned them arbitrary feature numbers):

$$f_{3743}: y_i = \text{VB and } x_i = \text{back}$$
$$f_{156}: y_i = \text{VB and } y_{i-1} = \text{MD}$$
$$f_{99732}: y_i = \text{VB and } x_{i-1} = \text{will and } x_{i+2} = \text{bill}$$

Word Shape feature: represent abstract letter patter of the word by mapping lower-case letters to 'x', upper-case to 'X', numbers to 'd' & retains the punctuations.

The known-word templates are computed for every word seen in the training set;

- the unknown word features can also be computed for all words in training, or only on training words whose frequency is below some threshold.

$$\text{prefix}(x_i) = \texttt{w}$$
$$\text{prefix}(x_i) = \texttt{we}$$
$$\text{suffix}(x_i) = \texttt{ed}$$
$$\text{suffix}(x_i) = \texttt{d}$$
$$\text{word-shape}(x_i) = \texttt{xxxx-xxxxxxx}$$
$$\text{short-word-shape}(x_i) = \texttt{x-x}$$

The result of the known-word templates and word-signature features is a very large set of features. Generally, a feature cut-off is used in which features are thrown out if they have count < 5 in the training set.

Note: *In a CRF we don't learn weights for each of these local features fk. Instead, we first sum the values of each local feature (for example feature f_{3743}) over the entire sentence, to create each global feature (for example F_{3743}). It is those global features that will then be multiplied by weight w_{3743}. Thus, for training and inference there is always a fixed set of K features with K weights, even though the length of each sentence is different*.

Features for CRF Named Entity Recognizers

identity of w_i, identity of neighboring words
embeddings for w_i, embeddings for neighboring words
part of speech of w_i, part of speech of neighboring words
presence of w_i in a **gazetteer**
w_i contains a particular prefix (from all prefixes of length ≤ 4)
w_i contains a particular suffix (from all suffixes of length ≤ 4)
word shape of w_i, word shape of neighboring words
short word shape of w_i, short word shape of neighboring words
gazetteer features

Figure 8.15 Typical features for a feature-based NER system.

Gazetteer – feature useful for locations, a list of place names. This can be implemented as a binary feature indicating a phrase appears in the list.

Other related resources like **name-lists**, like lists of corporations or products, although they may not be as helpful as a gazetteer.

prefix(x_i) = L	suffix(x_i) = tane
prefix(x_i) = L'	suffix(x_i) = ane
prefix(x_i) = L'0	suffix(x_i) = ne
prefix(x_i) = L'0c	suffix(x_i) = e
word-shape(x_i) = X'Xxxxxxx	short-word-shape(x_i) = X'Xx

Words	POS	Short shape	Gazetteer	BIO Label
Jane	NNP	Xx	0	B-PER
Villanueva	NNP	Xx	1	I-PER
of	IN	x	0	O
United	NNP	Xx	0	B-ORG
Airlines	NNP	Xx	0	I-ORG
Holding	NNP	Xx	0	I-ORG
discussed	VBD	x	0	O
the	DT	x	0	O
Chicago	NNP	Xx	1	B-LOC
route	NN	x	0	O
.	.	.	0	O

Figure 8.16 Some NER features for a sample sentence, assuming that Chicago and Villanueva are listed as locations in a gazetteer. We assume features only take on the values 0 or 1, so the first POS feature, for example, would be represented as $\mathbb{1}\{\text{POS} = \text{NNP}\}$.

Inference and training for CRF:

How do we find the best tag sequence $\hat{Y}$ for a given input X? We start with Eq. 8.22:

$$\hat{Y} = \operatorname*{argmax}_{Y \in \mathcal{Y}} P(Y|X)$$

$$= \operatorname*{argmax}_{Y \in \mathcal{Y}} \frac{1}{Z(X)} \exp\left(\sum_{k=1}^{K} w_k F_k(X,Y)\right) \tag{8.27}$$

$$= \operatorname*{argmax}_{Y \in \mathcal{Y}} \exp\left(\sum_{k=1}^{K} w_k \sum_{i=1}^{n} f_k(y_{i-1}, y_i, X, i)\right) \tag{8.28}$$

$$= \operatorname*{argmax}_{Y \in \mathcal{Y}} \sum_{k=1}^{K} w_k \sum_{i=1}^{n} f_k(y_{i-1}, y_i, X, i) \tag{8.29}$$

$$= \operatorname*{argmax}_{Y \in \mathcal{Y}} \sum_{i=1}^{n} \sum_{k=1}^{K} w_k f_k(y_{i-1}, y_i, X, i) \tag{8.30}$$

Ignore exp f() and denominator Z(X) as it doesn't changes the argmax & denominator Z(X) is constant for given observation sequence X.

Learning in CRFs relies on the same supervised learning algorithms we presented for logistic regression.

- Given a sequence of observations, feature functions, and corresponding outputs, we use stochastic gradient descent to train the weights to maximize the log-likelihood of the training corpus. The local nature of linear-chain CRFs means that a CRF version of the forward-backward algorithm can be used to efficiently compute the necessary derivatives. As with logistic regression, L1 or L2 regularization is important.

Evaluation of Named Entity Recognition

POS taggers are evaluated by – **accuracy**.

Named entity recognizers are evaluated by **recall, precision**, and **F1 measure**.

To know if the difference between the F1 scores of two NER systems is a significant difference, we use the paired bootstrap test, or the similar randomization test.

For named entity tagging, the *entity* rather than the word is the unit of response.

named entity tagging has a segmentation component which is not present in tasks like text categorization or part-of-speech tagging causes some problems with evaluation.

> E.g. a system that labeled *Jane* but not *Jane Villanueva* as a person would cause two errors, a false positive for O and a false negative for I-PER. In addition, using entities as the unit of response but words as the unit of training means that there is a mismatch between the training and test conditions.

Further Details

Bidirectionality-

- One problem with the CRF and HMM architectures as presented is that the models are exclusively run left-to-right. While the Viterbi algorithm still allows present decisions to be influenced indirectly by future decisions, it would help even more if a decision about word wi could directly use information about future tags ti+1 and ti+2.
- Alternatively, any sequence model can be turned into a bidirectional model by using multiple passes. For example
    - the first pass would use only part-of-speech features from already-disambiguated words on the left.
    - In the second pass, tags for all words, including those on the right, can be used.

  Alternately, the tagger can be run twice, once left-to-right and once right-to-left.

  In Viterbi decoding, the labeler would chooses the higher scoring of the two sequences (left-to-right or right-to-left).

  Bidirectional models are quite standard for neural models, as we will see with the biLSTM model.

Rule-based Methods

- commercial approaches to NER are often based on pragmatic combinations of lists and rules, with some smaller amount of supervised ML.

- One common approach is to make repeated rule-based passes over a text, starting with rules with very high precision but low recall, and, in subsequent stages, using machine learning methods that take the output of the first pass into account (an approach first worked out for coreference:
  1. First, use high-precision rules to tag unambiguous entity mentions.
  2. Then, search for substring matches of the previously detected names.
  3. Use application-specific name lists to find likely domain-specific mentions.
  4. Finally, apply supervised sequence labeling techniques that use tags from previous stages as additional features.

POS Tagging for Morphologically Rich Languages

- Augmentations to tagging algorithms become necessary when dealing with languages with rich morphology like Czech, Hungarian and Turkish.
- Large vocabularies → unknown words →cause significant performance degradations in a wide variety of languages (including Czech, Slovene, Estonian, and Romanian)
- Highly inflectional languages also have much more information than English
- coded in word morphology, like **case** (nominative, accusative, genitive) or **gender** (masculine, feminine). Because this information is important for tasks like parsing and coreference resolution, part-of-speech taggers for morphologically rich languages need to label words with case and gender information.
- Here's a Turkish example, in which the word izin has three possible morphological/part-of-speech tags and meanings

1. Yerdeki **izin** temizlenmesi gerek. **The trace** on the floor should be cleaned.	iz + Noun+A3sg+Pnon+Gen
2. Üzerinde parmak **izin** kalmiş **Your** finger **print** is left on (it).	iz + Noun+A3sg+P2sg+Nom
3. Içeri girmek için **izin** alman gerekiyor. You need **permission** to enter.	izin + Noun+A3sg+Pnon+Nom

- Using a morphological parse sequence like Noun+A3sg+Pnon+Gen as the POS tag greatly increases the number of POS, and so tagsets can be 4 to 10 times larger than the 50–100 tags we have seen for English. With such large tagsets, each word needs to be morphologically analyzed to generate the list of possible morphological tag sequences (part-of-speech tags) for the word.
- The role of the tagger is then to disambiguate among these tags. This method also helps with unknown words since morphological parsers can accept unknown stems and still segment the affixes properly.

Summary

This chapter introduced **parts of speech** and **named entities**, and the tasks of **part-of-speech tagging** and **named entity recognition**:

- Languages generally have a small set of **closed class** words that are highly frequent, ambiguous, and act as **function words**, and **open-class** words like **nouns**, **verbs**, **adjectives**. Various part-of-speech **tagsets** exist, of between 40 and 200 tags.
- **Part-of-speech tagging** is the process of assigning a part-of-speech label to each of a sequence of words.
- **Named entities** are words for proper nouns referring mainly to people, places, and organizations, but extended to many other types that aren't strictly entities or even proper nouns.

- Two common approaches to **sequence modeling** are a **generative** approach, **HMM** tagging, and a **discriminative** approach, **CRF** tagging. We will see a neural approach in following chapters.
- The probabilities in HMM taggers are estimated by maximum likelihood estimation on tag-labeled training corpora. The Viterbi algorithm is used for **decoding**, finding the most likely tag sequence
- **Conditional Random Fields** or **CRF taggers** train a log-linear model that can choose the best tag sequence given an observation sequence, based on features that condition on the output tag, the prior output tag, the entire input sequence, and the current timestep. They use the Viterbi algorithm for inference, to choose the best sequence of tags, and a version of the Forward-Backward algorithm (see Appendix A) for training,

9. Deep Learning Architectures for Sequence Processing

Language has a temporal flow. This temporal nature is reflected in some of the algorithms we use to process language.

- the Viterbi algorithm applied to HMM POS tagging, proceeds through the input a word at a time, carrying forward information gleaned along the way.
- Yet other machine learning approaches, like those we've studied for sentiment analysis or other text classification tasks don't have this temporal nature – they assume simultaneous access to all aspects of their input.

The simple feedforward sliding-window is promising, but isn't a completely satisfactory solution to temporality. By using embeddings as inputs, it does solve the main problem of the simple n-gram models of Chapter 3 (recall that n-grams were based on words rather than embeddings, making them too literal, unable to generalize across contexts of similar words). But feedforward networks still share another weakness of n-gram approaches:

- limited context. Anything outside the context window has no impact on the decision being made.
- the use of windows makes it difficult for networks to learn systematic patterns arising from phenomena like constituency and compositionality: the way the meaning of words in phrases combine together. E.g., *all the* is treated as different entity under different windows – *all, all the, the*.

Solution to these problems:

- RRN and transformer networks.
- Both approaches have mechanisms to deal directly with the sequential nature of language that allow them to capture and exploit the temporal nature of language.
- The **RNN** offers a new way to represent the prior context, allowing the model's decision to depend on information from hundreds of words in the past.

- The **transformer** offers new mechanisms (self-attention and positional encodings) that help represent time and help focus on how words relate to each other over long distances.

