Introduction to Natural Language Processing

Transcript of I. to N.L.P by Prof. Lucie Flek

2023.06.09

1 Introduction

1.1 Goals of this course

- Learn the basic concepts of Natural Language Processing
- Learn techniques and tools used in practice to provide NLP-based applications for the web and other document collections
- Gain insight into open research problems in natural language processing
- Understand the hype...

1.2 Content of this course

- **Language Processing & ML Basics:** Getting everyone onto the same page with the fundamentals of text processing & Python
- **Word features, embeddings:** Approaches to text classification that ignore linguistic structure within a sentence or document
- **Meaning in context:** Techniques (neural and non-neural) that model sentences as sequences of words - language modeling, part-of-speech tagging, NER... (+ a closer look into the large language models)
- **Hierarchical Sentence Structure:** Tree-based models of sentences that capture grammar and relationships within a sentence
- **Applications:** Overviews of language technologies for text such as argument mining, summarization or reasoning

1.3 Formulary

Chain Rule:

$$\frac{\partial}{\partial x}\left[f\left(g(x)\right)\right] = f'\left(g(x)\right) g'(x) \tag{1}$$

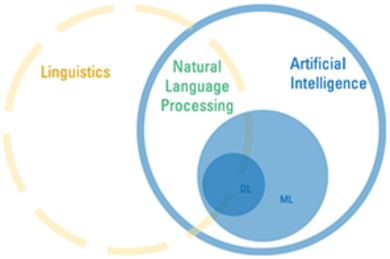
1.4 Study material

- **Introduction to Natural Language Processing**, Jacob Eisenstein
  https://github.com/jacobeisenstein/gt-nlp-class/blob/master/notes/eisenstein-nlp-notes-10-15-2018.pdf
- **Speech and Language Processing**. An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Daniel Jurafsky and James H. Martin.
  https://web.stanford.edu/~jurafsky/slp3/
- **Natural Language Processing with Python**
  https://www.nltk.org/book/

2 What is Natural Language Processing

- **Analysis**: (or "understanding" or "processing" ...): input is language, output is some representation that supports useful action
- **Generation**: input is that representation, output is language
- **Acquisition**: obtaining the representation and necessary algorithms, from knowledge and data

Is NLP like AI? NLP lies at the intersection of computational linguistics and artificial intelligence. NLP is (to various degrees) informed by linguistics.



2.1 What makes NLP difficult?

Language is ambiguous. Given the example: "One morning I shot an elephant in my pajamas", we can not say for certain, whether the shot elephant was wearing the pajamas of the shouter, or the shouter was in pajamas himself. A single sentence can be structurally interpreted in different ways with varying results.

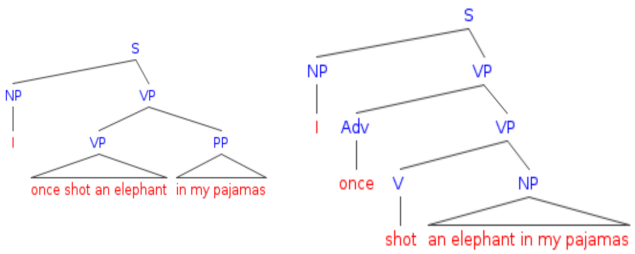


Figure 1: Different structurally interpretations.

Ambiguity has many levels

- **Word senses**: bank (finance or river?)
- **Part of speech**: chair (noun or verb?)
- **Syntactic structure**: I saw a man with a telescope
- **Quantifier scope**: Every child loves some movie
- **Multiple**: I saw her duck
- **Reference**: John dropped the goblet onto the glass table and it broke.
- **Discourse**: The meeting is canceled. Nicholas is not coming to the office today

Besides the ambiguity of language, language keeps on changing. And with it the meaning of words. A few prominent examples in the figure below.

Figure 2: https://nlp.stanford.edu/projects/histwords/

Language is noisy. If we look at how written language is used these days, you quickly realize, there is a lot of noise in language. Here are some examples: “amirite”, “wat did u say”, “c u ton8”, “goooooood!”.
Suppose we train a part of speech tagger on the Wall Street Journal and give it the input:
Mr. Vinken is chairman of Elsevier, the Dutch publishing group.

Figure 3: https://corenlp.run/

What will happen if we try to use this tagger for social media?
ikr smh he asked fir yo last name

Language is rich in expressivity, there a different forms to express the same meaning. Here some examples:

- She gave the book to Tom. *vs.* She gave Tom the book.
- Some kids popped by. *vs.* A few children visited.
- Is that window still open? *vs.* Please close the window.

Languages requires background knowledge of the world. If olive oil is made of olives, what is baby oil made of? If we just follow the pattern, we might think baby oil is made of babies, clearly it is made for not from babies.

2.2 What does an NLP system need to know?

Language consists of many levels of structure that are obvious to human. When we take apart a simple sentence, we are confronted with a abundance of linguistic information we use naturally in everyday life.

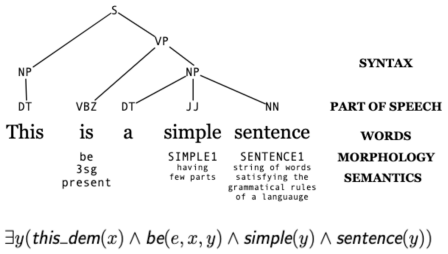


Figure 4: Taken of 'Introduction to Natural Language Processing' (Prof. Lucie Flek, UMR)

NLP systems often learn this by statistics. One basic rule here is, words in similar context are similar in meaning.

3 Splitting and counting words

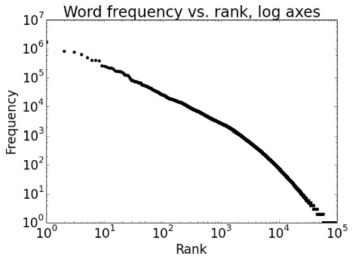
Given the sentence, "I took a pen and I wrote a message." we count 9 word tokens and 7 word types. If we look at European parliament speeches word count, we observe 24 million word tokens with 93638 word types.
When looking at our word counts we notice a logarithmic distribution (Zipf law). Many words are rare, few words are very frequent. The number differs between languages, but the shape is similar.
Regardless of how large our data is, there will be a lot of infrequent (and zero-frequency) words. This means we need to find good ways to estimate probabilities for things we have rarely or never seen during training a machine learning model.

any word

Frequency	Type
1,698,599	the
849,256	of
793,731	to
640,257	and
508,560	in
407,638	that
400,467	is
394,778	a
263,040	I

nouns

Frequency	Type
124,598	European
104,325	Mr
92,195	Commission
66,781	President
62,867	Parliament
57,804	Union
53,683	report
53,547	Council
45,842	States



3.1 Tokenization

Tokenization is the process of splitting sentences to tokens (word-like segments). Given the sentence "Good muffins cost $3.99 in New York. Please buy me two of them.", can't we just use `text.split()`. Is "." a token? Is "$3.88" one token? Is "New York" one token? Real data contain noise: HTML code, URL links, misspellings, punctuation!!!, smileys... therefor simply using python's split-method.
Tokenizers are usually regex-based or machine-learned.

3.2 Social Media Tokenization

If you work with social media, you may need special tools. Sentences typically don't have a full stop at the end, and words often have weird characters (# , @, ...) included, that you may want to keep together with the word. One example of such a special tokenization treatment for Twitter, here:
`sentiment.christopherpotts.net/code-data/happyfuntokenizing.py`

One special tokenizer of the above mentioned script, here in detail:

```
emoticon_string = r"""
    (?:
      [<>]?
      [:;=8]                     # eyes
      [\-o\*\']?                 # optional nose
      [\)\]\(\[dDpP/\:\}\{@\|\\] # mouth
      |
      [\)\]\(\[dDpP/\:\}\{@\|\\] # mouth
      [\-o\*\']?                 # optional nose
      [:;=8]                     # eyes
      [<>]?
    )"""
```

Listing 1: Emoticon tokenizer

3.3 Stemming and Lemmatization

Many languages have some inflectional (Wikipedia: Inflection is a process of word formation in which a word is modified to express different grammatical categories such as tense, case, voice, aspect, person, number, gender, mood, animacy, and definiteness.) and derivational morphology, where similar words have similar forms.
There are multiple ways of stemming and lemmatization.
Known as **Porter Stemmer** is the heuristic process for chopping off the inflected suffixes of a word, defined by a sequence of rules for removing suffixes from words. It has low precision by high recall.

organizes, organized, organizing... → organ
programs, programmer, programming... → program

- EMENT → ∅
- SSES → SS
- IES → I
- SS → ∅
- S → ∅

Another lemmatizer is the **Stanford Lemmatizer**. This lemmatizer uses morphological analysis to return the dictionary form of a word (the entry in

a dictionary, under which you would find all forms). This is more linguistically accurate, but slower to implement & use.

organizes, organized, organizing... → organize
corpora → corpus
better → good
rocks → rock

What about sentence boundaries? Is it enough to split sentences at punctuation? A few simple examples like "He lives in the U.S. John, however, lives in Canada." or "Mr. Collins said he was going." make apparent, that it is not.
Sentence segmentation tools can have various levels of complexity (and speed), here are three option in SpaCy:

https://spacy.io/api/sentencerecognizer
https://spacy.io/api/sentencizer
https://spacy.io/usage/linguistic-features#sbd

3.4 Scikit-Learn: Understanding CountVectorizer

The CountVectorizer is specifically used for counting words. The vectorizer, of CountVectorizer is (technically speaking!) the process of converting text into some sort of number-y thing that computers can understand.

```
from sklearn.feature_extraction.text import CountVectorizer

# Build the text
text = """The CountVectorizer is specifically used for
counting words. The vectorizer part of CountVectorizer
is (technically speaking!) the process of converting
text into some sort of number-y thing that computers
can understand."""
# Initialize the vectorizer
vectorizer = CountVectorizer()
# Create word matrix
matrix = vectorizer.fit_transform([text])
# get all found words, listed lexicographically sorted
found_words = vectorizer.get_feature_names_out()
```

Listing 2: CountVectorizer()

```
['can' 'computers' 'converting' 'counting' 'countvectorizer'
'for' 'into' 'is' 'number' 'of' 'part' 'process' 'some'
'sort' 'speaking' 'specifically' 'technically' 'text' 'that'
'the' 'thing' 'understand' 'used' 'vectorizer' 'words']
```

Listing 3: found words

4 Linear Text Classification

A mapping function let's call it $h(x)$, which takes some input data x from the space of all input data X and maps it to some label y from the output label space Y. Given the example of classifying the language of an document, the variables would be the following:

- X - The set of all documents.
- Y - The set of all languages (English, German, Greek,...).
- x - A specific document.
- y - One specific language label (e.g. Greek).
- $h(x)$ - Our learned classification criteria.

We may never know how the perfect mapping $h(x) = y$ looks like. But we try to approximate it by observing the input and output examples. So we get some $h'(x)$ which is close to $h(x)$. Given the example

$$h('\mu\eta\nu\iota\nu\alpha\epsilon\iota\delta\epsilon\theta\epsilon\alpha') = \text{Greek}$$

We can define a rule, "anything with Greek letters should be classified as Greek" (until we see a counter-example), manually. Given training data in the form of ¡x, y¿ pairs, the mapping function

$$h'(x) = y$$

which is as close as it gets, to the unknown ideal function

$$h(x) = y$$

is learning how to arrive at the wanted outcome y, given the input x. This method of creating a model is known as **supervised machine learning** (the labels of y for the training samples x are known). **Unsupervised machine learning** is when the training samples x have no labels y provided.

Example classification tasks

task	X	y
language ID	text	(English, Mandarin, Greek,...)
spam classification	email	(spam, not spam)
authorship attribution	text	(J.K. Rowling, James Joyce,...)
genre classification	novel	(detective, romantic,...)
sentiment analysis	text	(positive, negative, neutral, mixed)

The mapping function (classification function) $h'(x)$ we want to learn, is influenced by two components:

- The formal structure of the learning method h. (What is the relationship between input and output?)
  $\rightarrow$ Naive Bayes, logistic regression, SVM (Support Vector Machines),...
- The representation of the data x. (How can I express the distinctive characteristics of a document into some numbers?)

4.1 Representing text data

One (usual and simple) approach is the **bag-of-words**:
Your input text ω is a sequence of words $\omega = (\omega_1, \omega_2, \omega_3, \ldots \omega_n) \in V$.

$$\omega = \underset{\omega_1}{\text{The}}\ \underset{\omega_2}{\text{drinks}}\ \underset{\omega_3}{\text{were}}\ \underset{\omega_4}{\text{strong}}\ \underset{\omega_5}{\text{but}}\ \underset{\omega_6}{\text{the}}\ \underset{\omega_7}{\text{fish}}\ \underset{\omega_8}{\text{tacos}}\ \underset{\omega_9}{\text{were}}\ \underset{\omega_{10}}{\text{bland.}}$$

This is represented by word counts $x = (x_1, x_2, x_3, \ldots x_m) \in X, m = |V|$.
Note that this requires collecting the full possible vocabulary V ahead.

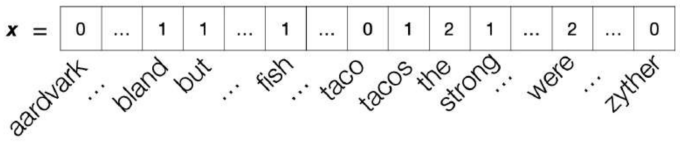


Figure 5: Taken of 'I. to N.L.P.' (Prof. Lucie Flek, UMR)

There are many other options to represent the data:

- Count only positive/negative words found in sentiment dictionaries.
- Only count individual words in isolation ("bag of words").
- Count sequences of words (sequential word pairs - bigrams, trigrams,..., skip n-grams, other non-linear combinations).
- Higher order linguistic structure (e.g. syntactic dependencies).

At the end, however you define your features, you will end up with a sequence of numbers (a vector) representing your document. This should have the same length for all documents, as you always look at the presence/absence (or count) of the same features.

4.2 Linear text classification

We need a scoring function of each document $x = (x_1, \ldots, x_n)$ for each possible label y_k. This scoring function is a linear combination of document features and their learned weights about how important these feature are.

$$\psi(x, y) = \theta \cdot f(x, y) = \sum_{j=1} \theta_j \times f_j(x, y) \tag{2}$$

$$\hat{y} = \underset{y}{\operatorname{argmax}}\, \psi(x, y) \tag{3}$$

The length of the weight vector θ is the size of the vocabulary V. The feature functions $f_j(x, y)$ are usually just copied feature values if the document belongs to the class.

$f(x, y = 1) =$

x_0	x_1	$\cdots$	$x_{\lvert V\rvert}$	0	0	0	0	0

$f(x, y = 2) =$

0	0	0	$\cdots$	0	x_0	x_1	$\cdots$	$x_{\lvert V\rvert}$

	$\mathbf{f_j(x, y)}$
loved	0
genius	1
not	0
fruit	1
loved	0
genius	0
not	0
fruit	0

	$\theta_\mathbf{j}$
loved	0.01
genius	0.03
not	1.4
fruit	3.1
loved	1.2
genius	0.5
not	-3.0
fruit	-0.8

For a slower intro into this part, read the chapter from the Jacob Eisenstein's book.

4.3 Some Main Linear Text Classifiers

4.3.1 Naïve Bayes

Let's say $p(x, y)$ is a joint probability of the document bag of words x and it's true label y.
We have a data set of N documents, about which we assume they are independent and coming from the same probability distribution of features &

labels.
We want to maximize the joint probability of the whole data set. I. e. the probability that each document gets assigned the true label.

Then the joint probability of the whole data set is

$$p(x^{1:N}, y^{1:N}) = \prod_{i=1}^{N} p(x^{(i)}, y^{(i)}) \tag{4}$$

And using the Bayes rule, we can replace the joint probabilities of individual documents and their true labels by

$$p(x, y) = p(x|y) \times p(y) \tag{5}$$

Where we estimate the $p(y)$ of the labels by their frequency in the training set

$$p(y) = \text{Categorical}(\mu)$$

And the probabilities of words conditioned by the labels

$$p(x|y) = \text{Multinominal}(\phi, T)$$

The multinomial distribution is a generalization of the binomial distribution. For example, it models the probability of counts for each side of a j-sided die rolled n times.

$$P_{mult}(x, \phi) = B(x) \prod_{j=1}^{V} \phi_j^{x-j} \tag{6}$$

$$B(x) = \frac{(\sum_{j=1}^{V} x_j)!}{\prod_{j=1}^{V}(x_j!)} \tag{7}$$

Where ϕ is the probability that any given token in a document is the word j and $B(x)$ is the number of possible orderings of the vector x. We learned that the choice of class label is done as

$$\hat{y} = \underset{y}{\text{argmax}}\, \psi(x, y)$$

Based on a scoring function ψ of a document (represented by a feature vector x) for each label y_i. And that the scoring function for linear classifier is

$$\psi(x, y) = \theta \cdot f(x, y) = \sum_{j=1} \theta_j \times f_j(x, y)$$

So we set the scoring function as the joint probability of the document and the label

$$\psi(x, y) = \log p(x, y) = \log p(y|x) + C$$
$$\psi(x, y) = \theta \cdot f(x, y) = \log p(x|y) + \log p(y)$$

$\theta =$ 

With θ, $K \times (V+1)$ long. We want to maximize the joint probability of the whole data set. I.e. the probability that each document gets assigned the true label (**Maximum Likelihood Estimation**)

$$\hat{\phi}, \hat{\mu} = \underset{\phi,\mu}{\operatorname{argmax}} \prod_{i=1}^{N} p(x^{(i)}, y^{(i)}) = \underset{\phi,\mu}{\operatorname{argmax}} \sum_{i=1}^{N} \log p(x^{(i)}, y^{(i)}) \tag{8}$$

Count the frequencies of everything

$$\hat{\phi}_{y,j} = \frac{\operatorname{count}(y,j)}{\sum_{j'=1}^{V} \operatorname{count}(y,j')} = \frac{\sum_{i:y^i=y} x_j^{(i)}}{\sum_{j'=1}^{V} \sum_{i:y^i=y} x_{j'}^{(i)}}$$

$$\hat{\mu}_y = \frac{\operatorname{count}(y)}{\sum_{y'} \operatorname{count}(y')}$$

Here an example on predicting spam with Naïve Bayes https://www.youtube.com/watch?v=02L2Uv9pdDA

4.3.2 Perceptron

Naïve Bayes ignores dependencies between words, and treats each word as equally informative. To avoid this, we can use classifiers that are not attempting to model the generative probability $p(x)$. Perceptron is one of such discriminative classifiers.
The basic idea is to run a current (some, imperfect, maybe randomly initialized) classifier on an instance in the training data, obtaining some prediction

$$\hat{y} = \underset{y}{\operatorname{argmax}}\, \phi(x, y)$$

If the prediction is incorrect

- Increase the weights for the features of the true label $y^{(i)}$
- Decrease the weights for the features of the predicted label $\hat{y}$
- Repeat until all training instances are correctly classified (or nearly)

$$\theta \leftarrow \theta + f(x^{(i)}, y^{(i)}) - f(x^{(i)}, \hat{y}) \tag{9}$$

If there is some θ that correctly labels all the training instances, this method is guaranteed to find it. In that case the data set is **linearly separable**.
Updating weights requires minimizing a **loss function** on the weights.

$$\ell_{\text{perceptron}}(\theta; x^{(i)}, y^{(i)}) = -\theta \cdot f(x^{(i)}, y^{(i)}) + \max_{y' \neq y^{(i)}} \theta \cdot f(x^{(i)}, y') \tag{10}$$

One way to minimize the loss function is **gradient descent**

$$\theta^{t+1} \leftarrow \theta^t - \eta \frac{\partial}{\partial \theta} \sum_{i=1}^{N} \ell(\theta^{(t)}; x^{(i)}, y^{(i)}) \tag{11}$$

Where η is the learning rate.
Stochastic gradient descent is a cheaper proxy of gradient descent, which does not look at all instances of the training set, but just one.

$$\frac{\partial}{\partial \theta} \sum_{i=1}^{N} \ell(\theta^{(t)}; x^{(i)}, y^{(i)}) \approx C \times \frac{\partial}{\partial \theta} \ell(\theta^{(t)}; x^{(i)}, y^{(i)}) \tag{12}$$

In practice, a mini batch gradient descent is used, looking at several instances at the same time.
Both Naïve Bayes and Perceptron are relatively easy to optimize, yet Perceptron requires iterating over the data set multiple times.
Naïve Bayes can suffer infinite loss on a single example, since the logarithm of 0 probability is $-\infty$; some examples will be over-emphasized, others will be under-emphasized. Naïve Bayes assumes the observed features are conditionally independent given the label; performance depends on the extent to which this holds.

Perceptron treats all correct answers equally, even if θ only gives the correct answer by a very small margin, the loss is still 0.

4.3.3 Support Vector Machines (SVM)

Support Vector Machines (SVMs) are a type of supervised learning algorithm that can be used for classification or regression tasks. The main idea behind SVMs is to find a hyperplane that maximally separates the different classes in the training data. This is done by finding the hyperplane that has the largest margin, which is defined as the distance between the hyperplane and the closest data points from each class. Once the hyperplane is determined, new data can be classified by determining on which side of the hyperplane it falls. SVMs are particularly useful when the data has many features, and/or when there is a clear margin of separation in the data.

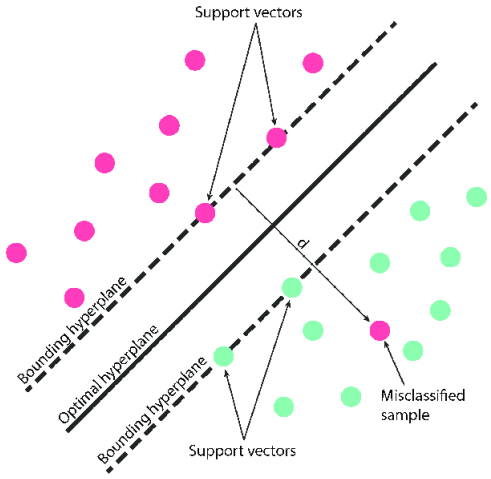


Figure 6: Hyperplane dividing a set of samples

For better generalization, the correct label should outscore all other labels by a **large margin**.

$$\gamma(\theta; x^{(i)}, y^{(i)}) = \theta \cdot f(x^{(i)}, y^{(i)}) - \max_{y' \neq y^{(i)}} \theta \cdot f(x^{(i)}, y') \tag{13}$$

Maximizing the margin is the same as minimizing the margin loss

$$\max\left(0, 1 - \gamma(\theta; x^{(i)}, y^{(i)})\right) \tag{14}$$

(loss is high if the margin is small – penalizing small margins)

4.3.4 Logistic Regression

Naïve Bayes is **probabilistic**, it assigns calibrated confidence scores to its predictions. Perceptron and SVMs are **discriminative** they learn to discriminate correct and incorrect labels. Logistic regression is both discriminative and probabilistic. The idea is the direct computation of the conditional probability of the label $p(y|x)$.

$$p(y|x; \theta) = \frac{\exp(\theta \cdot f(x, y))}{\sum_{y' \in \mathcal{Y}} \exp(\theta \cdot f(x, y'))} \tag{15}$$

The exponential is needed to make the score a positive number. Logit (the division by sum of exponentials) is needed to normalize to 0-1 in order to make the score a probability.

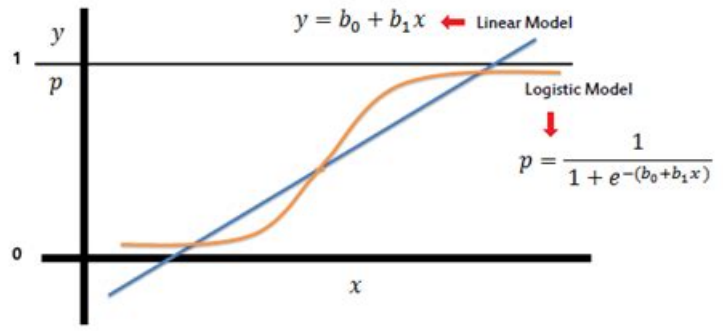


Figure 7: Comparison; linear & logistic model

	BIAS	love	loved
β	-0.1	3.1	1.2

$$\prod_i^N P((y_i|x_i, \beta)$$

	BIAS	love	loved	$a = \sum x_i \beta_i$	$\exp(-a)$	$1/(1 + \exp(-a))$
x^1	1	1	0	3	0.05	95.2%
x^2	1	1	1	4.2	0.015	98.5%
x^3	1	0	0	-0.1	1.11	47.4%

There are two equivalent views of logistic regression learning. The **maximization** of the conditional log-likelihood

$$\begin{aligned} \log p(y^{(1:N)}|x^{(1:N)}) &= \sum_{i=1}^{N} \log p(y^{(i)}|x^{(i)}; \theta) \\ &= \sum_{i=1}^{N} \theta \cdot f(x^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp\left(\theta \cdot f(x^{(i)}, y')\right) \end{aligned} \tag{16}$$

and the **minimization** of the logistic loss

$$\ell_{\text{LogReg}}(\theta; x^{(i)}, y^{(i)}) = -\theta \cdot f(x^{(i)}, y(i)) + \log \sum_{y' \in \mathcal{Y}} \exp\left(\theta \cdot f(x^{(i)}, y')\right) \tag{17}$$

Comparison of Linear Classifiers

Classifier	Pros	Cons
Naïve Bayes	Simple, fast, probabilistic, can be strong with well-crafted aggregate features	Not very accurate, not too well suited for textual data with a large number of highly correlated features
Perceptron	Simple, usually quite accurate	Not probabilistic – any good decision is equally good, needs iterations, may over fit the data
Support Vector Machines	Error-driven learning, usually can deal well with high-dimensional textual data	Not probabilistic
Logistic Regression	Error-driven learning, regularized	Trickier to implement

5 Non-Linear Text Classification: Neural Networks

We remember for **linear text classification**, we need a scoring function of each document $x = (x_1, \ldots)$ for each possible label y_k. This scoring function is a linear combination of document features and their learned weights about how important these feature are. θ forms the to be learned weight, depending on the classifier. $f(x, y)$ are the feature vectors representing the words in an input document, depending on feature representation.
These weights can be learned in multiple ways. Naïve Bayes, Perceptron, large-margin and logistic regression to name a few.

We want to have a closer look at the Perceptron, to do so we simplify our notation for a moment. Let's say $x = (x_1, \ldots)$, a vector of numbers, is our input. $\omega = (\omega_1, \ldots)$, again a vector of numbers, are our weights. $z = \omega \cdot x$.

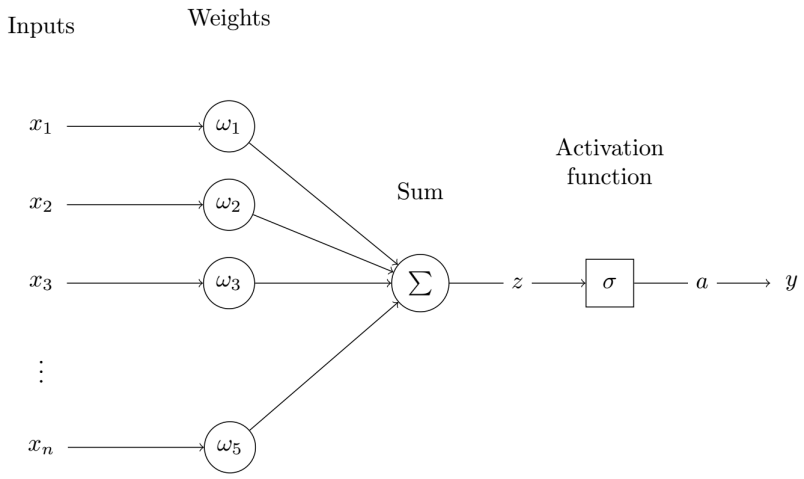


Figure 8: Perceptron; simplified

With

$$z = \omega \cdot x + b = b + \sum_i \omega_i \cdot x_i \tag{18}$$

and

$$y = a = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{19}$$

The output is a nonlinear transformation of a linear combination of input times weights (and a bias term) b.
Let's say we have the example "Will Max play tennis on the weekend, when the weather is good?". From that we take as input:

- x_1: Weekend? yes = 1, no = 0

- x_2: Weather good? yes = 1, no = 0

And the weights:

- ω_1: 6 (Max says it's important)
- ω_2: 2 (Max says it's not important)

We evaluate $z = x_1 \cdot \omega_1 + x_2 \cdot \omega_2$ with $x = (x_1, x_2) = (1, 1)$ and $\omega = (\omega_1, \omega_2) = (6, 2)$. So we have $z = 6 \cdot 1 + 2 \cdot 1 = 8$. Depending on the threshold we have

$$output = \begin{cases} 0 if \ \sum_i \omega_i \cdot x_i < threshold \\ 1 if \ \sum_i \omega_i \cdot x_i \geq threshold \end{cases} \tag{20}$$

or

$$output = \begin{cases} 0 if \ \sum_i \omega_i \cdot x_i + bias < 0 \\ 1 if \ \sum_i \omega_i \cdot x_i + bias \geq 0 \end{cases} \tag{21}$$

Once we established our output, we can compare it with the actual results and determine whether the predicted value $\hat{y}_i$ is the same as the value y stored in the data set. If there is discrepancy between the two, we update our weights as described earlier. What is the loss? Given

$$E_j = \frac{1}{2} e_j^2 = \frac{1}{2}(d_j - y_j)^2 \tag{22}$$

with d_j being the output for each input.
We update our weights by subtracting the derivative of our error function (22) in the direction of the weights linking j^{th} input to j^{th} perceptron.

$$\omega_{ij(k+1)} = \omega_{ij(k)} - \eta \frac{\partial E_j}{\partial \omega_{ij}} \tag{23}$$

With $\omega_{ij(k)}$ being the current weight, $\omega_{ij(k+1)}$ being the updated weight and η being the step size. With

$$y_j = \frac{1}{1 + e^{-A_j}}$$

and

$$A_j = \sum_{i=1}^{n} \omega_{ij} \cdot x_i + \theta$$

Using the chain rule (1) we determine

$$\frac{\partial E_j}{\partial \omega_{ij}} = \frac{\partial E_j}{\partial e_j} \cdot \frac{\partial e_j}{\partial y_j} \cdot \frac{\partial y_j}{\partial A_j} \cdot \frac{\partial A_j}{\partial \omega_{ij}}$$

We calculate the derivatives as follows

Function	Derivative
$E_j = \frac{1}{2} e_j^2$	$\frac{\partial E_j}{\partial e_j} = \frac{\partial}{\partial e_j} \frac{1}{2} \left(e_j^2\right) = e_j$
$e_j = d_j - y_j$	$\frac{\partial e_j}{\partial y_j} = \frac{\partial}{\partial y_j} (d_j - y_j) = -1$
$y_j = \frac{1}{1 + e^{-A_j}}$	$\frac{\partial y_j}{\partial A_j} = \frac{\partial}{\partial A_j} \left(\frac{1}{1 + e^{-A_j}}\right) = [1 - y_j] y_j$

And with it arrive at

$$\frac{\partial E_j}{\partial \omega_{ij}} = -e_j [1 - y_j] \cdot y_j \cdot x_i$$

We choose the step size η accordingly, to minimize the loss of updating our weights.

The problem here is, perceptron is a linear classifier, it learns only to classify linear boundaries. A possible solution is the composition of multiple perceptron to a network.

5.1 Fully Connected Neural Networks

5.1.1 Feed Forward Networks

A feed forward network is essentially nothing else than stacked perceptrons. The layers of this structure are made of an Input-Layer x, n hidden layers $(z_1, \ldots, z_n)$ and the Output-Layer. The weights of one layer, can be interpreted as a matrix. The calculation of the output for one input vector can be seen as a series of matrix multiplications, interleaved with activation functions.