Language Models Revisited

Here we will be exploring the RNN and transformer architectures through the lens of probabilistic language models (prediction of next word in seq. given some preceding context).

LM give us the ability to assign such a conditional probability to every possible next word, giving us a distribution over the entire vocabulary. We can also assign probabilities to entire sequences by using these conditional probabilities in combination with the chain rule:

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i|w_{<i})$$

evaluate language models: ability to predict unseen text. Using perplexity, as a measure of quality of LM.

RNN

network that contains a cycle within its network connections, meaning that the value of some unit is directly, or indirectly, dependent on its own earlier outputs as an input.

- While powerful, such networks are difficult to reason about and to train. However, within the general class of recurrent networks there are constrained architectures that have proven to be extremely effective when applied to language.

Elman Networks – simple RNN

Serve as the basis for more complex approaches like LSTM.

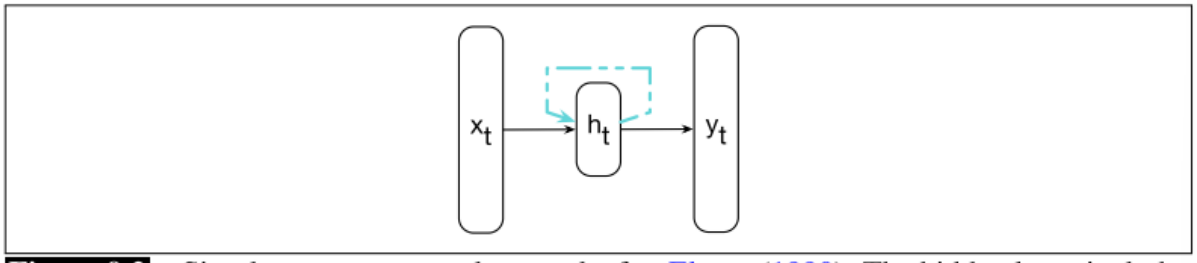


Figure 9.2 Simple recurrent neural network after Elman (1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous time step.

As with ordinary feedforward networks,

- an input vector representing the current input, x_t , is multiplied by a weight matrix and then passed through a non-linear activation function to compute the values for a layer of hidden units.
- This hidden layer is then used to calculate a corresponding output y_t.
- The key difference from a feedforward network lies in the recurrent link shown in the figure with the dashed line.
- This link augments the input to the computation at the hidden layer with the value of the hidden layer *from the preceding point in time*.

- hidden layer from the previous time step provides a form of memory, or context, that encodes earlier processing and informs the decisions to be made at later points in time.
- Critically, this approach does not impose a fixed-length limit on this prior context; the context embodied in the previous hidden layer can include information extending back to the beginning of the sequence.
- Adding this temporal dimension makes RNNs appear to be more complex than non-recurrent architectures. But in reality, they're not all that different.

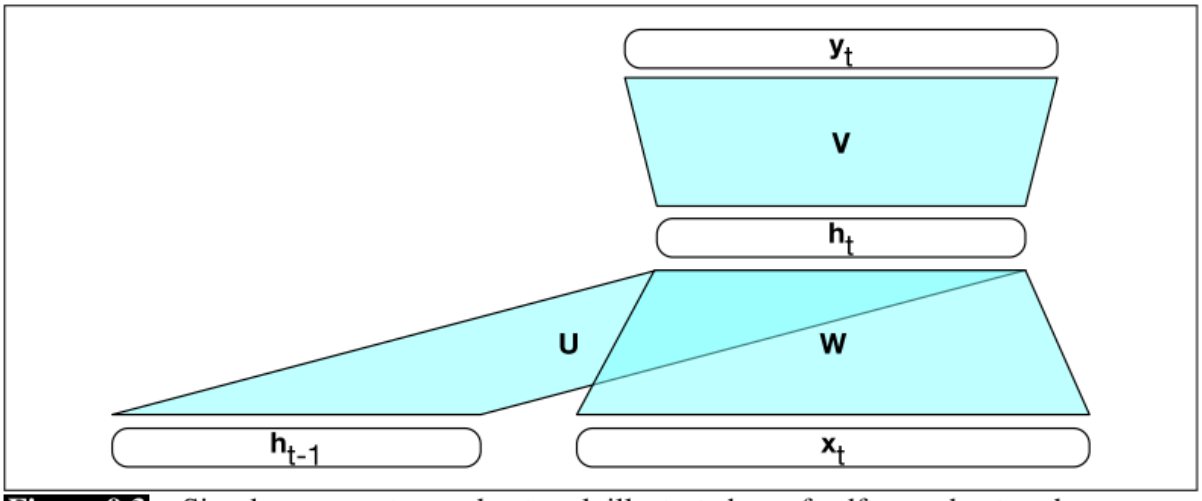


Figure 9.3 Simple recurrent neural network illustrated as a feedforward network.

The most significant change lies in the new set of weights, **U**, that connect the hidden layer from the previous time step to the current hidden layer. These weights determine how the network makes use of past context in calculating the output for the current input. As with the other weights in the network, these connections are trained via backpropagation.

Inference in RNNs

Forward inference (mapping a sequence of inputs to sequence of outputs) in RNN is nearly identical to what we've already seen with feedforward networks.

function FORWARDRNN(**x**, *network*) **returns** output sequence **y**

$\mathbf{h}^0 \leftarrow 0$
for $i \leftarrow 1$ **to** LENGTH(**x**) **do**
  $\mathbf{h}_i \leftarrow g(\mathbf{U}\mathbf{h}_{i-1} + \mathbf{W}\mathbf{x}_i)$
  $\mathbf{y}_i \leftarrow f(\mathbf{V}\mathbf{h}_i)$
return y

Figure 9.4 Forward inference in a simple recurrent network. The matrices **U**, **V** and **W** are shared across time, while new values for **h** and **y** are calculated with each time step.

Training

We have 3 sets of weights to update:

- **W**, the weights from the input layer to the hidden layer,
- **U**, the weights from the previous hidden layer to the current hidden layer, and finally
- **V**, the weights from the hidden layer to the output layer.

Backpropagation in feedforward networks:

1. To compute the loss function for the output at time t we need the hidden layer from time t −1.
2. The hidden layer at time t influences both the output at time t and the hidden layer at time t +1 (and hence the output and loss at t +1). It follows from this that to assess the error accruing to h_t, we'll need to know its influence on both the current output *as well as the ones that follow*.

We have two-pass algorithm for training the RNN weights, → **Backpropagation through time**

First pass – perform forward inference by computing $\mathbf{h}_t$, $\mathbf{y}_t$, accumulating the loss at each step in time, saving the value of hidden layer at each step for use at the next time step.

Second pass – here we process the sequence in reverse, computing the required gradients as we go, computing and saving the error term for use in the hidden layer for each step backward in time.

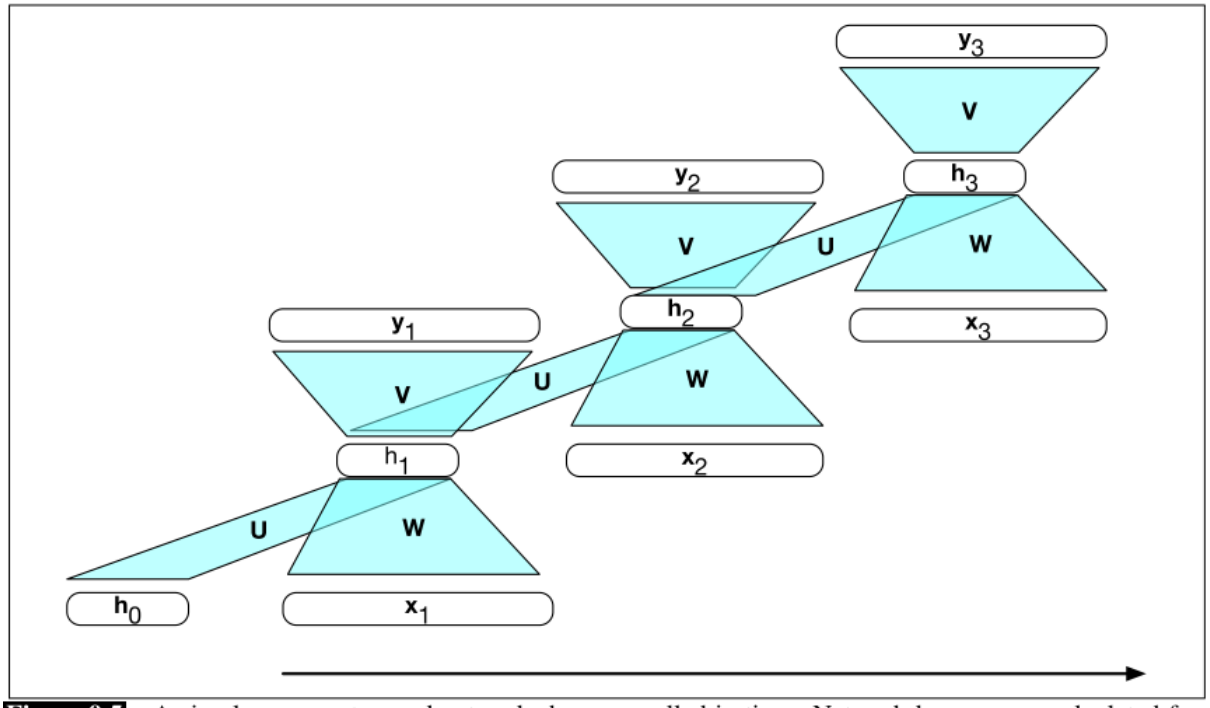


Figure 9.5 A simple recurrent neural network shown unrolled in time. Network layers are recalculated for each time step, while the weights **U**, **V** and **W** are shared in common across all time steps.

In such an approach, we provide a template that specifies the basic structure of the network, including all the necessary parameters for the input, output, and hidden layers, the weight matrices, as well as the activation and output functions to be used. Then, when presented with a specific input sequence, we can generate an unrolled feedforward network specific to that input, and use that graph to perform forward inference or training via ordinary backpropagation. For applications that involve much longer input sequences, such as speech recognition, character-level processing, or streaming of continuous inputs, unrolling an entire input sequence may not

be feasible. In these cases, we can unroll the input into manageable fixed-length segments and treat each segment as a distinct training item.

RNNs as Language Models

They process the input sequence one word at a time, attempting to predict the next word from the current word and previous hidden state.

RNNs don't have the limited context problem that n-gram models have,

- hidden state can in principle represent information about all of the preceding words all the way back to the beginning of the sequence.

Forward inference:

The input sequence **X** = [x1; ...;xt ; ...;xN] consists of a series of word embeddings each represented as a one-hot vector of size |V|×1, and the output prediction, **y**, is a vector representing a probability distribution over the vocabulary. At each step, the model uses the word embedding matrix **E** to retrieve the embedding for the current word, and then combines it with the hidden layer from the previous step to compute a new hidden layer. This hidden layer is then used to generate an output layer which is passed through a softmax layer to generate a probability distribution over the entire vocabulary. That is, at time t:

$$\mathbf{e}_t = \mathbf{E}\mathbf{x}_t \tag{9.6}$$

$$\mathbf{h}_t = g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t) \tag{9.7}$$

$$\mathbf{y}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t) \tag{9.8}$$

The vector resulting from **Vh** can be thought of as a set of scores over the vocabulary given the evidence provided in **h**. Passing these scores through the softmax normalizes the scores into a probability distribution.

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i|w_{1:i-1}) \tag{9.10}$$

$$= \prod_{i=1}^{n} \mathbf{y}_i[w_i] \tag{9.11}$$

To train the RNN as language model, we need corpus as training material, use cross-entropy as the loss function. Cross entropy loss → difference between predicted probability distribution and correct distribution.

$$L_{CE} = -\sum_{w \in V} \mathbf{y}_t[w] \log \hat{\mathbf{y}}_t[w] \tag{9.12}$$

In the case of language modeling, the correct distribution yt comes from knowing the next word. This is represented as a one-hot vector corresponding to the vocabulary where the entry for the actual next word is 1, and all the other entries are 0. Thus, the cross-entropy loss for language modeling is determined by the probability the model assigns to the correct next word. So at time t

the CE loss is the negative log probability the model assigns to the next word in the training sequence.

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\log \hat{\mathbf{y}}_t[w_{t+1}] \quad (9.13)$$

Thus at each word position t of the input, the model takes as input the correct sequence of tokens $w_{1:t}$, and uses them to compute a probability distribution over possible next words so as to compute the model's loss for the next token w_{t+1}. Then we move to the next word, we ignore what the model predicted for the next word and instead use the correct sequence of tokens $w_{1:t+1}$ to estimate the probability of token w_{t+2}.

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

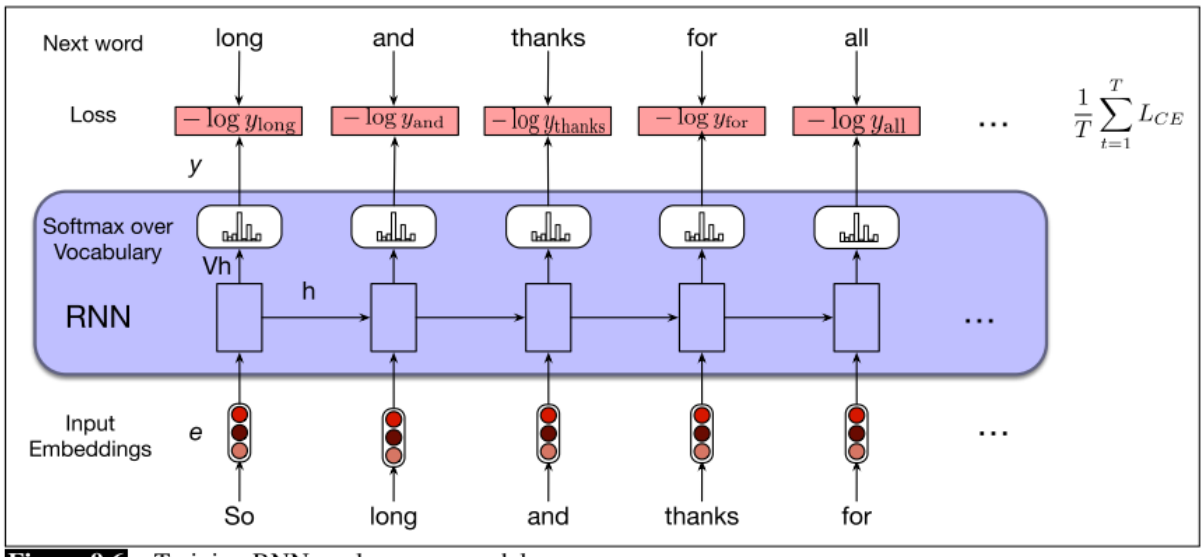


Figure 9.6 Training RNNs as language models.

The weights in the network are adjusted to minimize the average CE loss over the training sequence via gradient descent.

embedding matrix **E** and the final layer matrix **V**, which feeds the output softmax, are quite similar. The rows of E represent the word embeddings for each word in the vocabulary learned during the training process with the goal that words that have similar meaning and function will have similar embeddings. And, since the length of these embeddings corresponds to the size of the hidden layer d_h, the shape of the embedding matrix E is $|V| \times d_h$.

The final layer matrix **V** provides a way to score the likelihood of each word in the vocabulary given the evidence present in the final hidden layer of the network through the calculation of **Vh**. This entails that it also has the dimensionality $|V| \times dh$.

the rows of V provide a second set of learned word embeddings that capture relevant aspects of word meaning and function. This leads to an obvious question – is it even necessary to have both? **Weight tying** is a method that dispenses with this redundancy and uses a single set of embeddings at the input and softmax layers. That is, **E = V.**

To do this, we set the dimensionality of final layer to be the same d_h, (or add an additional projection layer to do the same thing), and simply use the same matrix for both layers. In addition to providing improved perplexity results, this approach significantly reduces the number of parameters required for the model.

RNN for other NLP tasks

Lets apply RNN to three types of NLP tasks:

- *sequence classification* tasks like sentiment analysis and topic classification,
- *sequence labelling* &
- *text generation tasks*.

Sequence Labelling

Network's task is to assign a label chosen from a small fixed set of labels to each element of a sequence.

In an RNN approach to sequence labelling, inputs are word embeddings and the outputs are tag probabilities generated by a SoftMax layer over the given tag-set, [Fig. 9.7].

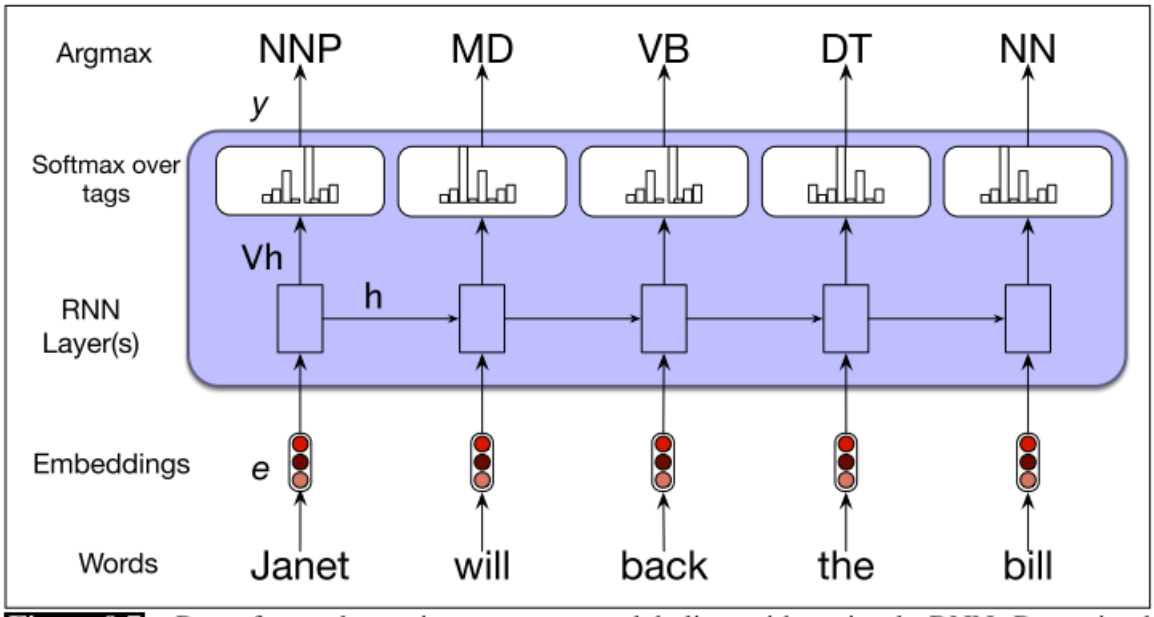


Figure 9.7 Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.

- The inputs at each time step are pre-trained word embeddings corresponding to the input tokens.
- The RNN block is an abstraction that represents an unrolled simple recurrent network consisting of an input layer, hidden layer, and output layer at each time step, as well as the shared **U, V** and **W** weight matrices that comprise the network.
- The outputs of the network at each time step represent the distribution over the POS tagset generated by a SoftMax layer.

To generate sequence of tags for input, we run forward inference over the input seq. and select the most likely tag from the softmax at each step. Since we're using a softmax layer to generate the probability distribution over the output tagset at each time step, we will again employ cross entropy loss during training.

RNN for Sequence Classification

- Classifying entire sequences instead of tokens within them.
- To apply RNNs in this setting:
    1. we pass the text to be classified through the RNN a word at a time generating a new hidden layer at each time step.
    2. We can then take the hidden layer for the last token of the text, h_n, to constitute a compressed representation of the entire sequence.
    3. We can pass this representation h_n to a feedforward network that chooses a class via a softmax over the possible classes.

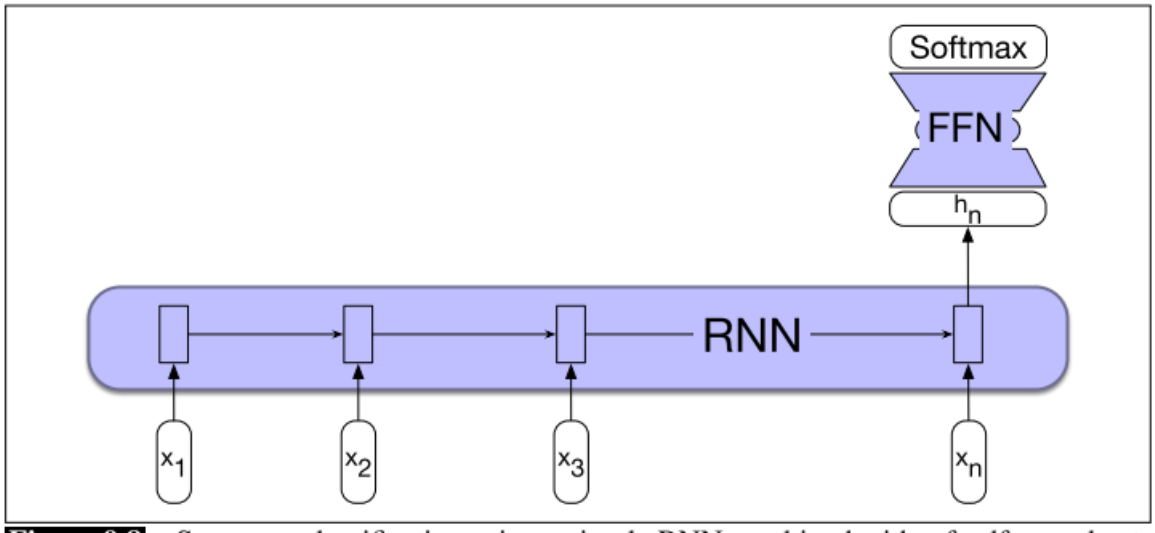


Figure 9.8 Sequence classification using a simple RNN combined with a feedforward network. The final hidden state from the RNN is used as the input to a feedforward network that performs the classification.

Here we don't need intermediate outputs for the words in the sequence preceding the last element. Therefore, there are no loss terms associated with those elements.

The loss function used to train the weights in the network is entirely based on final text classification task. The output from the softmax output from the feedforward classifier together with a cross-entropy loss drives the training. The error signal from the classification is backpropagated all the way through the weights in the feedforward classifier through, to its input, and then through to the three sets of weights in the RNN as

The training regimen that uses the loss from a downstream application to adjust the weights all the way through the network is referred to as **end-to-end training**.

Instead of using just the last token h_n to represent the whole sequence, we can use some sort of pooling function of all the hidden states h_i for each word i in the sequence.