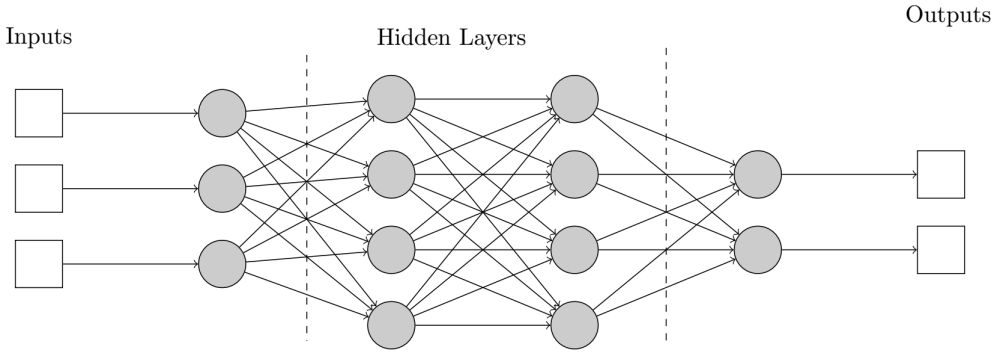


Figure 9: Neural Network; simplified

Suppose we want to label as y = good | bad | okay. What makes a good story? "Exciting plot, compelling characters, interesting setting, ..." Let's call this vector of features (hidden layer) z. z should be easy to learn from x (the words), and it should make it easy to predict the label y.

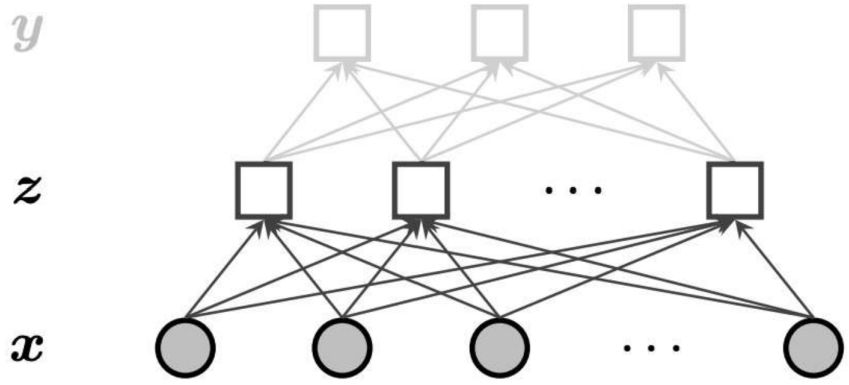


Figure 10: Feed Forward Network (I.t.N.L.P. Lucie Flek

It holds

$$Pr(z_k = 1|x) = \sigma\left(\theta_k^{(x\to z)} \cdot x\right) = \frac{1}{1 + e^{-\theta_k^{(x\to z)}} \cdot x} \tag{24}$$

We can predict y as follows

$$Pr(y = 1|z) = \frac{\exp\left(\theta_j^{(z\to y)} \cdot z + b_j\right)}{\sum_{j' \in \mathcal{Y}} \exp\left(\theta_{j'}^{(z\to y)} \cdot + b_{j'}\right)} \tag{25}$$

The weight in each layer will be a matrix of values. You can add more cascaded layer for more complexity. In some textbooks these are referred to as h, for hidden layers.

$$\Theta^{(x\to z)} = [\theta_1^{(x\to z)}, \theta_2^{(x\to z)}, \ldots, \theta_{K_z}^{(x\to z)}]^T \tag{26}$$

5.2 Training Neural Networks

How do we evaluate whether a network performs well? We define a **loss function** e.g. by

- y_i = actual output of the network
- $\hat{y}_i$ = expected output of the network
- N = size of training data

We want to find the weight, which give us the minimal loss. The idea is to use gradient descend (11) to change the weights in the direction towards the minimum of the loss function. But how do we find this minimum? We know, that the derivative of a function points to the direction of a local maximum / minimum. We build the derivative of the loss function with reference to the weights.

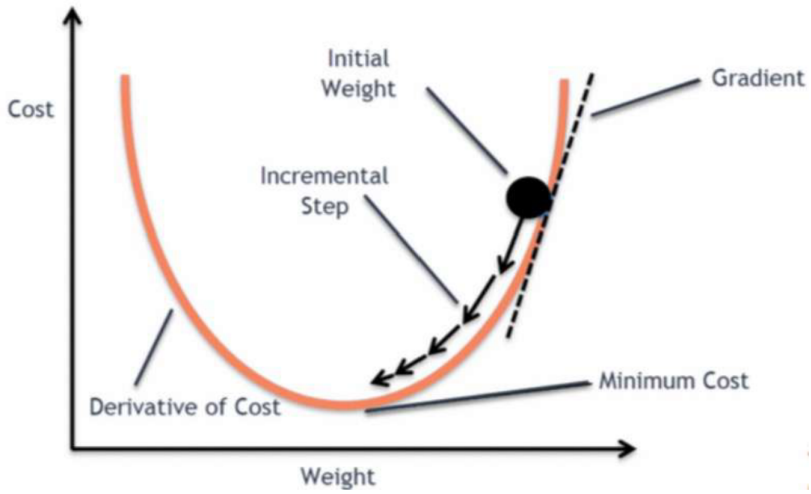


Figure 11: graph

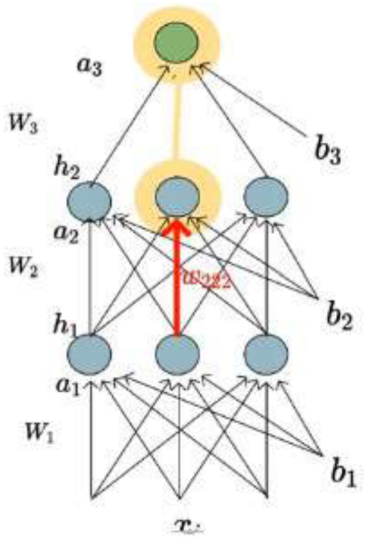
$$\theta_k^{(z \to y)} \leftarrow \theta_k^{(z \to y)} - \eta^{(t)} \nabla_{\theta_k^{(z \to y)}} \ell^{(i)} \tag{27}$$

where

- $\eta^{(t)}$ is the learning rate at update t
- $\ell^{(i)}$ is the loss on instance (mini batch) i
- $\nabla_{\theta_k^{(z \to y)}}$ is the gradient descent of the loss with respect to the output weights $\theta_k^{(z \to y)}$

$$\theta_k^{(z \to y)} = \left[\frac{\partial \ell^{(i)}}{\partial \theta_{k,1}^{(z \to y)}}, \frac{\partial \ell^{(i)}}{\partial \theta_{k,2}^{(z \to y)}}, \ldots, \frac{\partial \ell^{(i)}}{\partial \theta_{k,K_y}^{(z \to y)}} \right] \tag{28}$$

But if we don't observe z, how can we compute the weights? The answer is **Backpropagation**. We compute a loss on y, and apply the chain rule (1) from calculus to compute a gradient on all parameters. Applying the chain rule, means we are multiplying derivations (gradients), so we want theses gradients to have some reasonable properties.



Let's focus on the highlighted weight (ω_{222}). To learn this weight, we have to compute partial derivatives with relation to the loss function.

$$(\omega_{222})_{t+1} = (\omega_{222})_t - \eta \cdot \left(\frac{\partial L}{\partial \omega_{222}} \right) \tag{29}$$

$$\begin{aligned}\left(\frac{\partial L}{\partial \omega_{222}}\right) &= \left(\frac{\partial L}{\partial \alpha_{22}}\right) \cdot \left(\frac{\partial \alpha_{22}}{\partial \omega_{222}}\right) \\ &= \left(\frac{\partial L}{\partial h_{22}}\right) \cdot \left(\frac{\partial h_{22}}{\partial \alpha_{22}}\right) \cdot \left(\frac{\partial \alpha_{22}}{\partial \omega_{222}}\right) \\ &= \left(\frac{\partial L}{\partial a_{31}}\right) \cdot \left(\frac{\partial a_{31}}{\partial h_{22}}\right) \cdot \left(\frac{\partial h_{22}}{\partial a_{22}}\right) \cdot \left(\frac{\partial a_{22}}{\partial \omega_{222}}\right) \\ &= \left(\frac{\partial L}{\partial \hat{y}}\right) \cdot \left(\frac{\partial \hat{y}}{\partial a_{31}}\right) \cdot \left(\frac{\partial a_{31}}{\partial h_{22}}\right) \cdot \left(\frac{\partial h_{22}}{\partial a_{22}}\right) \cdot \left(\frac{\partial a_{22}}{\partial \omega_{222}}\right)\end{aligned}$$

The derivative of the Sigmoid function is

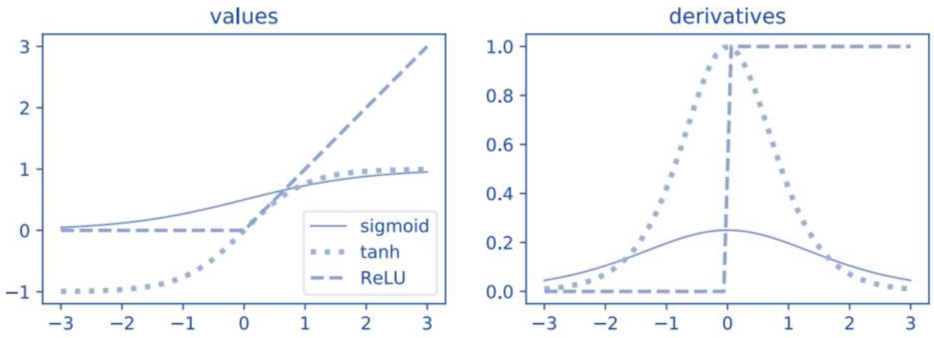
$$\begin{aligned} y &= \frac{1}{1+e^{-x}} \\ \frac{dy}{dx} &= \frac{1}{(1+e^{-x})^2}(-e^{-x}) = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e-x}\left(1 - \frac{1}{1+e-x}\right) = y(1-y)\end{aligned}$$

5.3 Other Activation Functions

The sigmoid in $z = \sigma(\Theta^{x \to z} x)$ is called an activation function. In general, we can write $z = f(\Theta^{x \to z} x)$ to indicate an arbitrary activation function. Other choices include

- **Hyperbolic tangent**: tanh, centered at 0, helps to avoid saturation
- **Rectified linear unit**: $\text{ReLU}(a) = \max(0, a)$ which is fast to evaluate, easy to analyze and avoids saturation even further.
- **Leaky ReLU**:

$$= \begin{cases} a, & a \geq 0 \\ .0001a, & \text{otherwise} \end{cases}$$



5.4 Feed Forward Networks - Issues

- Large number of parameters
- Easy to be over-fitted
- Large memory consumption
- Does not enforce any structure, e.g., local information In many applications, local features are important, e.g., images, language, etc.

Feed Forward Networks are not particularly suited for input data

- which has variable length
- where the order/time-information plays a role
- with long-term dependencies

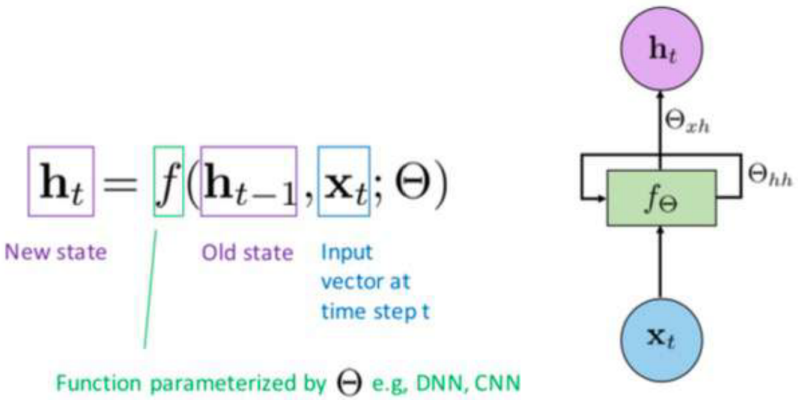
Pattern an sequences

- "I lived in France, unfortunately I haven't learned much ... [French]"
- The food is great, the ambiance is great ...

6 Recurrent Neural Networks

Recurrent neural networks take parts of the output-values (in the hidden or the output layer) at time t as input at time $t+1$. With it the modeling of temporal sequences is possible. The hidden states can be understood as a function of inputs and previous time step information $h_t = f(h_{t-1}, x_t; \Theta)$. Recurrent Neural Networks reuse the same weight matrix Θ at every time step. Temporal information is important in many applications:

- Language
- Speech
- Video



6.1 RNN Architectures

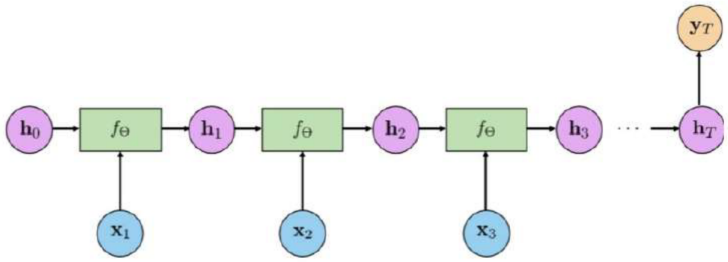


Figure 12: Many to one (sentence labeling or completion)

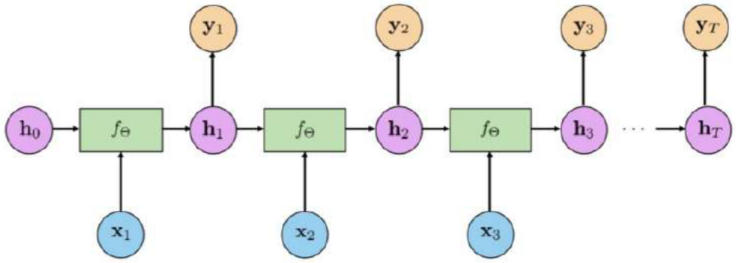


Figure 13: Many to many (machine translation)

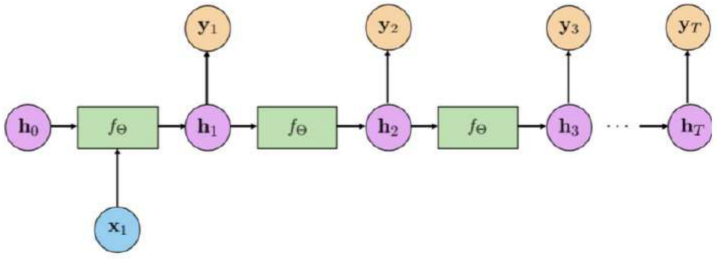
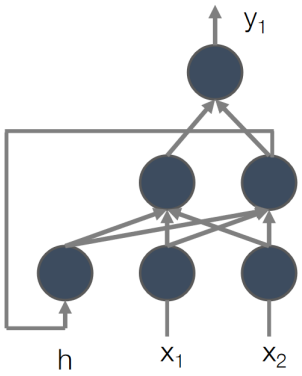


Figure 14: One to many (image caption)

Let's have a look at an example. Given are 2 input variables, 2 hidden layers, 1 output variable and the output of one hidden neuron is also taken as input for the next time step.



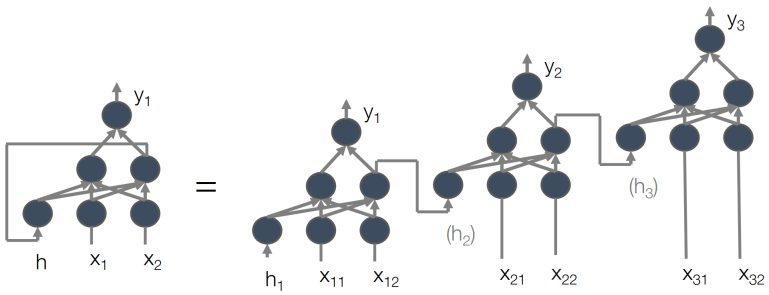
Assume we have the following input and target sequence

- $x = \{(0.8, 0.1), (0.2, 0.6), (0.6, 0.2)\}$
- $y = \{1.0, 1.0, 0.0\}$

To calculate the output, we need

- 1^{st} step ($t = 0$): Set $x_1 = 0.8, x_2 = 0.1, h_0 = 0$ and calculate the output as we are used to.
  - assume we get $y_0 = 0.7 and h_0 = 0.1$
- 2^{nd} step ($t = 1$): Set $x_1 = 0.2, x_2 = 0.6, h_1 = 0.1$ and calculate...
  - ...
- ...
- Finally we get $y' = \{0.7, \ldots, \ldots\}$
- Important: The weights are the same each time step.