- For example, we can create a representation that pools all the n hidden states by taking their element-wise mean:

$$\mathbf{h}_{mean} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{h}_i \qquad (9.14)$$

- We can take the element-wise max: the element wise of a set of n vectors in a new vector whose *k*th element is the max of the *k*th elements of all the n vectors.

Generation with RNN based Language Models

We first randomly sample a word to begin a sequence based on its suitability as the start of a sequence. We then continue to sample words *conditioned on our previous choices* until we reach a pre-determined length, or an end of sequence token is generated.

Today, this approach of using a language model to incrementally generate words by repeatedly sampling the next word conditioned on our previous choices is called **autoregressive generation**.

Procedure:

1. Sample a word in the output from the softmax distribution that results from using the beginning of sentence marker, <s>, as the first input.
2. Use the word embedding for that first word as the input to the network at the next time step, and then sample the next word in the same fashion.
3. Continue generating until the end of sentence marker, </s>, is sampled or a fixed length limit is reached.

Autoregressive model is a model that predicts a value at time t based on a linear function of the previous values at times $t-1$, $t-2$, and so on.
Although the LM are not linear (since they have many layers of non-linearities), we loosely refer to this generation technique as autoregressive generation since the word generated at each time step is conditioned on the word selected by the network from the previous step.
In the image below, details of the RNN's hidden layers and recurrent connections are hidden within the blue block.

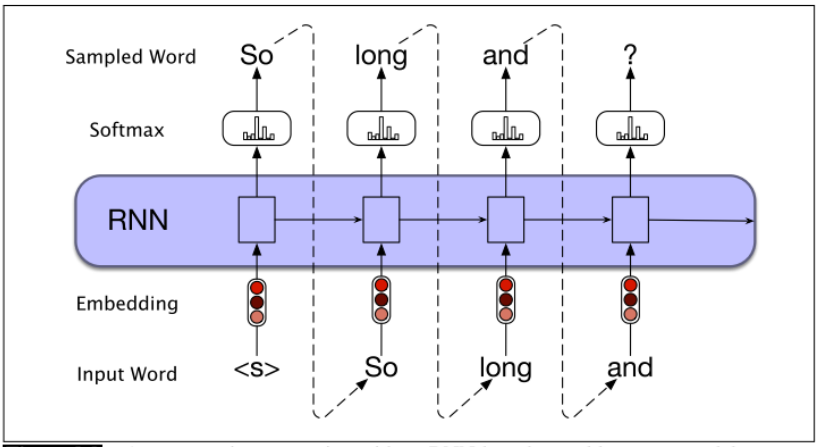


Figure 9.9 Autoregressive generation with an RNN-based neural language model.

The key to these approaches is to prime the generation component with an appropriate context. That is, instead of simply using <s> to get things started we can provide a richer task-appropriate context;

for translation the context is the sentence in the source language; for summarization it's the long text we want to summarize.

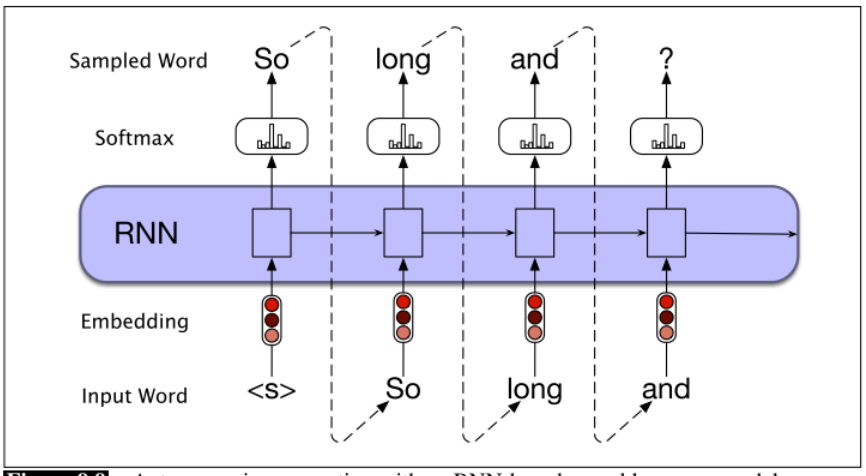


Figure 9.9 Autoregressive generation with an RNN-based neural language model.

Stacked and Bidirectional RNN architectures

By combining the feed forward network of unrolled computational graphs with vectors as common inputs and outputs, complex networks can be treated as modules that can be combined in creative ways. Here let's introduce 2 more common network architectures used in language processing with RNNs.

Stacked RNNs

Till now RNN consisted of seq of words/character embeddings (vectors) and outputs have been vectors useful for predicting words, tags or sequence labels.

However, we can use entire sequence of outputs from one RNN as an input sequence to another one. **Stacked RNNs** consist of multiple networks where the output of one layer serves as the input to a subsequent layer.

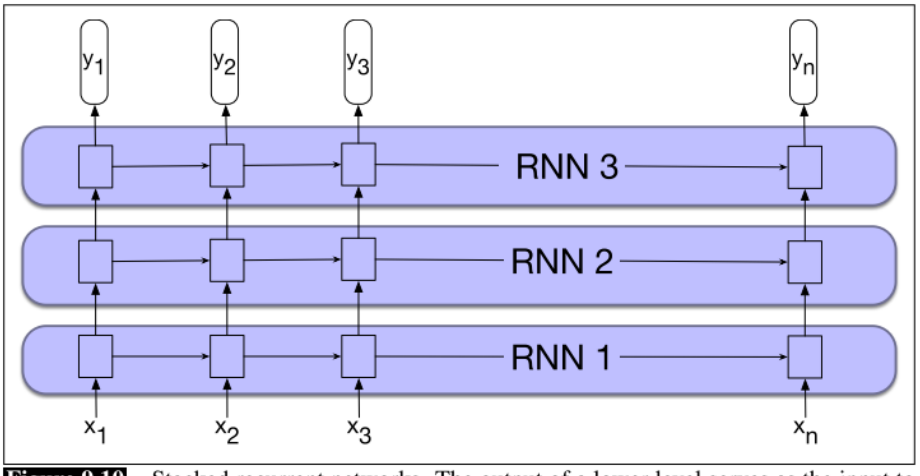


Figure 9.10 Stacked recurrent networks. The output of a lower level serves as the input to higher levels with the output of the last network serving as the final output.

Stacked RNNs outperform single-layer networks.

1. The reason is → the network induces representations at differing levels of abstraction across layers. Just as the early stages of the human visual system detect edges that are then used

for finding larger regions and shapes, the initial layers of stacked networks can induce representations that serve as useful abstractions for further layers—representations that might prove difficult to induce in a single RNN.

2. The optimal number of stacked RNNs is specific to each application and to each training set. However, as the number of stacks is increased the training costs rise quickly.

Bidirectional RNNs

RNN uses information from the left context to make its predictions at time *t*. But in many applications we have access to the entire input sequence; in those cases we would like to use words from the context to the right of *t*. How to do it?

- Run 2 separate RNNs, one L → R, and R → L and concatenate their representations.
- In L → R RNN, hidden state at given time t represents everything the network knows about the sequences upto that point. The state is a function of the inputs x1, ...,xt and represents the context of the network to the left of the current time.

$$\mathbf{h}_t^f = RNN_{forward}(\mathbf{x}_1, \ldots, \mathbf{x}_t) \quad (9.15)$$

  - $\mathbf{h}_t^f$ corresponds to normal hidden state at time t, representing everything the network has gleaned from the sequence so far.
- To the advantage of context to the right of the current input, we can train an RNN on a *reversed* input sequence. With this approach the hidden state at time t represents info about the sequence to the right of the current input:

$$\mathbf{h}_t^b = RNN_{backward}(\mathbf{x}_t, \ldots \mathbf{x}_n) \quad (9.16)_D$$

  - $\mathbf{h}_t^b$ represents all information we have discerned about the sequence from *t* to the end of the sequence.

The **Bidirectional RNN** combines two independent RNNs, one where the input is processed from start to the end, and other from the end to start. We then concatenate the two representations computed by the networks into a single vector that captures both the left and right contexts of an input at each point in time. Vector concatenation can be represented in 2 ways: (using ';' or ⊕ symbol)

$$\mathbf{h}_t = [\mathbf{h}_t^f ; \mathbf{h}_t^b] = \mathbf{h}_t^f \oplus \mathbf{h}_t^b \quad (9.17)$$

Bidirectional network that concatenates the outputs of forward and backward pass.

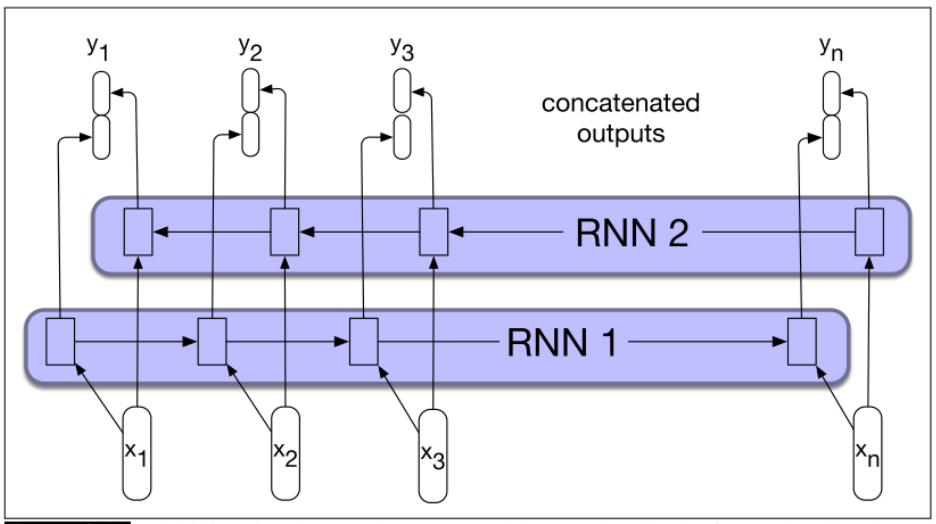


Figure 9.11 A bidirectional RNN. Separate models are trained in the forward and backward directions, with the output of each model at each time point concatenated to represent the bidirectional state at that time point.

Other ways to combine forward and backward contexts include element-wise addition or multiplication. The output at each step-in time thus captures information to the left and to the right of the current input. In sequence labeling applications, these concatenated outputs can serve as the basis for a local labeling decision.

Bidirectional RNN – effective for sequence classification.

Reminder: For sequence classification we used the final hidden state of the RNN as the input to a subsequent feedforward classifier. Difficulty with this approach final state naturally reflects more information about the end of sentence than its beginning.
Bidirectional RNNs provide a simple solution to this problem, as shown (9.12), we simply combine the final hidden states from the forward and backward passes (for example by concatenation) and use that as input for follow-on processing.

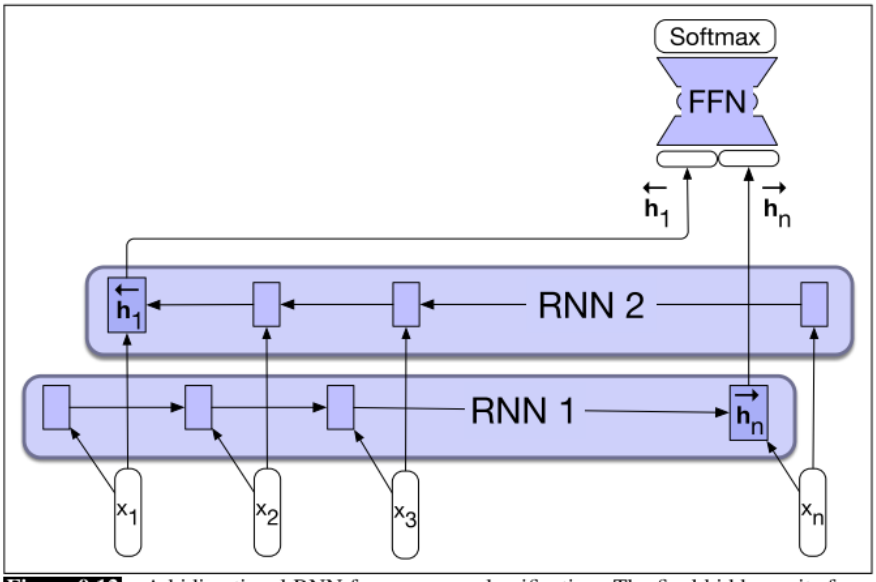


Figure 9.12 A bidirectional RNN for sequence classification. The final hidden units from the forward and backward passes are combined to represent the entire sequence. This combined representation serves as input to the subsequent classifier.

LSTM:

In practice it is difficult to train RNNs for tasks requiring network to make use of information distant from the current point of processing.
The information encoded in the hidden states is pretty local, more relevant to most recent part of input sequence and recent decisions.

The network should retain information alteast by that length so that, it remembers it is dealing with singular/plural objects.

RNN drawbacks:

1. Hidden layers - Weights that determine the values in the hidden layers, are being asked to perform two tasks simultaneously:
    a. provide information useful for the current decision,
    b. updating and carrying forward information required for future decisions.

2. Need of backpropagation of error signal through time.
    a. Hidden layer at time t contributes to the loss at the next time step since it takes part in that calculation. As a result, during the backward pass of training, the hidden layers are subject to repeated multiplications, as determined by the length of the sequence. A frequent result of this process is that the gradients are eventually driven to zero, a situation called the **vanishing gradients** problem.

Solution:

More complex n/w architectures. They enable the network to learn to forget information that is no longer needed and to remember information req. for decision still to come.

Extension to RNNS → **LSTM** (**Long Short-term Memory**)

LSTM divide context management problem into 2 sub problems:

1. Removing info no longer needed from the context.
2. Adding info likely to be needed for later decision making.

How to achieve both of them: learning how to manage this context rather than hard-coding a strategy into architecture.

LSTMs accomplish this by first adding an explicit context layer to the architecture (in addition to the usual recurrent hidden layer), and through the use of specialized neural units that make use of gates to control the flow of information into and out of the units that comprise the network layers. These gates are implemented through the use of additional weights that operate sequentially on the input, and previous hidden layer, and previous context layers.

Gates in LSTM – common design pattern, each consists of feed forward layer, followed by sigmoid activation function (coz we need output as 0 or 1), followed by pointwise multiplication with the layer being gated. Values in the layer being gated that align with values near 1 in the mask are passed through nearly unchanged; values corresponding to lower values are essentially erased.

Forget Gate: deletes information from the context that is no longer needed. The forget gate computes a weighted sum of the previous state's hidden layer and the current input and passes that through a sigmoid. This mask is then multiplied elementwise by the context vector to remove the information from context that is no longer required. Elementwise multiplication of two vectors (represented by the operator $\odot$, and sometimes called the **Hadamard product**) is the vector of the same dimension as the two input vectors, where each element i is the product of element i in the two input vectors:

$$\mathbf{f}_t = \sigma(\mathbf{U}_f\mathbf{h}_{t-1} + \mathbf{W}_f\mathbf{x}_t) \quad (9.19)$$

$$\mathbf{k}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t \quad (9.20)$$

Next task – compute the actual info we need to extract from previous hidden state to current inputs –

$$\mathbf{g}_t = \tanh(\mathbf{U}_g\mathbf{h}_{t-1} + \mathbf{W}_g\mathbf{x}_t) \quad (9.21)$$

We now, generate mask for add gate to select the information to add to the current context.

$$\mathbf{i}_t = \sigma(\mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{W}_i\mathbf{x}_t) \quad (9.22)$$

$$\mathbf{j}_t = \mathbf{g}_t \odot \mathbf{i}_t \quad (9.23)$$

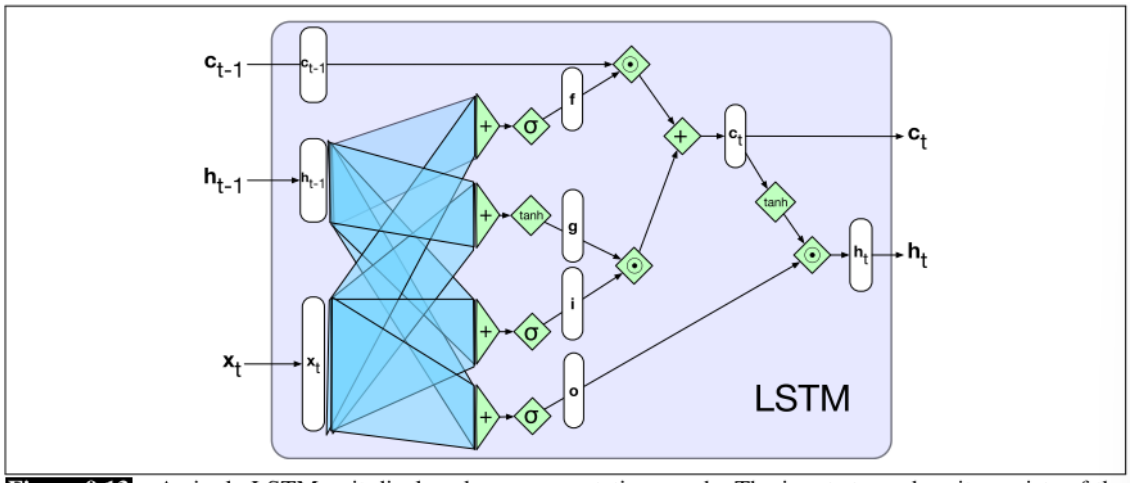


Figure 9.13 A single LSTM unit displayed as a computation graph. The inputs to each unit consists of the current input, x, the previous hidden state, h_{t-1}, and the previous context, c_{t-1}. The outputs are a new hidden state, h_t and an updated context, c_t.

Next, we add this to the modified context vector to get our new context vector.

$$\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t \qquad (9.24)$$

Output Gate: used to decide what information is required for the current hidden state (as opposed to what information needs to be preserved for future decisions).

$$\mathbf{o}_t = \sigma(\mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{W}_o\mathbf{x}_t) \qquad (9.25)$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \qquad (9.26)$$

Fig. 9.13 – complete computation for single LSTM unit.

Given the appropriate weights for the various gates, an LSTM accepts as input the context layer, and hidden layer from the previous time step, along with the current input vector. It then generates updated context and hidden vectors as output. The hidden layer, h_t , can be used as input to subsequent layers in a stacked RNN, or to generate an output for the final layer of a network.

Gated Unit, Layers and networks

Complexity of LSTM is encapsulated within basic processing units, allowing us to maintain modularity and to easily experiment with different architectures.

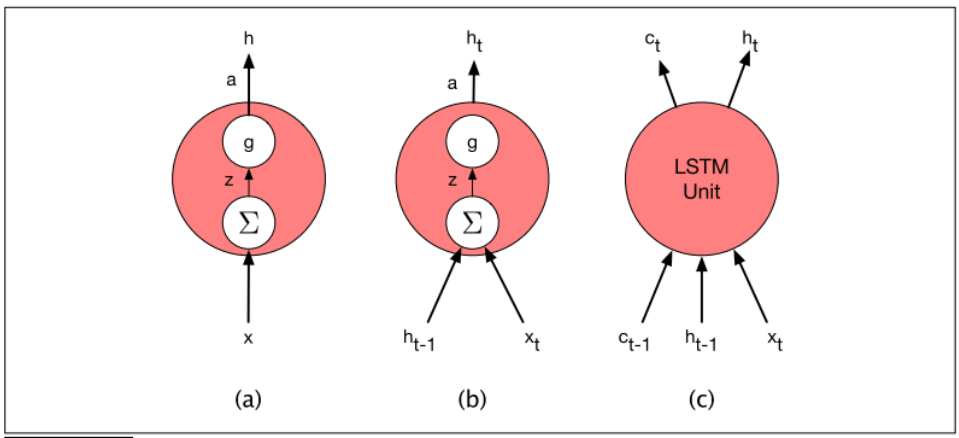


Figure 9.14 Basic neural units used in feedforward, simple recurrent networks (SRN), and long short-term memory (LSTM).

a. basic feedforward unit where a single set of weights and a single activation function determine its output, and when arranged in a layer there are no connections among the units in the layer.
b. represents the unit in a simple recurrent network. two inputs and an additional set of weights to go with it. still a single activation function and output.
c. The increased complexity of the LSTM units is encapsulated within the unit itself. The only additional external complexity for the LSTM over the basic recurrent unit (b) is the presence of the additional context vector as an input and output. The modularity is key to power and widespread applicability of LSTM units.

Self-Attention Networks: Transformers

Drawback of LSTM: they can't solve the problem of passing information through an extended series of recurrent connections leading to information loss and difficulties in training.

- the inherently sequential nature of recurrent networks makes it hard to do computation in parallel.

These considerations led to the development of **transformers** – an approach to sequence processing that eliminates recurrent connections and returns to architectures reminiscent of the fully connected networks.

- Transformers map sequences of input vectors (x1, ...,xn) to sequences of output vectors (y1, ...,yn) of the same length.
- Are made up of stack s of transformer blocks, which are multilayer network networks made by combining simple linear layers, feedforward networks and **self-attention** layers, key innovation of transformers.
- Self-attention allows a network to directly extract and use information from arbitrarily large contexts without the need to pass it through intermediate recurrent connections as in RNNs.

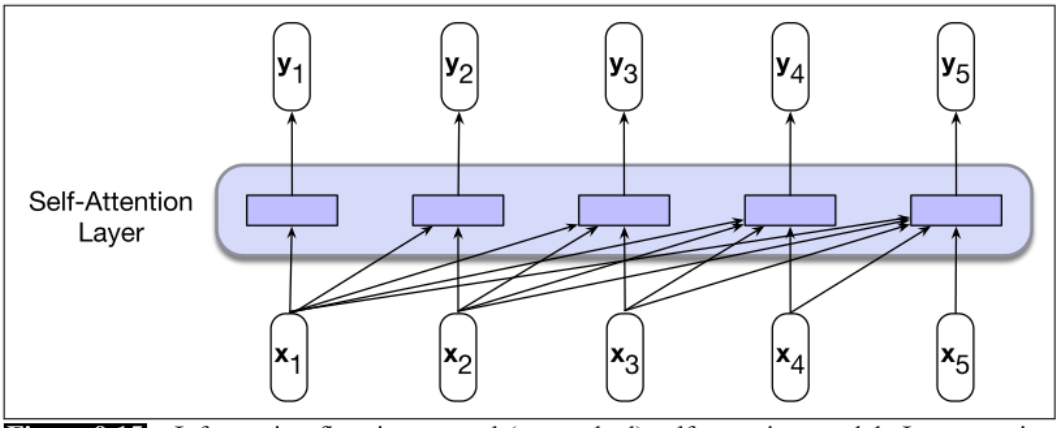


Figure 9.15 Information flow in a causal (or masked) self-attention model. In processing each element of the sequence, the model attends to all the inputs up to, and including, the current one. Unlike RNNs, the computations at each time step are independent of all the other steps and therefore can be performed in parallel.