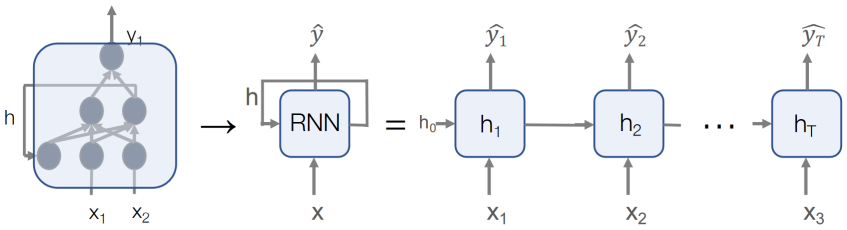
1^{st} Rule on Computer Science: Try to reduce the problem at hand to a problem, where you know the solution. Here: "Unfold" the network, so that it looks like a feed forward network. Now we can use simple backpropagation.



This looks like a normal feed forward network with shared weights ($\Leftarrow$ important point). Input and output can now be calculated in one go.

6.2 Train Recurrent Neural Networks

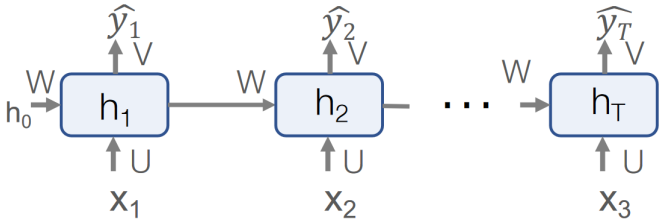
Let's have a look at our example again.



We introduce generic weight matrices

- U = weight matrix for the input variables x
- W = weight (matrix) for the input h
- V = weight matrix between the h and y

Keep in mind: U, V, W do not depend on t



The calculation of the output at time t / input at time t+1

- $h_t = g(Wht - 1 + Ux_t)$
- $\hat{y}_t = g * (Vh_t)$
- where g and $g*$ are our activation functions

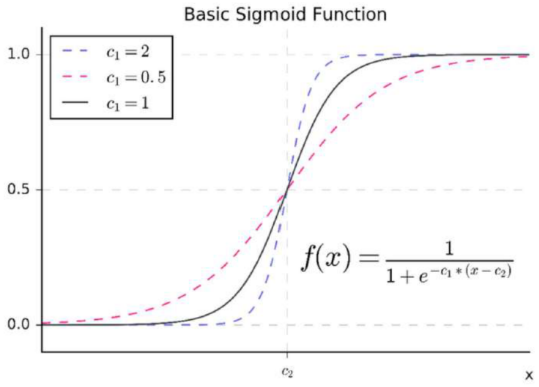
How do define the loss function? Assume we want to get the output correct at each time-step. Assume we are only interested in the last time-step.

6.2.1 Vanishing Gradients

The change with relation to ω_1 at time step 3 is given by

$$\frac{\partial L}{\partial \omega_1} = \frac{\partial L}{\partial \hat{y}} \sigma'(s_3) h_2 \sigma'(s_2) h_1 \sigma'(s_1) x \tag{30}$$

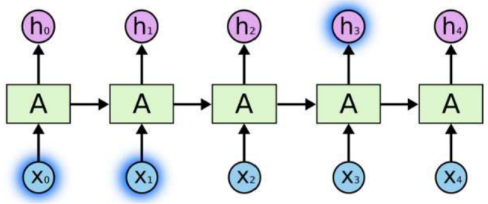
and depends 3 times on σ'.
If σ is a sigmoid function, then all gradients are < 1 and hence the change of ω_1 will be small. The problem gets larger when there are many layers (as in an RNN), i.e. the gradients for the first layers are small. One solution is the so called **Long short-term memory Networks - LSTMs**.



Gradients are everywhere smaller than 1

6.2.2 Long Term Dependencies

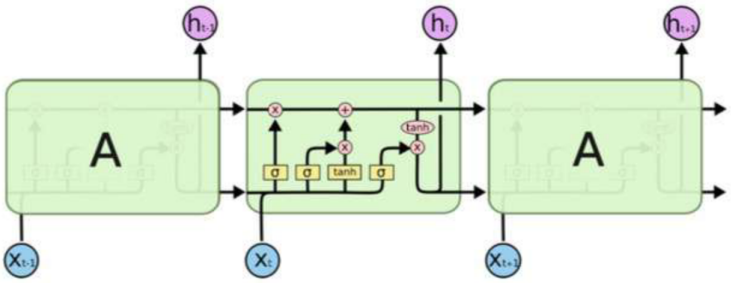
Trying to predict the last word in "The clouds are in the xxx" =? sky. Input is here one vector per word. Easy: the gab between the relevant information and the place that it's needed is small, RNN can learn to use the past information.



In a different example "I grew up in France ... I speak fluent xxx.", predicting the missing words is more difficult. The distance between France and the solution "French" can be quite long. Remember the **vanishing gradient** problem.

6.3 LSTM – core idea

LSTM are explicitly designed to avoid the long-term dependency problem. Instead of having a single neural network layer, there are four, interacting in a very special way.



7 Convolutional Neural Networks

Convolutional Neural Networks are defined by weight sharing and local connectivity (convolution). Multiple filters are being use to convolve over the inputs.

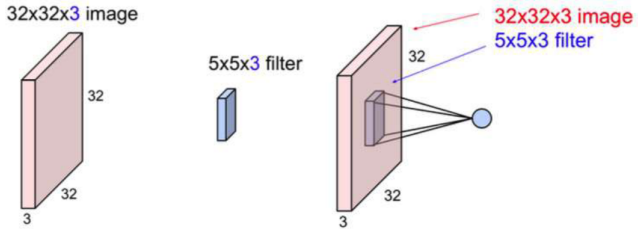
+ Reduce the number of parameters (less over-fitting).
+ Learn local features
+ Translation invariance

Pooling or sub-sampling

+ Make the representation smaller
+ Reduce number of parameters and computation

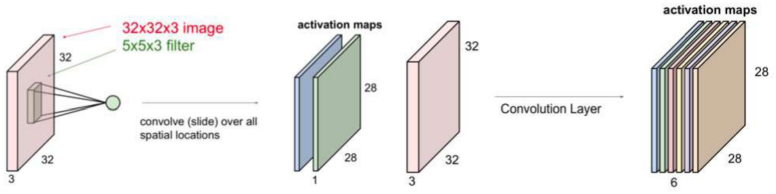
7.1 Convolution and Pooling

Convolutional Neural Networks use **weight sharing** and **translation invariance**. The same weights are applied over different spatial regions. One can achieve translation invariance (not perfect though). If an input is changed spatially (translated or shifted), the corresponding output to recognize the object should not be changed. CNNs can produce the same output, even though the input image is shifted, due to weight sharing. Filters always extend the full depth of the input volume. Convolve the filters with the image i.e. "slide over the image spatially, computing dot products". The first number is the result of taking a dot product between the filter and a small $5 \times 5 \times 3$ chunk of the image (i.e. $5 \cdot 5 \cdot 3 = 75$-dimensional dot product + bias) $\omega^T \cdot x + b$

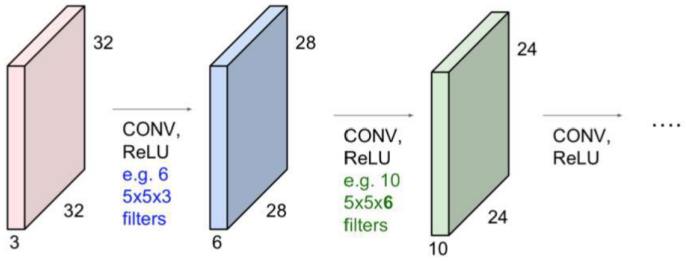


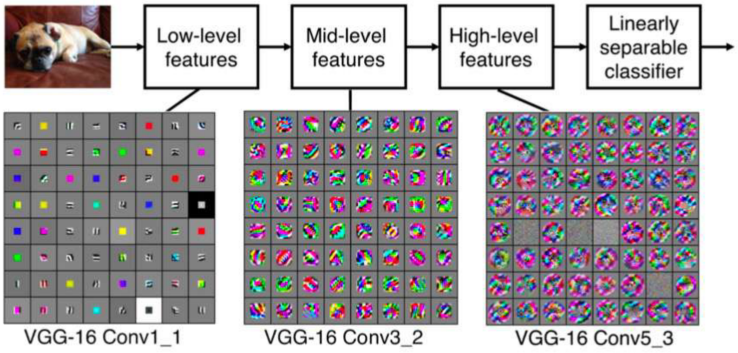
7.2 Convolution Layer

Consider a second, green filter. Then we add 6 5×5 filters and get 6 separate activation maps.



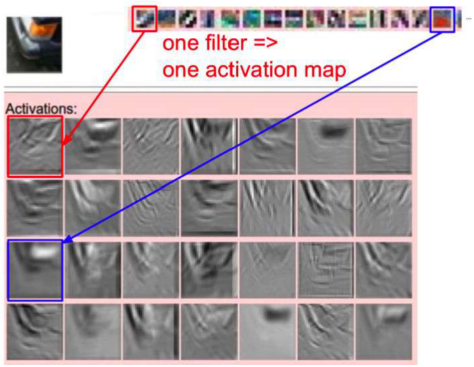
ConvNet is a sequence of Convolution Layers, interspersed with activation functions (e.g. ReLU).

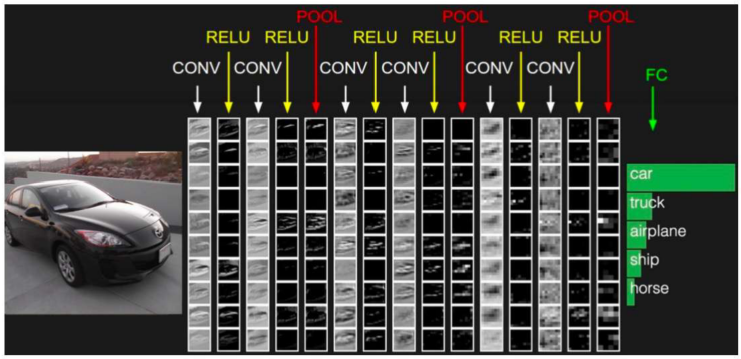




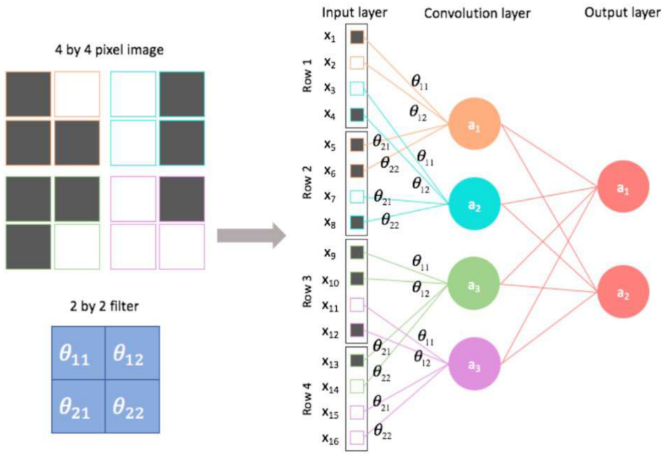
We call the layer convolutional because it is related to the convolution of two signals:

- $f(x,y) \cdot g(x,y) = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f(n_1,n_2) \cdot g(x-n_1, y-n_2)$
- I.e. element wise multiplication and sum of a filter and the signal (e.g. the image)

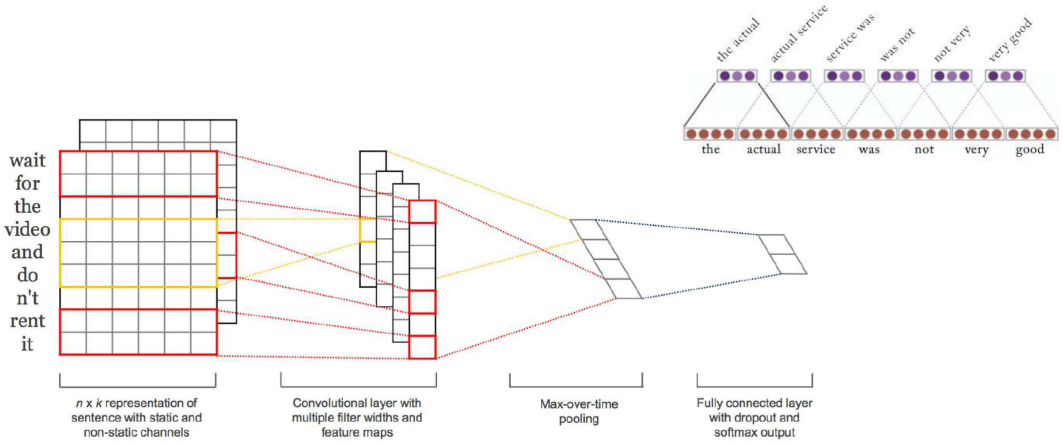




7.3 Convolutional Networks - Weights



7.4 Convolutional Nets in NLP



8 NLP Classification Applications

8.1 Simple cases: label a document

There are many reasons to label a document, some of them are movie review polarity, news article topic tags, identify language of a tweet, identify a book author, filter out spam, predict depression in social media, classify political preference of a social media user or label a chat-bot command with the task requested.

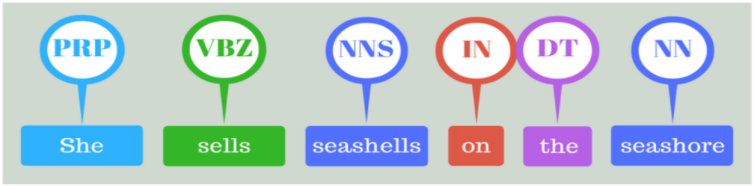
8.1.1 Sentiment Classification

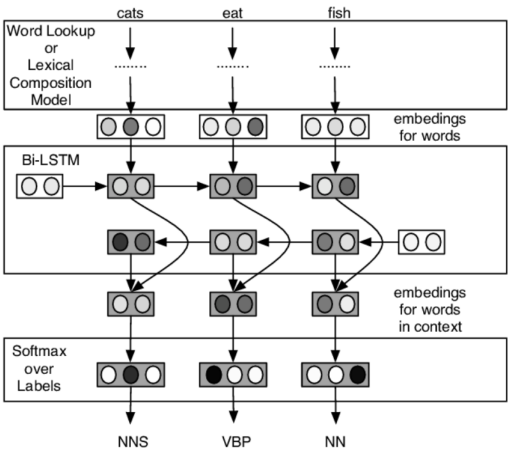
What about unseen words? What about frequent words? What about negation? When it comes to negation one method is to add tags to each word after a negation.

```
didn't like this movie , but I
                ⬇
didn't NOT_like NOT_this NOT_movie but I
```

Lexicons? https://mpqa.cs.pitt.edu/lexicons/subj_lexicon/

8.1.2 Label Each Word





8.2 Harder: Identify sentence areas

We are trying to identify (Named) entities, opinion target, sentiment aspect or argument premises

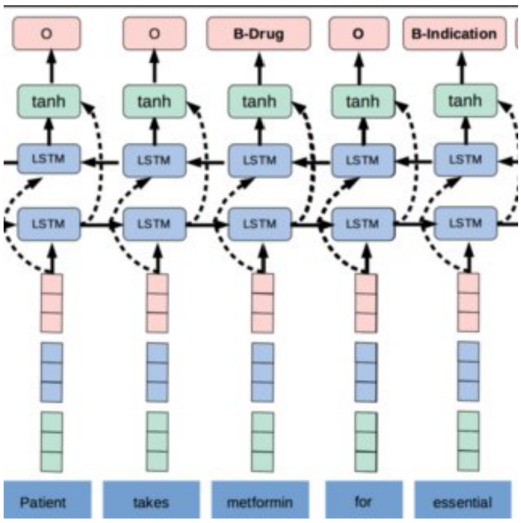
When Sebastian Thrun PERSON started working on self - driving cars at Google ORG in 2007 DATE , few people outside of the company took him seriously . " I can tell you very senior CEOs of major American NORP car companies would shake my hand and turn away because I was n't worth talking to , " said Thrun PERSON , in an interview with Recode ORG earlier this week DATED .

The BIO format (short for beginning, inside, outside) is a common tagging format for tagging tokens in computational linguistics.

Foreign	ORG	B-ORG
Ministry	ORG	I-ORG
spokesman	O	O
Shen	PER	B-PER
Guofang	PER	I-PER
told	O	O
Reuters	ORG	B-ORG
that	O	O
:	:	BIO encoding

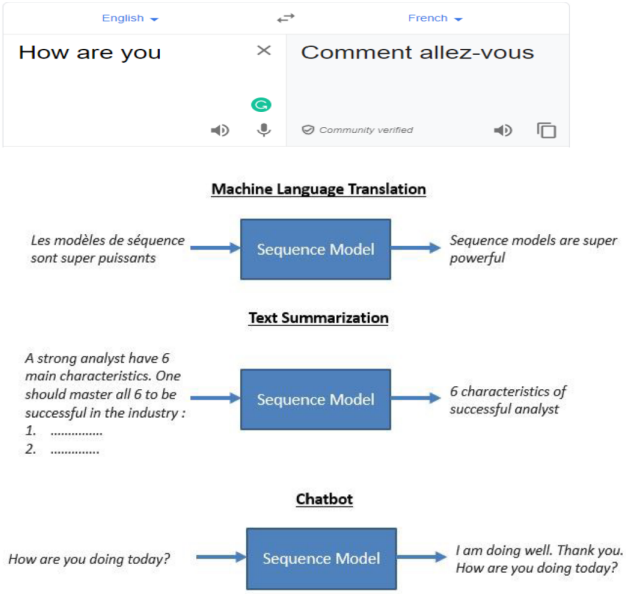
Evaluation: "Precision is the percentage of named entities found by the learning system that are correct. Recall is the percentage of named entities present in the corpus that are found by the system. A named entity is correct only if it is an exact match of the corresponding entity in the data

file." (CoNLL) read more here: https://www.davidsbatista.net/blog/2018/05/09/Named_Entity_Evaluation/



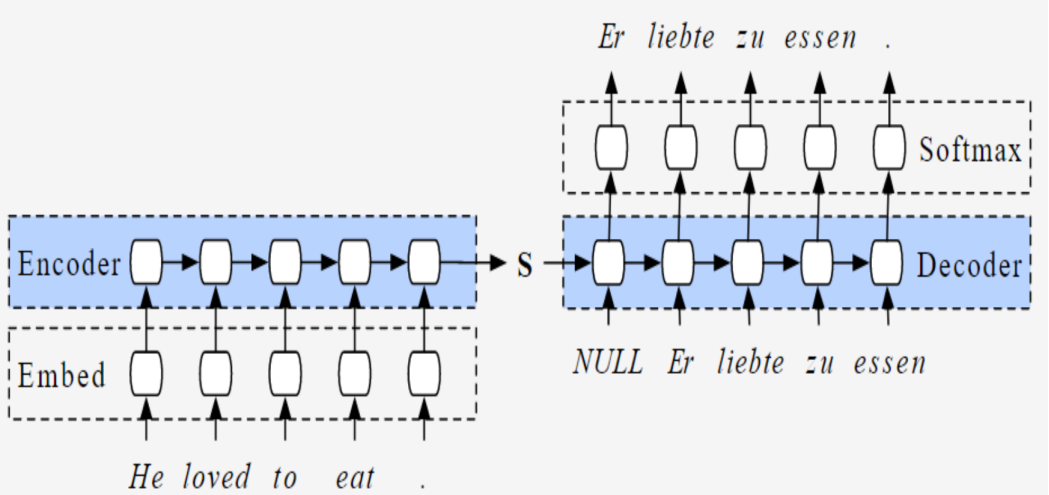
8.3 Tricky: Map Sequence to Sequence

To problem of mapping sequence to sequence is commonly found in machine translation, question answering, text summarization, response generation and speech recognition.



Mapping sequence to sequence is done by Encoder-decoder: During in-

ference, we generate one word at a time. The initial states of the decoder are set to the final states of the encoder. The initial input to the decoder is always the START token. At each time step, we preserve the states of the decoder and set them as initial states for the next time step. At each time step, the predicted output is fed as input in the next time step. We break the loop when the decoder predicts the END token.



Another map sequence to sequence problem is **question answering**. Here we need to translate questions into answers.

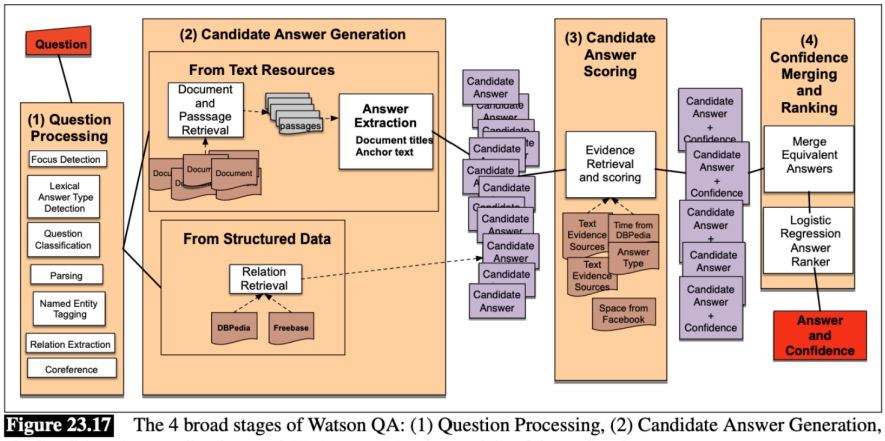
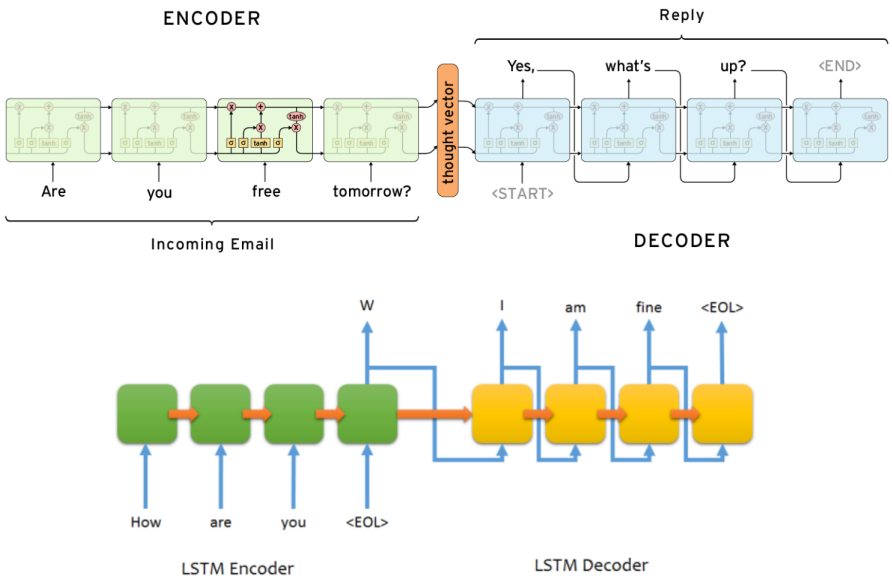


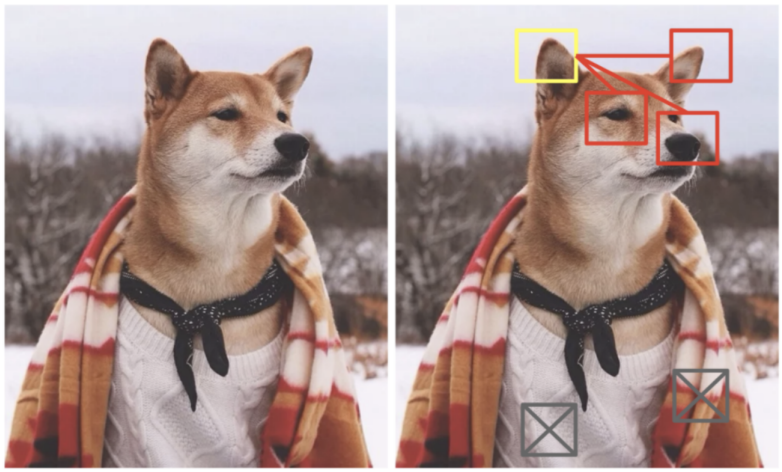
Figure 23.17 The 4 broad stages of Watson QA: (1) Question Processing, (2) Candidate Answer Generation, (3) Candidate Answer Scoring, and (4) Answer Merging and Confidence Scoring.



9 Neural Attention

9.1 Attention intuition for images

Some areas of the image are more important for the correct recognition, especially in combination (interpreting ear by seeing also eye and nose).



9.2 Attention intuition for images

This holds also for generating image descriptions, for example

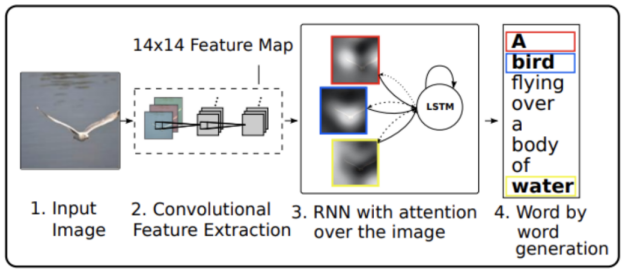
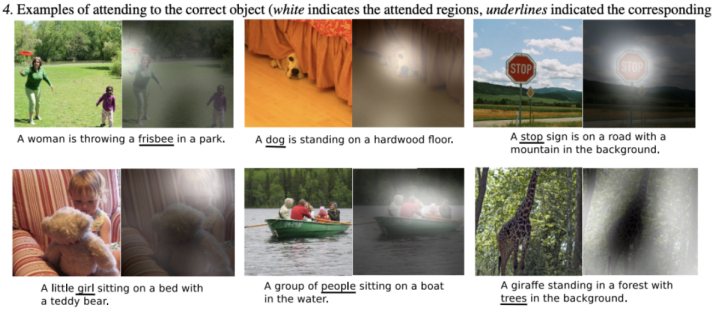
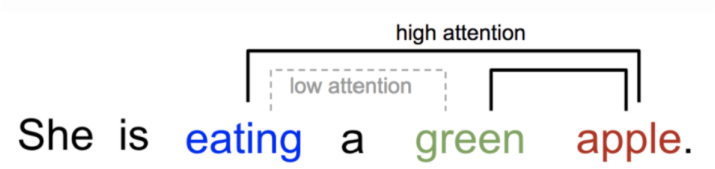


Figure 4. Examples of attending to the correct object (*white* indicates the attended regions, *underlines* indicated the corresponding word)



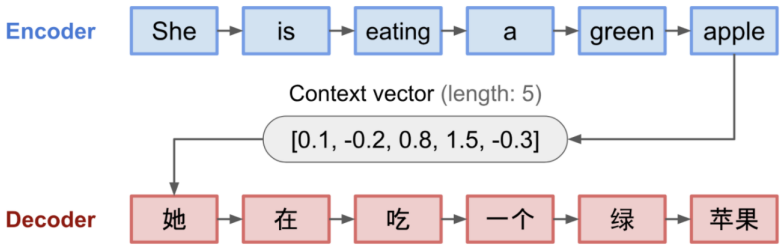
9.3 Attention for text

The same principle can be applied on natural language. Some words in the sentence are more important when trying to interpret new words.



9.4 Refresher: RNN

Seq2seq encoder-decoder:
An encoder processes the input sequence and compresses the information into a vector (sentence embedding) of a fixed length. This representation is expected to be a good summary of the meaning of the whole source sequence.
A decoder is initialized with the context vector to emit the transformed output. The early work only used the last state of the encoder network as the decoder initial state.
Problem: Capturing long contexts (forgetting). https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html



9.5 Attention

The idea is instead of building a single context vector out of the encoder's last hidden state, create shortcuts between the context vector and the entire source input:

- encoder hidden states
- decoder hidden states
- alignment between source and target

The weights of these shortcut connections are customizable for each output.

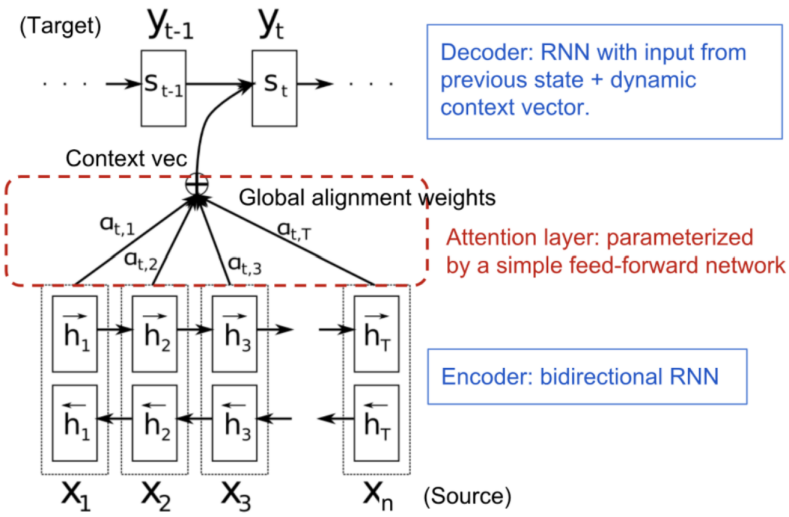


Figure 15: https://lilianweng.github.io/posts/2018-06-24-attention/

9.5.1 Computing Attention

Learn the parameters alpha (the attention weights). For example just by adding a fully connected DNN with one hidden layer. The goal is to predict

how well an input word aligns with an output word (an alignment score $e^{t,t'}$).

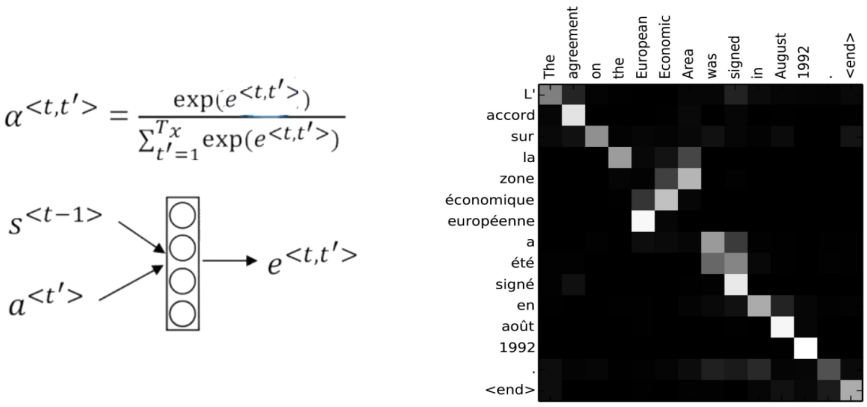


Figure 16: https://lilianweng.github.io/posts/2018-06-24-attention/

9.5.2 Self-Attention

Self-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the same sequence.