- Above image represents flow of info in single casual, or backward looking, self-attention layer.
- As with the overall transformer, a self-attention layer maps input sequences (x1, ...,xn) to output sequences of the same length (y1, ...,yn).
- When processing input, the model has access to all the input up to and including the one under consideration, but no access to info about the input beyond current one.

- The computation performed for each item is independent of all other computations.
- So, we can use them for autoregressive generation, we can easily parallelize both forward inference and training of such models.
- Core of attention-based approach – ability to *compare* an item of interest to a collection of other items in a way that reveals their relevance in the current context. In self attention, the set of comparisons are to other elements within a given sequence. The result of these comparisons is then used to compute an output for the current input.
- Result of this comparison as a score: $$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j \quad (9.27)$$
- The result of . product is scalar ranging from -infinity to +infnity. Larger the value, the similar values are being compared. For y3 calculation, we need to normalize the 3 scores: x3.x1, x3.x2, x1.x2 .

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \ \forall j \leq i \quad (9.28)$$

$$= \frac{\exp(\text{score}(\mathbf{x}_i, \mathbf{x}_j))}{\sum_{k=1}^{i} \exp(\text{score}(\mathbf{x}_i, \mathbf{x}_k))} \ \forall j \leq i \quad (9.29)$$

$$\mathbf{y}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{x}_j \quad (9.30)$$

- Eq. 9.27 and 9.30 represent the core of an attention based approach – set of comparison to relevant item in some context, a normalization of those scores to provide a probability distribution, followed by a weighted sum using this distribution. The output y is the result of this straightforward computation over the inputs.
- Later we will see the idea of attention for LSTM-based encoder-decoder models for machine translation.

How transformers play role in each input embedding plays during the course of attention process.

- As *the current focus of attention* when being compared to all of the other preceding inputs. We'll refer to this role as a **query**.
- In its role as *a preceding input* being compared to the current focus of attention. We'll refer to this role as a **key**.
- And finally, as a **value** used to compute the output for the current focus of attention.

To capture these three different roles, transformers introduce weight matrices $\mathbf{W^Q}$, $\mathbf{W^K}$, and $\mathbf{W^V}$. These weights will be used to project each input vector xi into a representation of its role as a key, query, or value.

$$\mathbf{q}_i = \mathbf{W^Q}\mathbf{x}_i; \ \ \mathbf{k}_i = \mathbf{W^K}\mathbf{x}_i; \ \ \mathbf{v}_i = \mathbf{W^V}\mathbf{x}_i \quad (9.31)$$

The inputs x and outputs y of transformers, as well as the intermediate vectors after the various layers, all have the same dimensionality 1×d.

For now let's assume the dimensionalities of the transform matrices are $W^Q \in R^{d \times d}$, $W^K \in R^{d \times d}$, and $W^V \in R^{d \times d}$.

d_k – dimension for key and query vectors.

d_v – dimension for value vectors.

Both will be set to *d*.

Score between current focus of attention xi and element in the proceeding context xj consist of a dot product between its query vector qi and the preceding element's key vector k_j. This dot product has the right shape since both the query and the key are of dimensionality 1×d.

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{q}_i \cdot \mathbf{k}_j \quad (9.32)$$

softmax calculation resulting in αi, j remains the same, but the output calculation for yi is now based on a weighted sum over the value vectors v.

$$\mathbf{y}_i = \sum_{j \le i} \alpha_{ij} \mathbf{v}_j \tag{9.33}$$

Exponentiating such large dot product values can lead to numerical issues and to an effective loss of gradients during training. To avoid this, dot product needs to be scaled – by dividing the result of dot product by a factor related to the size of embeddings before passing them through softmax.
A typical approach is to divide the dot product by the square root of the dimensionality of the query and key vectors (dk), leading us to update our scoring function one more time,

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} \tag{9.34}$$

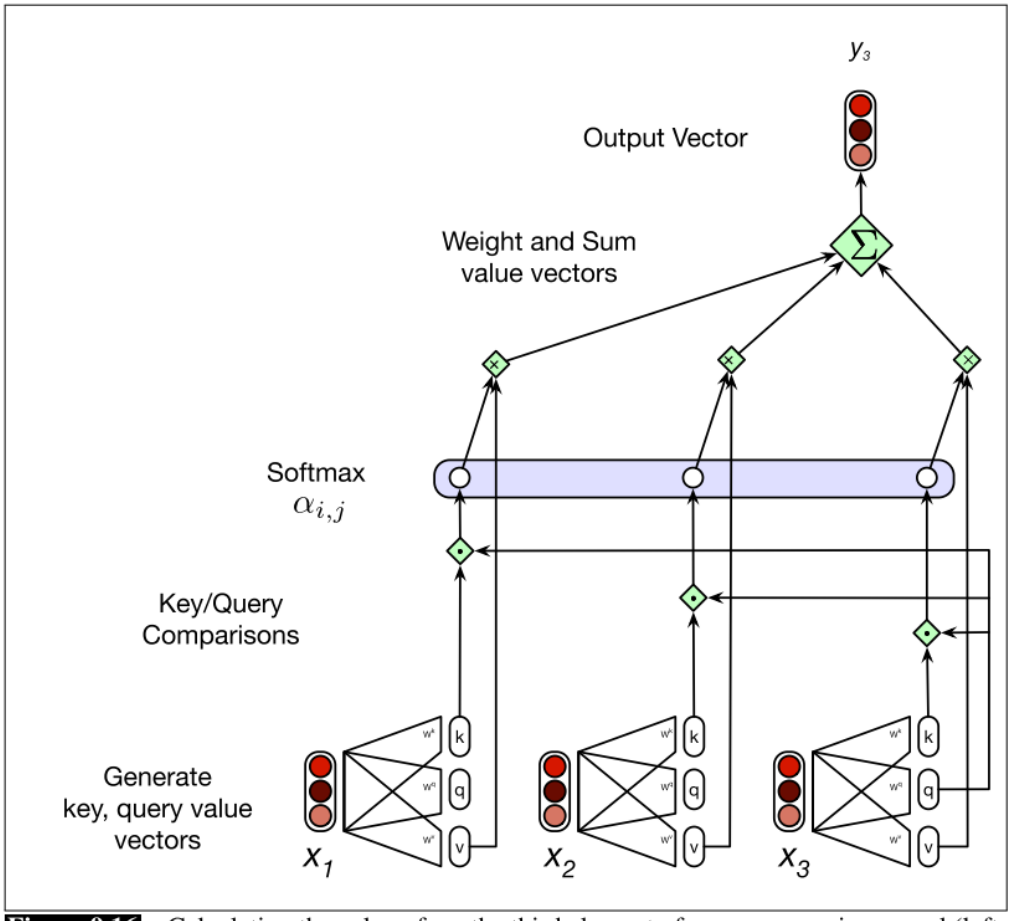


Figure 9.16 Calculating the value of $\mathbf{y}_3$, the third element of a sequence using causal (left-to-right) self-attention.

$$\textit{SelfAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\intercal}{\sqrt{d_k}}\right)\mathbf{V} \tag{9.36}$$

Fig. 9.17 also makes it clear that attention is quadratic in the length of the input, since at each layer we need to compute dot products between each pair of tokens in the input. This makes it extremely expensive for the input to a transformer to consist of long documents (like entire Wikipedia pages, or novels), and so most applications have to limit the input length, for example to at most a page or a paragraph of text at a time. Finding more efficient attention mechanisms is an ongoing research direction.

Transformer Blocks

The self-attention calculation lies at the core of what's called a transformer block, which, in addition to the self-attention layer, includes additional feedforward layers, residual connections, and normalizing layers. The input and output dimensions of these blocks are matched so they can be stacked just as was the case for stacked RNNs.

In image, we see a standard transformer block consisting of a single attention layer followed by a fully-connected feedforward layer with residual connections and layer normalizations following each.

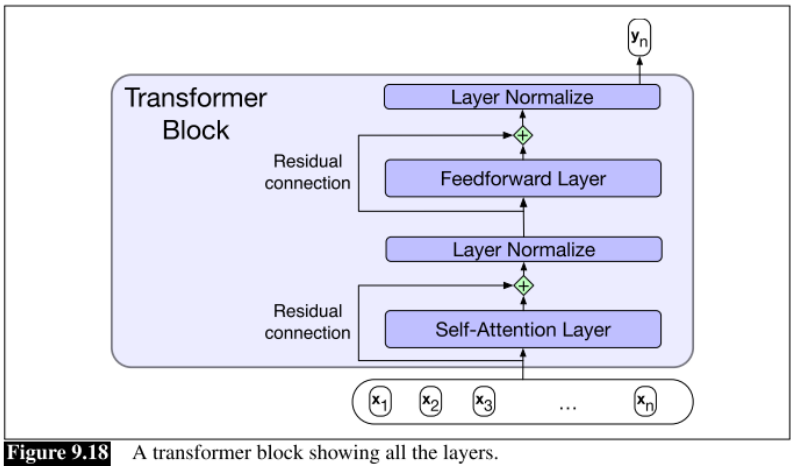


Figure 9.18 A transformer block showing all the layers.

Residual connections and layer norm –

residual connections are connections that pass information from a lower layer to a higher layer without going through the intermediate layer. Allowing information from the activation going forward and the gradient going backwards to skip a layer improves learning and gives higher level layers direct access to information from lower layers.

Residual connections in transformers are implemented by added a layer's input vector to its output vector before passing it forward. In the image, residual connections are used with both the attention and feedforward sublayers. These summed vectors are then normalized using layer normalization.

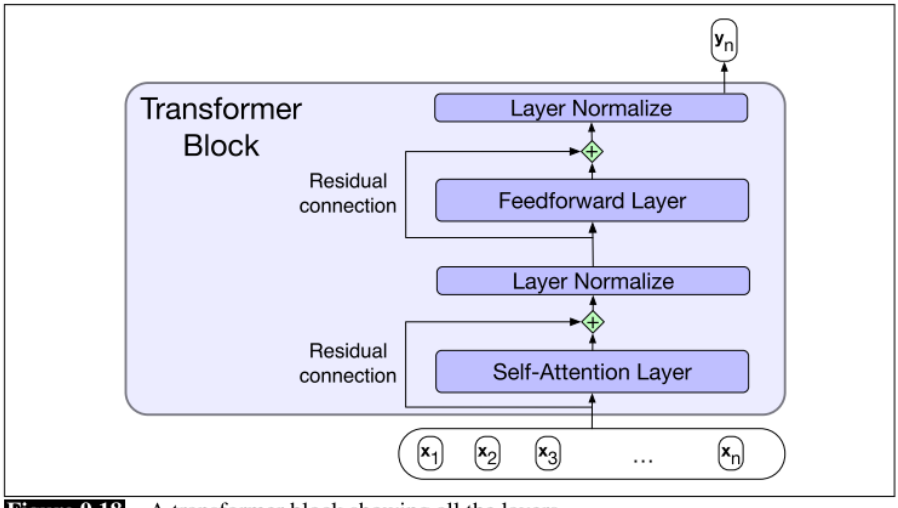


Figure 9.18 A transformer block showing all the layers.