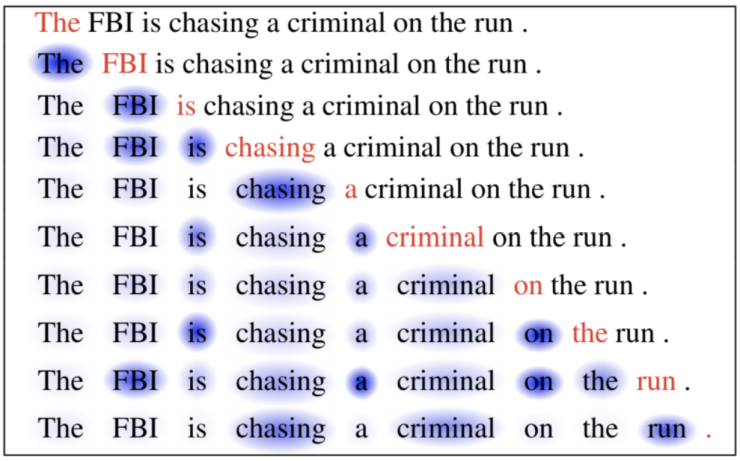


Figure 17: https://lilianweng.github.io/posts/2018-06-24-attention/

10 Classification Evaluation

Does my trained model work? Models are evaluated on the basis of correct and incorrectly classified instances on new data; not on training! We construct a confusion matrix (truth vs. prediction) for each class. For problems with 2 classes, four cases can occur.

- True Positive (TP): positive class correctly predicted.
- False Positive (FP): positive class incorrectly predicted.
- True Negative (TN): negative class correctly predicted.
- False Negative (FN): Negative class incorrectly predicted.

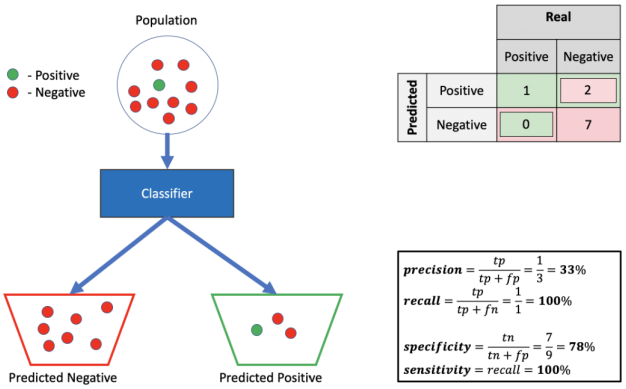
		PREDICTED CLASS	
		Class=Yes	Class=No
ACTUAL CLASS	Class=Yes	True Positive (TP)	False Negative (FN)
	Class=No	False Positive (FP)	True Negative (TN)

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

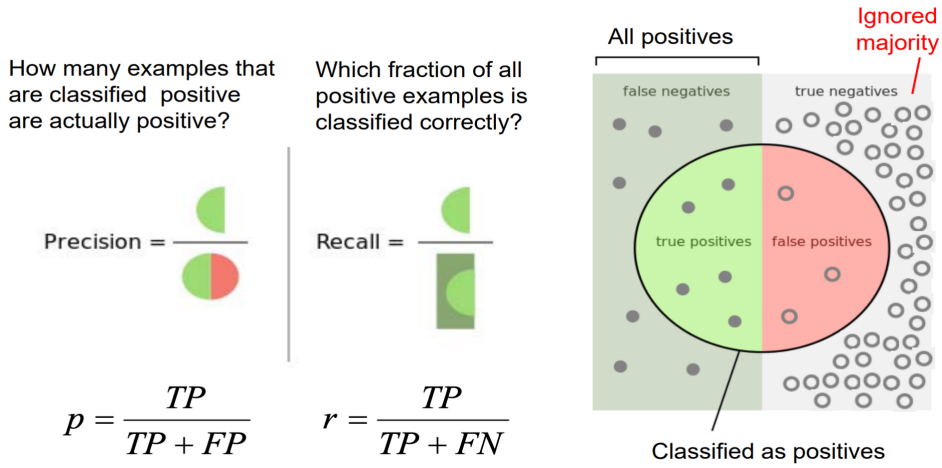
10.1 Problem: Unbalanced Data

If 99% belong to the class "Yes" and the classifier always says "Yes": $\Rightarrow$ 99% Accuracy.
Precision: How many of the examples labeled "Yes" are really "Yes"? ("the number of correctly labeled examples divided by the number of all examples that were labeled with this class ")
Recall: How many of the examples that are really "Yes" were labeled "Yes"? ("the number of correctly labeled examples divided by the number of all examples that actually belong to this class")

$$Precision = \frac{TP}{TP + FP} \qquad Recall = \frac{TP}{TP + FN}$$



10.2 Precision and Recall Visually



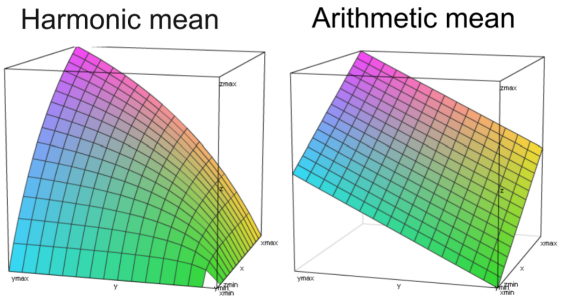
$$p = \frac{TP}{TP + FP} \qquad r = \frac{TP}{TP + FN}$$

10.3 F-score Visually

F_1-score is the harmonic mean of precision and recall. The harmonic mean of two numbers tends to be closer to the smaller of the two. Thus for the F_1-score to be large, both p and r must be large.

$$F_1 = \frac{2pr}{p+r}$$

$$= \frac{2TP}{2TP + FP + FN}$$



10.4 Examples of imbalanced results

	PREDICTED CLASS		
ACTUAL CLASS		Class=Yes	Class=No
	Class=Yes	10	0
	Class=No	10	980

$$\text{Precision (p)} = \frac{10}{10+10} = 0.5$$

$$\text{Recall (r)} = \frac{10}{10+0} = 1$$

$$\text{F}_1\text{ - measure (F}_1\text{)} = \frac{2*1*0.5}{1+0.5} = 0.62$$

$$\text{Accuracy} = \frac{990}{1000} = 0.99$$

	PREDICTED CLASS		
ACTUAL CLASS		Class=Yes	Class=No
	Class=Yes	1	9
	Class=No	0	990

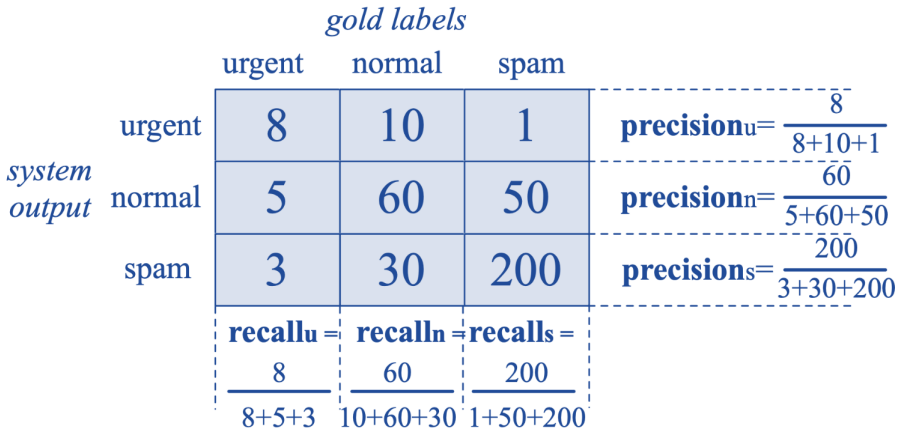
$$\text{Precision (p)} = \frac{1}{1+0} = 1$$

$$\text{Recall (r)} = \frac{1}{1+9} = 0.1$$

$$\text{F}_1\text{ - measure (F}_1\text{)} = \frac{2*0.1*1}{1+0.1} = 0.18$$

$$\text{Accuracy} = \frac{991}{1000} = 0.991$$

10.5 Example: more than two classes

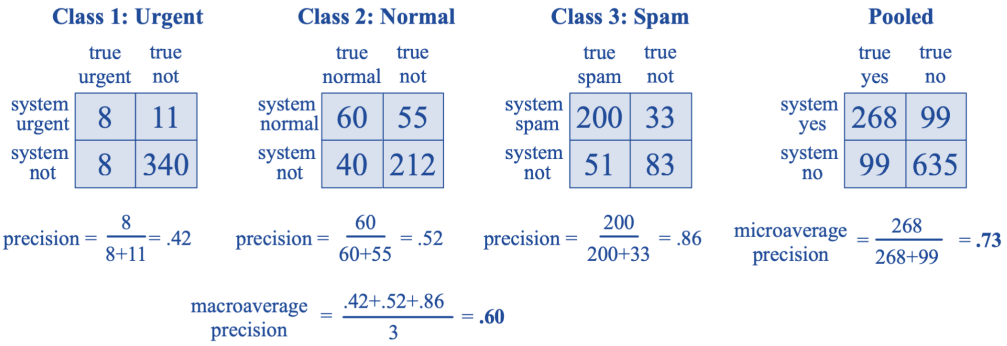


10.6 Micro-average vs Macro-average

Micro and macro across classes:
Let's have a classification model on a data-set with 3 classes. Then:
Macro-averaging:
compute the performance for each class, and then average over classes
Micro-averaging: collect decisions for all classes into one confusion matrix
compute precision and recall from that table.

Micro and macro across datasets:
Let's run a binary classification model on two different data-sets, 1 and 2. Then:
Micro-average of precision = (TP1+TP2)/(TP1+TP2+FP1+FP2)
Microaverage of recall = (TP1+TP2)/(TP1+TP2+FN1+FN2)
Macroaverage precision = (P1+P2)/2
Macroaverage recall = (R1+R2)/2

10.7 More than two classes



Why do you say "F_1"-Score?

$$F_\beta = \frac{(\beta^2+1)PR}{\beta^2 P + R} \quad F_1 = \frac{2PR}{P+R} \tag{31}$$

In rare cases, precision and recall are not equally important for the measure.

10.8 Overfitting

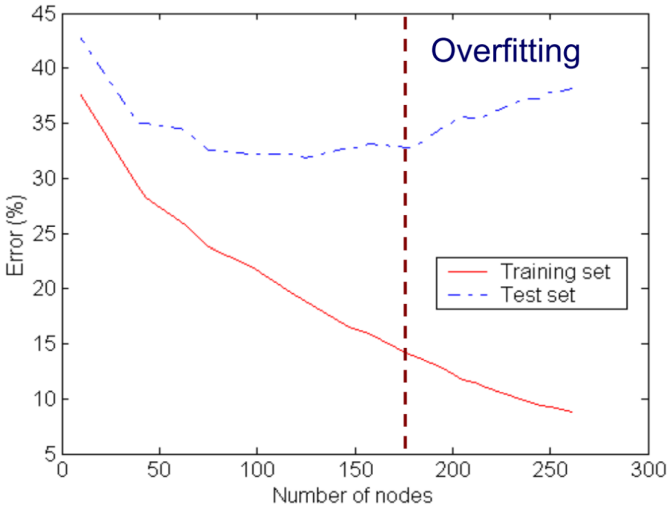
Symptoms:

1. Model is too complicated for the data-set size (too many rules ...).
2. Model works well on training set but performs bad on test set.

Typical causes of overfitting:

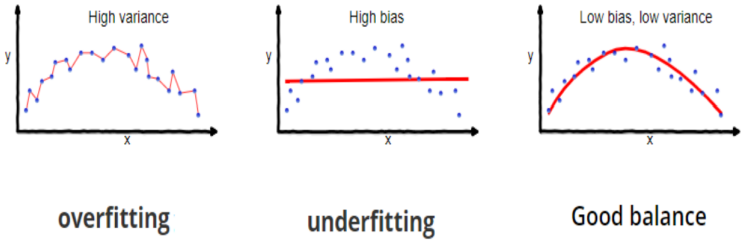
1. Too little training data.
2. Noise / outliers in training data.
3. High model complexity.

An over-fitted model does not generalize well to unseen data.



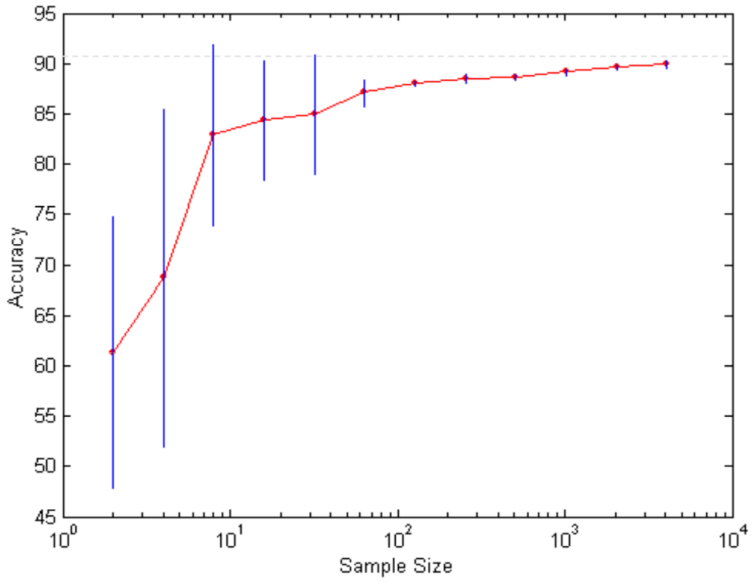
10.9 Bias and variance

Variance of the parameter estimates across samples can be reduced by increasing the bias in the estimated parameters and vice versa.
Conflict in trying to simultaneously minimize these two sources of error.
Bias = erroneous assumptions in the learning algorithm. Can cause an algorithm to miss the relevant relations between features and target outputs (under-fitting).
Variance = sensitivity to small fluctuations in the training set. High variance can cause an algorithm to model the random noise in the training data, rather than the intended outputs (over-fitting).

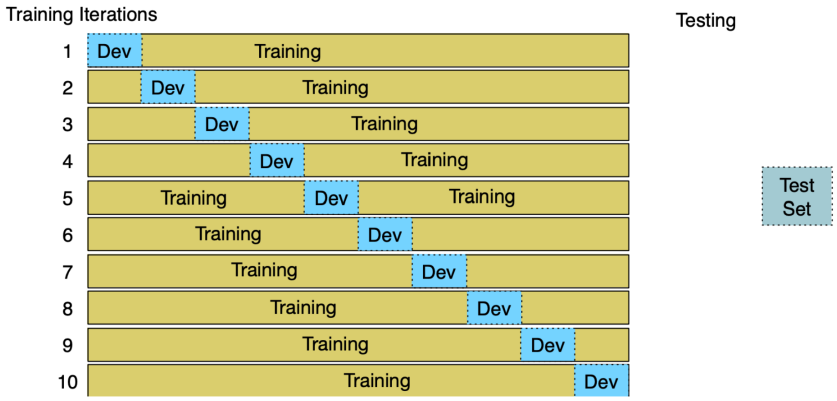


10.10 Increasing the training set

The learning curve shows how accuracy changes with growing training set size. If model performance is low and unstable, get more training data. Use labeled data rather for training than testing.



10.11 Robust results: Cross-validation

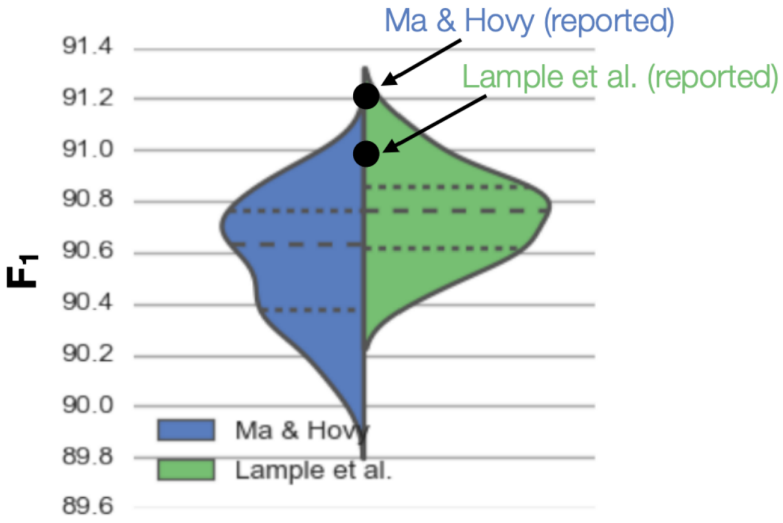


10.12 Comparing classifiers

Suppose two teams build classifiers to solve a problem:

- C_1 gets 82% accuracy
- C_2 gets 73% accuracy

Will C_1 be more accurate in the future? What if the test set has 1000 examples? What if the test set has 10 examples?



11 Word representation I – Lexical resources

What is a meaning of a word? In the text representation methods we have discussed so far, words are just strings (or indices in a vocabulary list). But humans "know" that some words are more related than others. E.g. a dog is closer to a cat than to tax declaration. So how do we know? There are generally, two options, look it up in a dictionary or guess it from the context.

11.1 Lexical Resources

A **lexical resource** is a digital knowledge base which provides information on words of a particular language. This information is typically accessed through a **lemma**, i.e. one lemmatized base form of a word. In some cases the term **lexeme** is used instead. Lexeme is a combination of a lemma and its part of speech, e.g. serve (verb). A word sense is a pairing between a lemma of the word, and one of the definitions or examples of its possible meanings. Lexical resources e.g. are WordNet, BabelNet, GermaNet, Wiktionary or (Wikipedia).

11.1.1 WordNet / GermaNet

Lexical database for nouns, verbs and adjectives/adverbs. Each word sense is arranged in a **synset** (category of near-synonyms). And each synset is related to others in terms of their sense **relations**.

synset	gloss
mark, grade, score	a number or letter indicating quality
scratch, scrape, scar, mark	an indication of damage
bell ringer, bull's eye, mark, home run	something that exactly succeeds in achieving its goal
chump, fool, gull, mark, patsy, fall guy, sucker, soft touch, mug	a person who is gullible and easy to take advantage of
mark, stigma, brand, stain	a symbol of disgrace or infamy

11.1.2 Relationships between senses

Relation	Also Called	Definition	Example
Hypernym	Superordinate	From concepts to superordinates	*breakfast*^{1} → *meal*^{1}
Hyponym	Subordinate	From concepts to subtypes	*meal*^{1} → *lunch*^{1}
Instance Hypernym	Instance	From instances to their concepts	*Austen*^{1} → *author*^{1}
Instance Hyponym	Has-Instance	From concepts to concept instances	*composer*^{1} → *Bach*^{1}
Member Meronym	Has-Member	From groups to their members	*faculty*^{2} → *professor*^{1}
Member Holonym	Member-Of	From members to their groups	*copilot*^{1} → *crew*^{1}
Part Meronym	Has-Part	From wholes to parts	*table*^{2} → *leg*^{3}
Part Holonym	Part-Of	From parts to wholes	*course*^{7} → *meal*^{1}
Substance Meronym		From substances to their subparts	*water*^{1} → *oxygen*^{1}
Substance Holonym		From parts of substances to wholes	*gin*^{1} → *martini*^{1}
Antonym		Semantic opposition between lemmas	*leader*^{1} ⟺ *follower*^{1}
Derivationally Related Form		Lemmas w/same morphological root	*destruction*^{1} ⟺ *destroy*^{1}

Figure 17.2 Noun relations in WordNet.

11.1.3 Synonymy

Two senses of different words are synonymy of each other if their meaning is nearly identical. Synonyms can be exchanged for each other without changing the truth of a sentence. Synonymy holds between word senses, not words.
How **big** is that plane? = Would I be flying on a **large** or small plane?
Miss Nelson became a kind of **big** sister to him. ≠ Miss Nelson became a kind of **large** sister to him.

11.1.4 Antonymy

Two senses of different words are antonymy of each other if their **meaning is nearly opposite**. All aspects of meaning are nearly identical between antonymy, except one.

long	short	both describe length
big	little	both describe size
fast	slow	both describe speed
cold	hot	both describe temperature
dark	light	both describe luminescence

11.1.5 Hypernymy/Hyponymy

Sense A is a hyponym of sense B if A is a subclass of B.

hyponym/subordinate	hypernym/superordinate
car	vehicle
mango	fruit
chair	furniture
dog	mammal
mammal	animal

11.1.6 Meronymy/Holonymy

Part-whole relations. A meronym is a part of a holonym.

meronym	holonym
leg	chair
wheel	car
car	automobile

11.2 Word similarity using WordNet

Here an example of working with WordNet

```python
# First, you're going to need to import wordnet:
from nltk.corpus import wordnet

# Then, we're going to use the term "program" to find synsets like so:
syns = wordnet.synsets("program")

# An example of a synset:
print(syns[0].name())

# Just the word:
print(syns[0].lemmas()[0].name())

# Definition of that first synset:
print(syns[0].definition())

# Examples of the word in use in sentences:
print(syns[0].examples())
```

The output will look like:

```
plan.n.01
plan
a series of steps to be carried out or goals to be accomplished
['they drew up a six-step plan', 'they discussed plans for a new bond issue']
```

```python
import nltk
from nltk.corpus import wordnet
synonyms = []
antonyms = []

for syn in wordnet.synsets("good"):
    for l in syn.lemmas():
        synonyms.append(l.name())
        if l.antonyms():
            antonyms.append(l.antonyms()[0].name())

print(set(synonyms))
print(set(antonyms))
```

There are multiple ways to measure word similarity using lexical resources. Two main strategies are; graphing the distance in the path of connections between words (e.g. Wu-Palmer similarity) and comparison of word definitions (e.g. Lesk algorithm).

```python
import nltk
from nltk.corpus import wordnet
# Let's compare the noun of "ship" and "boat:"

w1 = wordnet.synset('run.v.01') # v here denotes the tag verb
w2 = wordnet.synset('sprint.v.01')
print(w1.wup_similarity(w2))
```

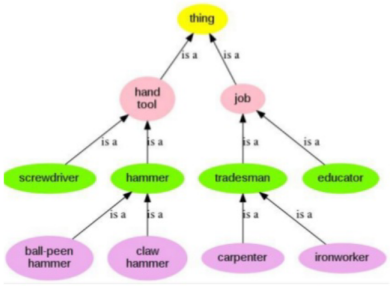
Output:

```
0.857142857143
```

```python
w1 = wordnet.synset('ship.n.01')
w2 = wordnet.synset('boat.n.01') # n denotes noun
print(w1.wup_similarity(w2))
```

Output:

```
0.9090909090909091
```



12 Word representation II – context similarity

Some words are not similar, but related in some way; surgeon, nurse, scalpel and anesthetic are related by the place they can be found in, a hospital. Vector similarities are good in capturing that, too.

12.1 Word meaning space

We are defining meaning as a point in space based on distribution. Each word is a vector (not just "good" or ω_{45}). Similar words are "nearby in se-

mantic space". We build this space automatically by accessing which words are nearby in the text. We call this word representation an **"embedding" of a word** because it's embedded into a space. It is the standard way to represent meaning in NLP. Every modern NLP algorithm uses embeddings as the representation of word meaning for similarity assessments.
There are multiple ways to convert words into vectors. Some frequently used and simple approaches are

- TF-IDF (older, comes from the web search area)
- Word2vec (popular in the last ±5 years)

12.1.1 TF-IDF

Encoding context with TF-IDF our starting point is a term-document matrix. In this context one document is (one book). In practice, one document can be also one tweet etc.

	Hamlet	Macbeth	Romeo & Juliet	Richard III	Julius Caesar	Tempest	Othello	King Lear
knife	1	1	4	2		2		2
dog	2		6	6		2		12
sword	17	2	7	12		2		17
love	64		135	63		12		48
like	75	38	34	36	34	41	27	44

Vector representation of a document

One word is represented as a distribution over documents. The word vector length is the size of the document collection. The word vector is made of the counts of the occurrences of the word within the given documents. Some words appear more frequently everywhere. **TF-IDF** stands for Term Frequency (TF), Inverse Document Frequency (IDF). A scaling to represent a feature as function of how frequently it appears in a data point but accounting for its frequency in the overall collection. TF is the number of times a term occurs in a document. IDF is the number of total documents in collection, divided by the number of documents that contain the term.

$$tf_{t,d} = \log(count(t,d)+1) idf_t = \log \frac{N}{df_t} \tag{32}$$

knife	1	1	4	2		2		2

sword	17	2	7	12		2		17

Where N is the number of documents in the corpus and df is the document frequency, the number of documents a term appears in. **TF-IDF** combines these two terms; $tf - idf_{t,d} = tf_{t,d} \cdot idf_t$.
But what if we need to represent words across a very large number of documents? We need some dimensionality reduction for the vector size.

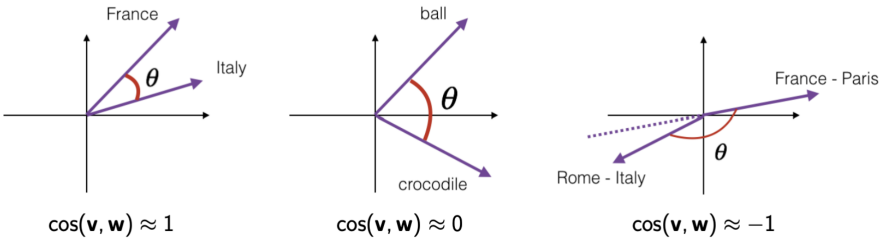
12.1.2 Measuring vector similarity

Frequent words (of, the, you) have long vectors (since they occur many times with other words). The dot product is higher when the vectors are longer.

$$v \cdot w = \sum_{i=1}^{N} v_i w_i = v_1 w_1 + v_2 w_2 + \cdots + v_N w_N \tag{33}$$

The dot product favors frequent words. To tackle this imbalance we normalize by vector norm, using the **cosine similarity**

$$\cos(v, w) = \frac{v \cdot w}{|v| \cdot |w|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2} \sqrt{\sum_{i=1}^{N} w_i^2}} \tag{34}$$



We are using cosine as similarity metric

- -1: the vectors point in opposite directions
- +1: the vectors point in the same direction
- +0: the vectors are orthogonal to each other

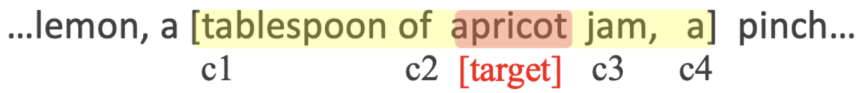
But since raw frequency values are non-negative, the cosine for term-term matrix vectors, ranges from 0 to 1.

12.2 Word2Vec

TF-IDF vectors very sparse, Word2vec vectors are dense. Short vectors are often easier to use as features in a classifier (fewer parameters). On the other hand dense vectors may generalize better, rather than storing explicit counts. Sparse vectors may better capture synonymy. In practice, they work better for most tasks.

12.3 Word2Vec Embeddings

The idea is instead of counting how often each word w occurs near e.g. "apricot", train a classifier on a binary prediction task; is w likely to show up near apricot? We don't actually care about performing this task, but we'll take the learned classifier weights as the word embeddings. The training is self-supervised, no annotated data required, just raw text!



Here two algorithms for the task

- Context bag-of-words (CBOW): predict the current word using context

$$P(w_t|w_{t+1},\ldots,w_{t+k},w_{t-1},\ldots,w_{t-k})$$

- Skip-gram: predict each context word using current word

$$P(w_{t+1},\ldots,w_{t+k},w_{t-1},\ldots,w_{t-k}|w_t)$$

12.4 Word2vec Training: SGNS

Looking at the example above, we train a classifier that is given a candidate (**w**ord, **c**ontext) pair, {(apricot, jam),(apricot, aardvark),...} and assign each pair a probability, $P(+|w,c), P(-|w,c) = 1 - P(+|w,c)$. $\text{Sim}(w,c) \approx w \cdot c$. To turn this into a probability we will use the sigmoid from logistic regression.

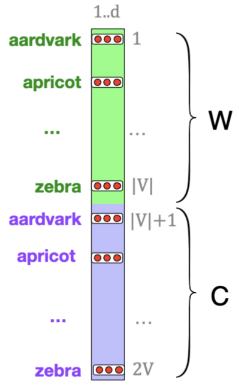
$$\begin{aligned} P(+|w,c) &= \sigma(c \cdot w) = \frac{1}{1+\exp(-c \cdot w)} \\ P(-|w,c) &= 1 - P(+|w,c) \\ &= \sigma(-c \cdot w) = \frac{1}{1+\exp(c \cdot w)} \end{aligned}$$

This is for one context word, but we have lots of context words. We assume independence and multiply them.

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

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

A probabilistic classifier, given a test target word w and its context window of L words $c_{1:L}$, estimates the probability that w occurs in this window based on similarity of w (embeddings) to $c_{1:L}$ (embeddings). To compute this, we just need embeddings for all the words.

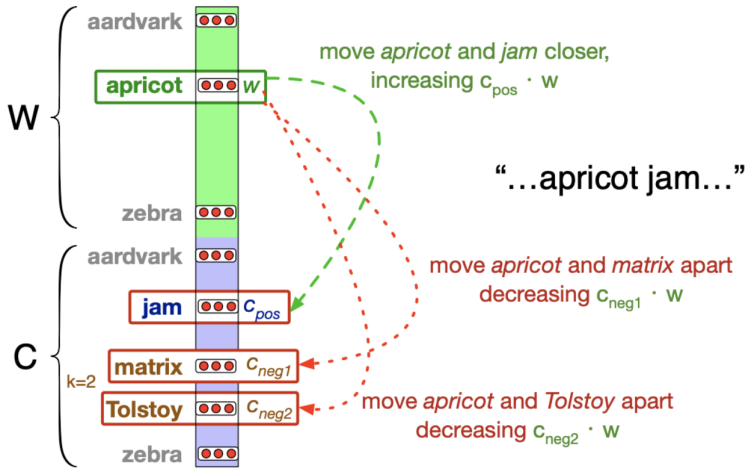


SGNS stands for Skip Ngrams with **Negative Sampling**. In our case the positive examples would be {(apricot, tablespoon), (apricot, of), (apricot, jam), (apricot, a)}. For each positive example we grab k negative examples, sampling by frequency. {(apricot, aardvark), (apricot, my), (apricot, where), ...}.
Given the set of positive and negative training instances, and an initial set of embedding vectors. The goal of learning is to adjust those word vectors such that we; maximize the similarity of the target word's, context word pairs (w, c_{pos}) drawn from the positive data pool, and minimize the similarity of the (w, c_{neg}) pairs drawn from the negative data pool.
Maximize the similarity of the target with the actual context words, and minimize the similarity of the target with the k negative sampled non-neighbor

words.

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

To maximize/minimize the positive/negative similarities we use **stochastic gradient descent**.

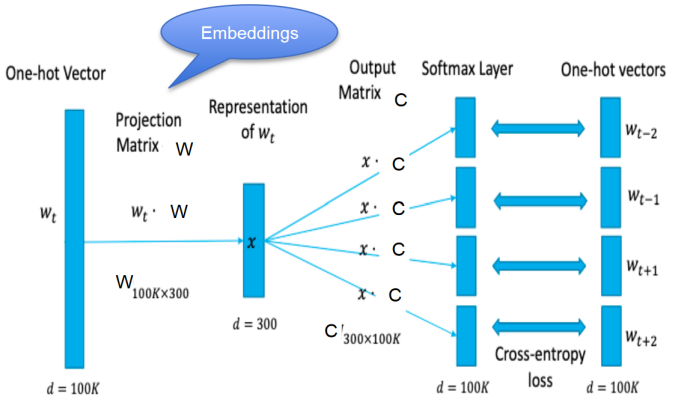


$$
\begin{aligned}
\frac{\partial L_{CE}}{\partial c_{pos}} &= [\sigma(c_{pos} \cdot w) - 1]w \\
\frac{\partial L_{CE}}{\partial c_{neg}} &= [\sigma(c_{neg} \cdot w)]w \\
\frac{\partial L_{CE}}{\partial w} &= [\sigma(c_{pos} \cdot w) - 1]c_{pos} + \sum_{i=1}^{k}[\sigma(c_{neg_i} \cdot w)]c_{neg_i}
\end{aligned}
$$

We start with randomly initialized C and W matrices, then incrementally

do updates.