If we think of a layer as one long vector of units, the resulting function computed in a transformer block can be expressed as →

$$\mathbf{z} = \text{LayerNorm}(\mathbf{x} + \text{SelfAttn}(\mathbf{x})) \quad (9.37)$$

$$\mathbf{y} = \text{LayerNorm}(\mathbf{z} + \text{FFNN}(\mathbf{z})) \quad (9.38)$$

Improving the training performance of deep NN – using **Layer Normalization** (**layer norm**) by keeping the values of a hidden layer in a range that facilitates gradient-based training. Layer norm is a variation of the standard score, or z-score, from statistics applied to a single hidden layer.

Steps of layer normalization:

1. Calculate the mean, μ, and standard deviation, σ, over the elements of the vector to be normalized. Given a hidden layer with dimensionality dh, these values are calculated as follows.

$$\mu = \frac{1}{d_h}\sum_{i=1}^{d_h} x_i \quad (9.39)$$

$$\sigma = \sqrt{\frac{1}{d_h}\sum_{i=1}^{d_h}(x_i-\mu)^2} \quad (9.40)$$

2. The vector components are normalized by subtracting the mean from each and dividing the standard deviation. The result of this computation is new vector with zero mean and standard deviation of one.

$$\hat{\mathbf{x}} = \frac{(\mathbf{x}-\mu)}{\sigma} \quad (9.41)$$

3. Finally, in the standard implementation of layer normalization, two learnable parameters, γ and β, representing gain and offset values, are introduced.

$$LayerNorm = \gamma\hat{\mathbf{x}} + \beta \quad (9.42)$$

Multi-head Attention

It would be difficult for a single transformer block to learn to capture all of the different kinds of parallel relations among its inputs.

Transformers address this issue with **multihead self-attention layers**. These are sets of self-attention layers, called heads, that reside in parallel layers at the same depth in a model, each with its own set of parameters. Provided these distinct sets of parameters, each head can learn different aspects of the relationships that exist among inputs at the same level of abstraction.

$$MultiHeadAttn(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2 \ldots \oplus \mathbf{head}_h)\mathbf{W}^O \quad (9.43)$$

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_i^Q\ ;\ \mathbf{K} = \mathbf{X}\mathbf{W}_i^K\ ;\ \mathbf{V} = \mathbf{X}\mathbf{W}_i^V \quad (9.44)$$

$$\mathbf{head}_i = SelfAttention(\mathbf{Q},\mathbf{K},\mathbf{V}) \quad (9.45)$$

Modelling word order: positional embeddings

How does the T model the position of each token in the input sequence?

With RNN - info about the order of inputs was built into the sequence of the model.

T – they don't have any notion of relative, absolute, positions of the token in the input.

Because, even after scrambling the order of the inputs on the attention computations, the same answer is obtained. Fig 9.19

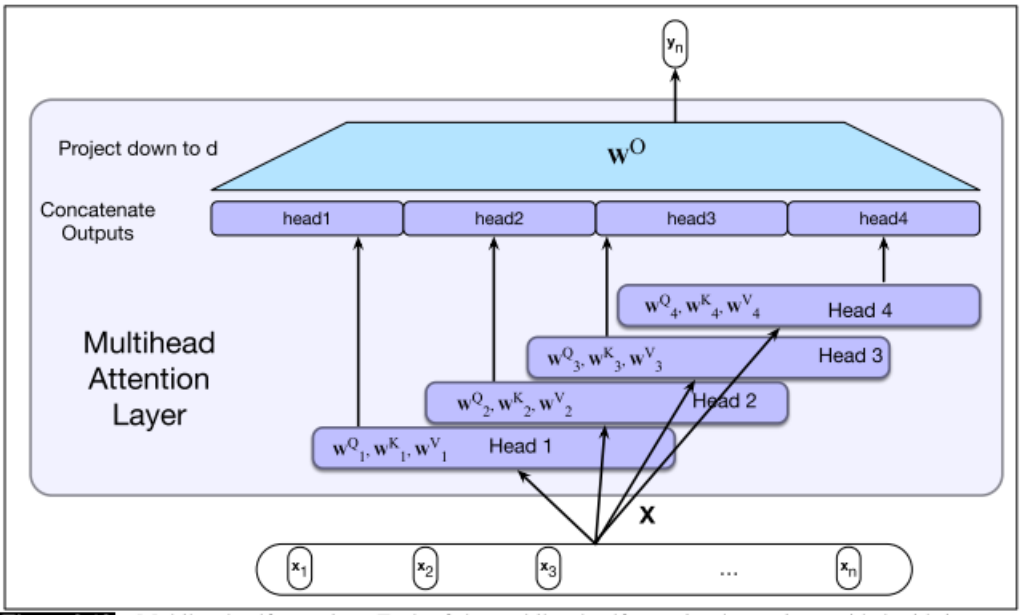


Figure 9.19 Multihead self-attention: Each of the multihead self-attention layers is provided with its own set of key, query and value weight matrices. The outputs from each of the layers are concatenated and then projected down to d, thus producing an output of the same size as the input so layers can be stacked.

Solution: modify the input embeddings by combing them with **positional embeddings** specific to each position in an input sequence.

Where to get these positional embeddings?

- Start with randomly initialized embeddings corresponding to each possible input position up to some maximum length.
- As with the word embeddings, these positional embeddings are learned along with other parameters during training. In the input embedding that captures positional information, we add word embeddings for each input to its corresponding positional embedding. Fig 9.20

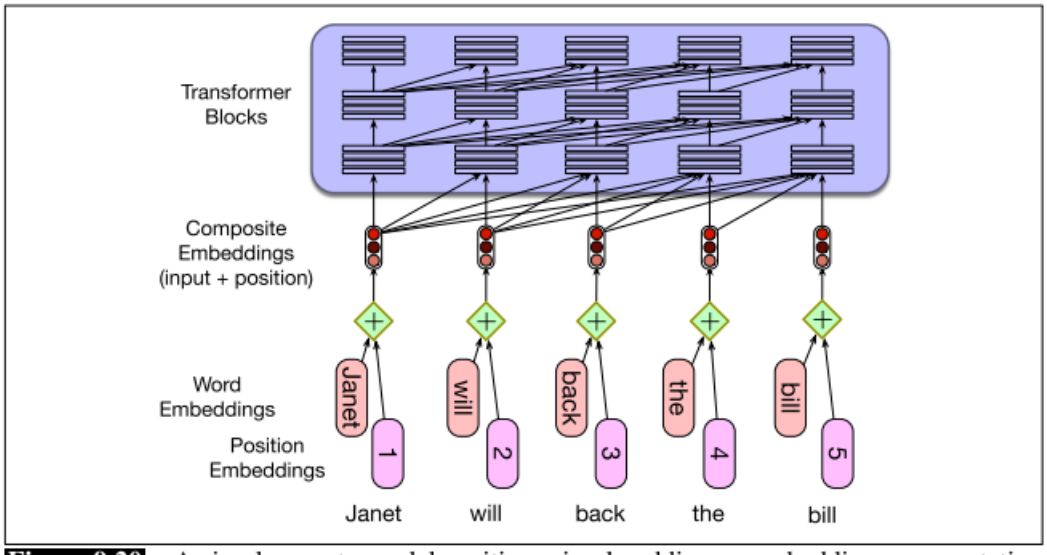


Figure 9.20 A simple way to model position: simply adding an embedding representation of the absolute position to the input word embedding.

- Potential problem with simple absolute position embedding approach is that there will be plenty of training examples for the initial positions in our inputs and correspondingly fewer at the outer length limits.
- These latter embeddings may be poorly trained and may not generalize well during testing.

Alternative approach:

- Choosing a static function that maps integer input to real-valued vectors in a way that captures the inherent relationships among the positions.
    - Eg. Position 4 in an input is more closely related to position 5 that position 17.
- A combination of sine and cosine f() with differing frequencies was used in the original transformer work.
- Developing better position representations is an ongoing research topic.

Transformers as Language Models:

How to deploy them as language models via semi-supervised learning?

Similar technique as in RNN – given a training corpus of plain text we'll train a model to predict the next word in sequence using teacher forcing.

General approach: fig 9.21

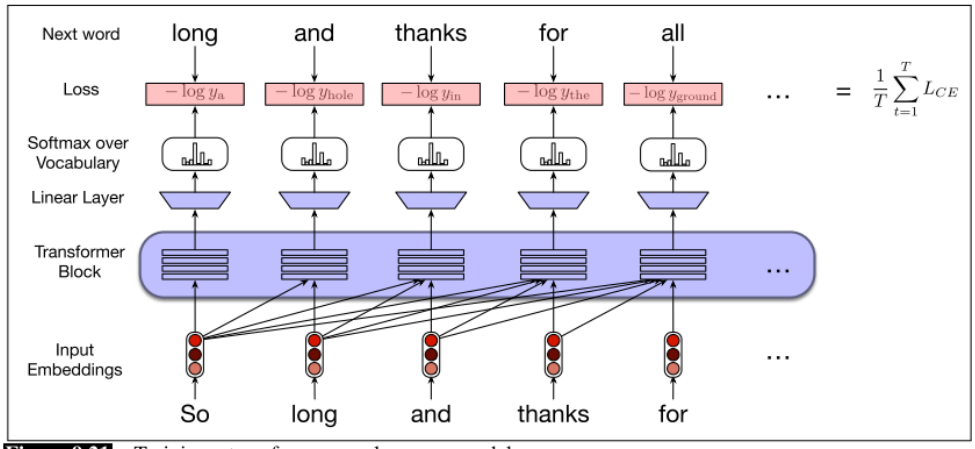


Figure 9.21 Training a transformer as a language model.

- At each step, given all the preceding words, the final transformer layer produces an output distribution over the entire vocabulary.
- During training, the probability assigned to the correct work is used to calculate the cross-entropy loss for the each item in the sequence.
- For RNNs – the loss for training seq – avg cross-entropy loss over the entire sequence.
- Key difference between img 9.21 and 9.6 is:
    - There the calculate of output and losses at each step was inherently serial given the recurrence in the calculation of hidden states.
    - Here with T- each training item can be processed in parallel since the output for each element in the sequence is computed separately.

Contextual Generation and Summarization

Simple variation on autoregressive generation – uses a prior context to prime the autoregressive generation process. Fig 9.22 : task of text completion.

- A standard LM is given the prefix to some text and is asked to generate a possible completion to it.
- As generation proceeds, the model has direct access to the priming context as well as to all of its own subsequently generated outputs.
- Ability to incorporate the entirety of the earlier context and generated outputs at each time step is the key to the power of these models.

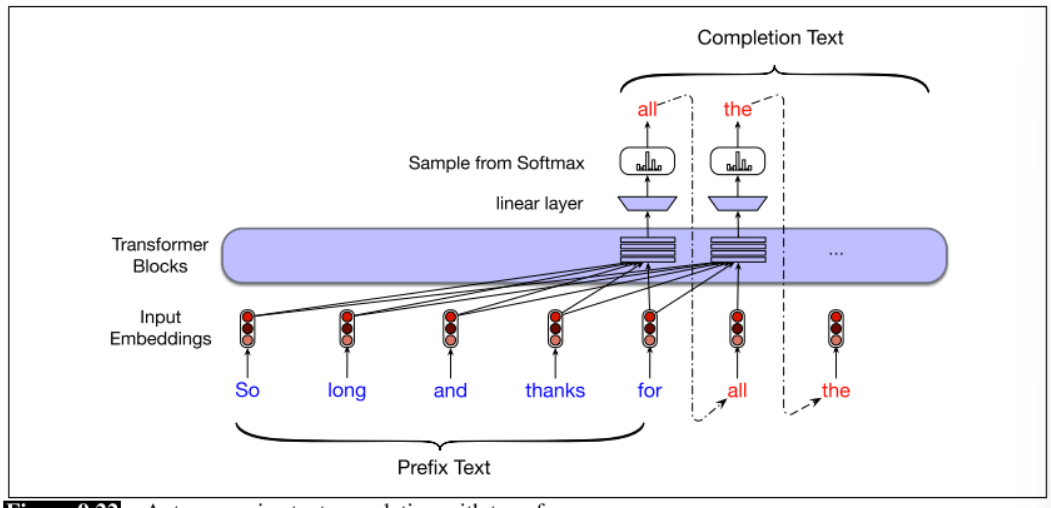


Figure 9.22 Autoregressive text completion with transformers.

Text Summarization: practical application of context-based autoregressive generation. It involves taking full length article and produce an effective summary of it.

Training process – start with corpus consisting of full-length articles accompanied by their corresponding summaries.

Another effective technique of applying transformers to summarization is to append the summary to each full-length article in a corpus, with a unique marker separating the two.
Ie. each article-summary pair (x1, ...,xm), (y1, ...,yn) in a training corpus is converted into a single training instance (x1, ...,xm,δ,y1, ...yn) with an overall length of n+m+1.

These training instances are treated as long sentences and then used to train an autoregressive LM using teacher forcing, *exactly as we did earlier*.

Once trained, full articles ending with the special marker are used as context to prime the generation process to produce a summary as shown in fig 9.24.
In contrast to RNN, the model has access to the original article as well as to the newly generated text throughout the process.

Applying Transformers to other NLP tasks

T – can be used for sequence labelling tasks, and sequence classification tasks.

Original Article

The only thing crazier than a guy in snowbound Massachusetts boxing up the powdery white stuff and offering it for sale online? People are actually buying it. For $89, self-styled entrepreneur Kyle Waring will ship you 6 pounds of Boston-area snow in an insulated Styrofoam box – enough for 10 to 15 snowballs, he says.
But not if you live in New England or surrounding states. "We will not ship snow to any states in the northeast!" says Waring's website, ShipSnowYo.com. "We're in the business of expunging snow!"
His website and social media accounts claim to have filled more than 133 orders for snow – more than 30 on Tuesday alone, his busiest day yet. With more than 45 total inches, Boston has set a record this winter for the snowiest month in its history. Most residents see the huge piles of snow choking their yards and sidewalks as a nuisance, but Waring saw an opportunity.
According to Boston.com, it all started a few weeks ago, when Waring and his wife were shoveling deep snow from their yard in Manchester-by-the-Sea, a coastal suburb north of Boston. He joked about shipping the stuff to friends and family in warmer states, and an idea was born. His business slogan: "Our nightmare is your dream!" At first, ShipSnowYo sold snow packed into empty 16.9-ounce water bottles for $19.99, but the snow usually melted before it reached its destination...

Summary

Kyle Waring will ship you 6 pounds of Boston-area snow in an insulated Styrofoam box – enough for 10 to 15 snowballs, he says. But not if you live in New England or surrounding states.

Figure 9.23 Examples of articles and summaries from the CNN/Daily Mail corpus (Hermann et al., 2015b), (Nallapati et al., 2016).

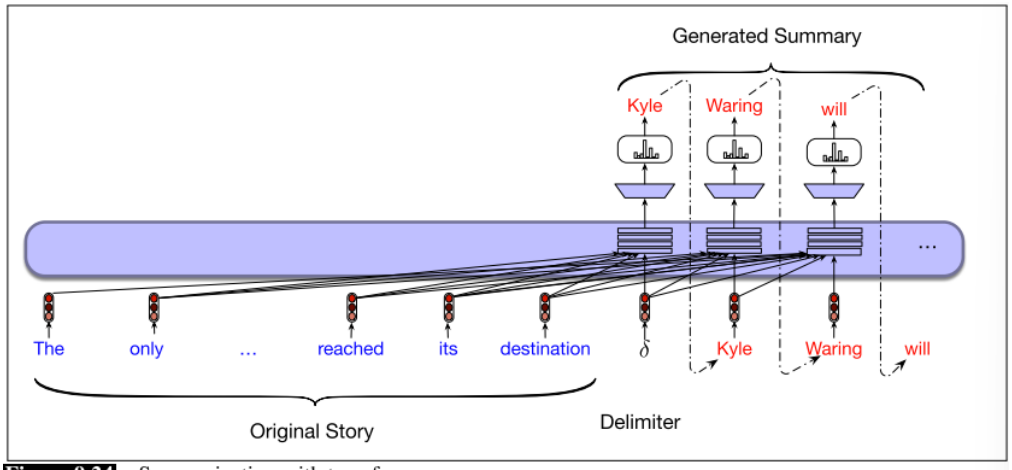


Figure 9.24 Summarization with transformers.

We don't train the raw transformer on these tasks, we use the technique – **pretraining**.

- Here, we train the transformer LM on large text corpus, in a normal self-supervised way, then add a linear/feedforward layer on the top that we **finetune** on a smaller dataset hand-labelled with POS/Sentiment labels.
- Pretraining on large amount of data via self supervised LM objective turns out to be a very useful way of incorporating rich information about language, and resulting representations make it much easier to learn from the generally smaller supervised datasets for tagging or sentiment.

9.10 Summary

This chapter has introduced the concepts of recurrent neural networks and transformers and how they can be applied to language problems. Here's a summary of the main points that we covered:

- In simple Recurrent Neural Networks sequences are processed one element at a time, with the output of each neural unit at time t based both on the current input at t and the hidden layer from time $t-1$.
- RNNs can be trained with a straightforward extension of the backpropagation algorithm, known as **backpropagation through time** (BPTT).
- Simple recurrent networks fail on long inputs because of problems like **vanishing gradients**; instead modern systems use more complex gated architectures such as **LSTMs** that explicitly decide what to remember and forget in their hidden and context layers.
- Transformers are non-recurrent networks based on **self-attention**. A self-attention layer maps input sequences to output sequences of the same length, based on a set of attention heads that each model how the surrounding words are relevant for the processing of the current word.
- A transformer block consists of a single attention layer followed by a feed-forward layer with residual connections and layer normalizations following each. Transformer blocks can be stacked to make deeper and more powerful networks.
- Common language-based applications for RNNs and transformers include:
  - Probabilistic language modeling: assigning a probability to a sequence, or to the next element of a sequence given the preceding words.
  - Auto-regressive generation using a trained language model.
  - Sequence labeling like part-of-speech tagging, where each element of a sequence is assigned a label.
  - Sequence classification, where an entire text is assigned to a category, as in spam detection, sentiment analysis or topic classification.

Machine Translation and Encoder Decoder Models

- MT systems are routinely used to produce a draft translation that is fixed up in a **post-editing** phase by a human translator. This task is often called **computer-aided translation** or CAT.
- CAT is commonly used as part of **localization**: the task of adapting content or a product to a particular language community
- Standard algo for MT→ **encoder-decoder** network. (**Sequence to sequence** n/w)
- We know RNN or T achitectures can be used for classification or seq. labelling.
- Encoder-decoder or sequence-to-sequence models are used for a different kind of sequence modeling in which the output sequence is a complex function of the entire input sequencer; we must map from a sequence of input words or tokens to a sequence of tags that are not merely direct mappings from individual words.
- EC-DC nw are very successful at handling these sort of complicated cases of sequence mappings where information is represented in different languages and each has its own set of rules and representation formats.
- ENDC is not just only for MT, it's the stat eof the art for many other tasks where complex mappings b/w 2 sequences are involved. These include Summarization, dialogue, semantic parsing etc.

Language Divergences and Typology

- Some aspects of human language seem to be **universal**, holding true for every language, or are statistical universals, holding true for most languages.
- Yet languages also **differ** in many ways, and an understanding of what causes such **translation divergences** will help us build better MT models.
- We often distinguish the idiosyncratic and lexical differences that must be dealt with one by one (the word for "dog" differs wildly from language to language), from systematic differences that we can model in a general way (many languages put the verb before the direct object; others put the verb after the direct object).
- The study of these systematic cross-linguistic similarities and differences is called linguistic typology. WALS, the World Atlas of Language Structures, which gives many typological facts about languages.

Word Order Typology

- German, French, English, and Mandarin, for example, are all **SVO** (**Subject-Verb-Object**) languages, meaning that the verb tends to come between the subject and object.
- Hindi and Japanese, by contrast, are **SOV** languages, meaning that the verb tends to come at the end of basic clauses, and Irish and Arabic are **VSO** languages.
- Two languages that share their basic word order type often have other similarities.

- For example, **VO** languages generally have prepositions, whereas **OV** languages generally have postpositions.

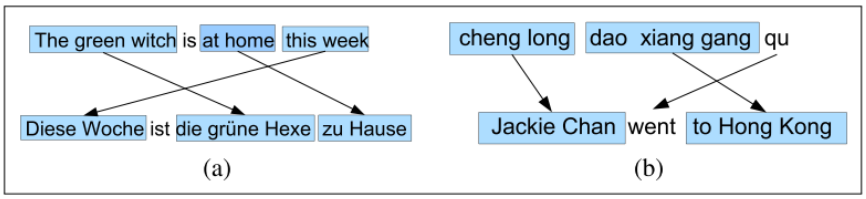


Figure 10.1 Examples of other word order differences: (a) In German, adverbs occur in initial position that in English are more natural later, and tensed verbs occur in second position. (b) In Mandarin, preposition phrases expressing goals often occur pre-verbally, unlike in English.

Lexical Divergences

- Different words have different meaning under different contexts. So, MT and Word Sense Disambiguation are closely linked.
- **Lexical gap**: while translation when the later language has no words to describe the exact meaning for first language. E.g. Mandarin xi`ao or Japanese oyak¯ok¯oo (in English one has to make do with awkward phrases like filial piety or loving child, or good son/daughter for both)
- English: *The bottle floated out*.
  Spanish: *La botella salio floatando* (The bottle exited floating)

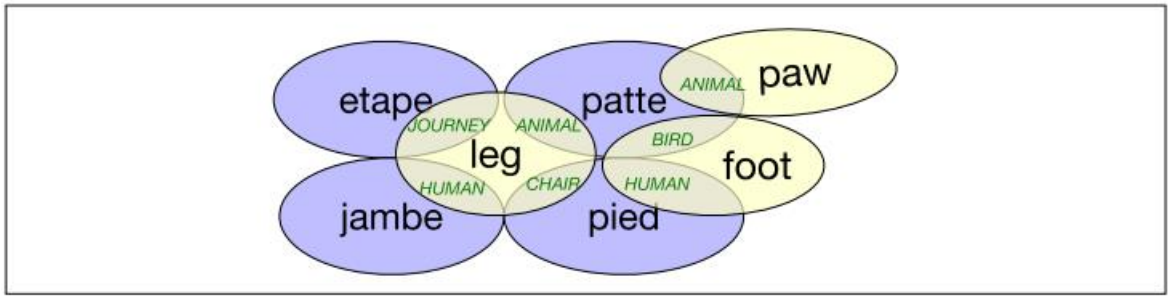


Figure 10.2 The complex overlap between English *leg*, *foot*, etc., and various French translations as discussed by Hutchins and Somers (1992).