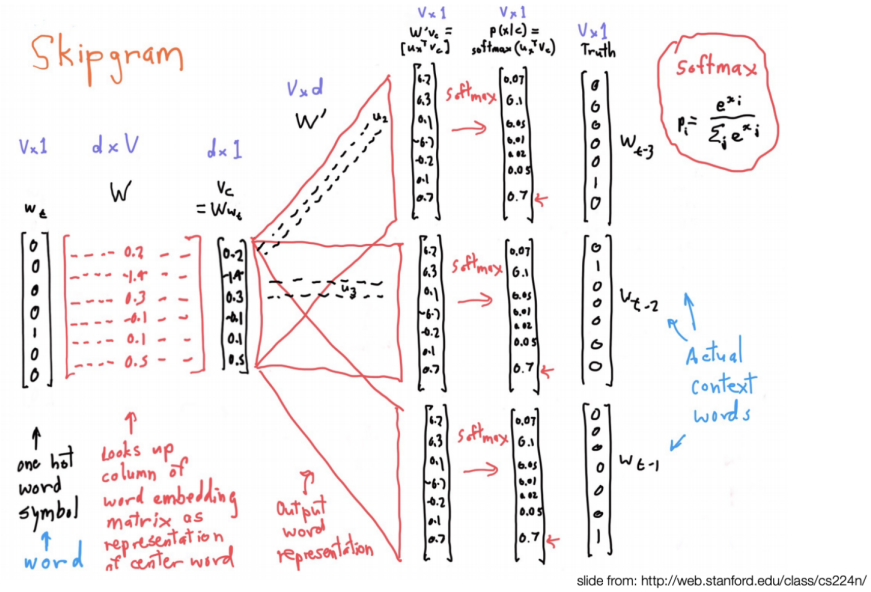
$$c_{pos}^{t+1} = c_{pos}^{t} - \eta[\sigma(c_{pos}^{t} \cdot w^{t}) - 1]w^{t}$$
$$c_{neg}^{t+1} = c_{neg}^{t} - \eta[\sigma(c_{neg}^{t} \cdot w^{t})]w^{t}$$
$$w^{t+1} = w^{t} - \eta \left[[\sigma(c_{pos} \cdot w^{t}) - 1]c_{pos} + \sum_{i=1}^{k} [\sigma(c_{neg_i} \cdot w^{t})]c_{neg_i} \right]$$



To summarize word presentation via Word2Vec:

- Choose the embedding dimension, e.g., d=300.
- Start with random 300-dimensional vectors as initial embeddings.
- Take a corpus and take pairs of words that co-occur as positive examples.
- Construct negative examples.
- Train a classifier to distinguish positive from negative examples.
- Throw away the classifier and keep the embeddings.

You can find tutorial to this topic here
https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html
https://www.kaggle.com/code/pierremegret/gensim-word2vec-tutorial/notebook
textminingonline

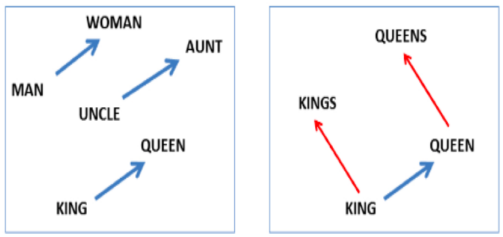


12.5 Embeddings: Practical tips

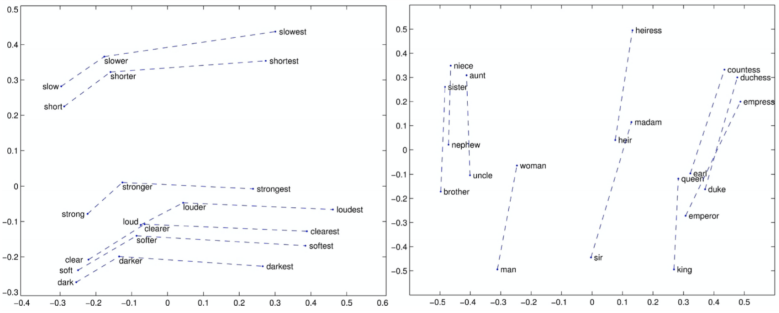
Small windows ($C = \pm 2$), nearest words are syntactically similar. Large windows ($C = \pm 5$), nearest words are semantically related. Vector operations on embeddings help to solve analogy tasks, which in turn helps with relation extraction. This seems to work just for frequent words, small distances, some relations. But it can contain lots of historical bias.

vector('king') - vector('man') + vector('woman') ≈ vector('queen')

vector('Paris') - vector('France') + vector('Italy') ≈ vector('Rome')



Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza



The easiest way to train Word2Vec is probably via the Gensim libarary for Python https://radimrehurek.com/gensim/models/word2vec.html. Some pretrained embeddings are Word2Vec [Mikolov et al. 2013]: https://code.google.com/archive/p/word2vec/, GloVe [Pennington et al. 2014]: http://nlp.stanford.edu/projects/glove/ and Fasttext [Bojanowsi et al. 2017]: ttp://www.fasttext.cc/.
There are many libraries for embedding visualizations (PCA or T-SNE projections to 2D) e.g. http://ronxin.github.io/wevi/.

13 Probabilistic Language Modeling

Probability of a sentence – why? Language modelling is assigning probabilities to words in a sentence. Which sequences of words are likely to appear together? An example in **machine translation**

- P(high winds tonight) > P(large winds tonight)
- P(my big sister) > P(my large sister)

in **spelling correction**

- The office is about fifteen minuets from my house
- P(about fifteen minutes) > P(about fifteen minuets)

in **speech recognition**

- P(I saw a van) >> P(eyes awe of an)
- P(Recognize speech) >> P(Wreck a nice beach)

and summarization, question answering and many more.

13.1 Probabilistic language models

The goal is to calculate the probability of a sentence or a sequence of words. $P(w) = P(w_1, w_2, \ldots, w_n)$. A related task is the probability of the next word $P(w_5|w_4, w_3, w_2, w_1)$. A model that computes either of these tasks is called a **language model** or **LM**. Given the example; complete the holiday destination description "Its water is so transparent, that ...". We want to compute the joint probability P(its, water, is, so, transparent, that). Intuitively we use the chain rule of probability

$$P(B|A) = \frac{P(A,B)}{P(A)} \rightarrow P(A,B) = P(A) \cdot P(B|A)$$

$$\begin{aligned} P(x_1, x_2, \ldots, x_n) &= P(x_1) \cdot P(x_2|x_1) \cdot P(x_3|x_2, x_1) \cdot \cdots \cdot P(x_n|x_1, x_2, \ldots x_{n-1}) \\ &= \prod_{i=1}^{n} P(x_i|x_1, x_2, \ldots, x_{i-1}) \end{aligned}$$

Back to our example

$$\begin{aligned} P(\text{its, water, is, so, transparent, that}) = P(\text{its}) \cdot P(\text{water}|\text{its}) \cdot \\ P(\text{is}|\text{its, water}) \cdot \cdots \cdot P(\text{that}|\text{its, water, is, so, transparent}) \end{aligned}$$

Can we just count and divide (relative frequency estimate)?

$$\begin{aligned} &P(\text{the}|\text{its, water, is, so, transparent, that}) \\ &= \frac{count(\text{its, water, is, so, transparent, that}, textthe)}{count(\text{its, water, is, so, transparent, that})} \end{aligned}$$

No, it results in too many (infinitely) possible sentences. Its to sparse, we will never observe enough data to estimate.

13.2 Markov Assumption

The Markov Assumption says, that conditional probability distribution of future states (words) depends only on the present state, not the sequence of events that preceded it. We make the simplified assumption

$$P(\text{the}|\text{its, water, is, so, transparent, that}) \approx P(\text{the}|\text{that})$$

or maybe

$$\begin{aligned} P(\text{the}|\text{its, water, is, so, transparent, that}) &\approx P(\text{the}|\text{transparent, that}) \\ P(w_1, w_2, w_3, \ldots, w_n) &\approx \prod_{i=1} P(w_i|w_{i-1}, \ldots, w_{i-k}) \end{aligned}$$

In other words, approximate each component of the product.
$P(w_i|w_{i-1}, \ldots, w_2, w_1) \approx P(w_i|w_{i-1}, \ldots, w_{i-k})$. The simplest case is the

unigram model with $k = 0$, $P(w_1, w_2, \ldots, w_n) \approx \prod_i P(w_i)$. Here some examples to results of different n-grams

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.

Its not perfect, since its not capturing long term dependencies, but one can get quite far with it. N-grams require accounting for V^n events. $V = 10^4, n = 7 \rightarrow 10^{28}$ events.

13.3 Estimating bigram probabilities

Maximum likelihood estimate

$$P(w_i|w_{i-1}) = \frac{count(w_{i-1}, w_i)}{count(w_{i-1})} \tag{35}$$

Given the examples $< s >$ I am Sam $< /s >$, $< s >$ Sam I am $< /s >$, $< s >$ I do not like green eggs and ham $< /s >$, we estimate the values as follows

$$P(\text{I}| < s >) = \frac{2}{3} = 0.67 \qquad P(\text{Sam}| < s >) = \frac{1}{3} = 0.33$$
$$P(< /s > |\text{Sam}) = \frac{1}{2} = 0.5 \qquad P(\text{Sam}|\text{am}) = \frac{1}{2} = 0.5$$

Bigram counts:

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

Normalized bigram counts:

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

	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

We calculate the probability of a sentence, by multiplying the bigram estimates.
P($< s >$ I want english food $< /s >$) = P(I| $< s >$) $\times$ P(want| I) $\times$ P(english | want) $\times$ P(food| english) $\times$ P($< /s >$ | food) = 0.000031.
In practice: log P(I| $< s >$) + log P(want| I) + log P(english | want) + $\cdots$

What did the language model learn?

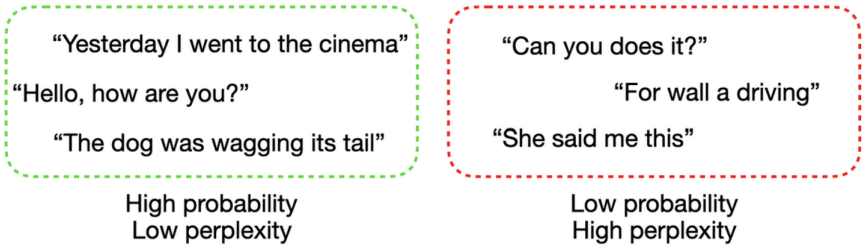
- World knowledge
  - P(english|want) = 0.0011
  - P(chinese|want) = 0.0065
- Grammar (infinitive verb)
  - P(to|want) = 0.66
  - P(to|eat) = 0.28
- Ungrammatically
  - P(food|to) = 0
  - P(want|spend) = 0
- People like to talk about themselves
  - P(I| $< s >$) = 0.25

14 Language Model Evaluation

How good is our model? Does our language model prefer "good" sentences to "bad" ones? Assign higher probability to "real" or "frequently observed" sentences than "ungrammatical" or "rarely observed" sentences?
We train parameters of our model on a training set. We test the model's performance on data we haven't seen. An evaluation metric tells us how well our model does on the test set.

14.1 Perplexity

In practice, we do not use raw probability, but its variant, perplexity. Perplexity (PP or PPL) is the inverse probability of the test set, normalized by the number of words. Minimizing perplexity is the same as maximizing probability.



Intuition tells us, a better model of text is one that assigns a higher probability to the word that actually occurs. In turn that means, a model is less "perplexed" by unseen data. Lower perplexity is better.

$$PP(W) = P(w_1\, w_2\, \ldots\, w_N)^{-\frac{1}{N}} = \sqrt[N]{\frac{1}{P(w_1\, w_2\, \ldots\, w_N)}}$$

Using the Chain Rule

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

For Bigrams

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

In practice we use log probabilities. Perplexity is then also closely related to entropy.

$$\ell(\mathbf{w}) = \sum_{m=1}^{M} \log P(w_m \mid w_{m-1}, \ldots, w_1) \qquad \mathrm{PP}(W) = 2^{-\frac{\ell(w)}{M}}$$

entropy: avg negative log likelihood per word

What is a good perplexity? Training 38 million words, test 1.5 million words, financial news (Wall Street Journal)

n-gram order	unigram	bigram	trigram
perplexity	962	170	109

RANK	MODEL	TEST PERPLEXITY ↓	VALIDATION PERPLEXITY	PARAMS	EXTRA TRAINING DATA	PAPER	CODE	RESULT	YEAR
1	GPT-3 (Zero-Shot)	20.5		175000M	✓	Language Models are Few-Shot Learners			2020
2	BERT-Large-CAS	31.3	36.1	395M	×	Language Models with Transformers			2019
3	GPT-2	35.76		1542M	✓	Language Models are Unsupervised Multitask Learners			2019
4	Mogrifier LSTM + dynamic eval	44.9	44.8	24M	×	Mogrifier LSTM			2019

15 Language Model Generalization

15.1 Zero N-grams

Bigrams with zero probability mean that we will assign 0 probability to the test set! We can not compute perplexity, since $\log(0)$ is undefined. Let's look at an example. Shakespeare's works $N = 884,647$ tokens, $V = 29,066$ types, $300,000$ bigram types out of $V2 = 884$ million possible bigrams 99.96% of the possible bigrams are not observed in the corpus. These bigrams are called zero bigrams.

15.2 N-gram Smoothing

As with counts for text classification, we can use smoothing to improve generalization. The intuition is to steal some probability mass from observed n-grams and distribute the wealth.

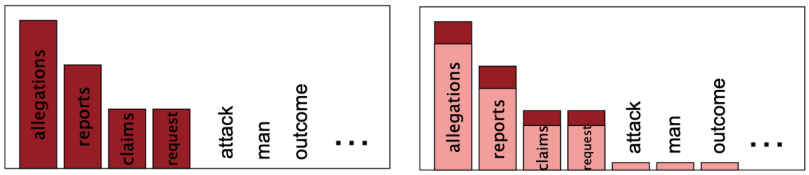
training set:

...denied the allegations
...denied the reports
...denied the claims
...denied the request

test set:

...denied the offer
...denied the loan

$P(offer \mid denied\ the) = 0$



15.2.1 Laplace Smoothing

One ways of doing this is the Laplace smoothing. We pretend we saw each word one more time than we did. Just add one to all the counts!

	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

This method is not used often in practice, since it is too simple to work appropriate in most applications.

15.2.2 Absolute discounting

Another way to go about it is **absolute discounting as smoothing**. We subtract some mass rather than adding some. d is some reasonable constant, like 0.75.

$$P_{\text{AbsoluteDiscounting}}(w_i \mid w_{i-1}) = \underbrace{\frac{c(w_{i-1}, w_i) - d}{c(w_{i-1})}}_{\text{discounted bigram}} + \underbrace{\lambda(w_{i-1})}_{\text{Interpolation weight}} \underbrace{P(w)}_{\text{unigram}}$$

But should the unigram probability $P(w)$ stay as is? Consider the following example (Shannon game)

I can't see without my reading ___glasses___ ?

I can't see without my reading ___Kong___ ?

Kong is more common than glasses, $P(\text{Kong}) > P(\text{glasses})$, but Kong very frequently follows Hong.

15.2.3 Kneser-Ney

Kneser-Ney smoothing is a better estimate for unigrams. Instead of $P(w)$ "How likely is w" compute $P_{\text{CONTINUATION}(w)}$ "How likely is w to appear as a novel continuation?" For each word, count the number of bigram types it completes. Every bigram type was a novel continuation the first time it was seen. A frequent word (like "Kong") occurring in only one context ("Hong")

$$\lambda(w_{i-1}) = \frac{d}{c(w_{i-1})} \left|\{w : c(w_{i-1}, w) > 0\}\right|$$

the normalized discount

The number of word types that can follow w_{i-1}
= # of word types we discounted
= # of times we applied normalized discount

will have a low continuation probability. Intuitively the unigram is useful exactly when we haven't seen the bigram.

$$P_{\text{CONTINUATION}}(w) = \frac{|\{w_{i-1} : count(w_{i-1} > 0\}|}{|\{(w_{j-1}, w_j) : count(w_{j-1}, w_j) > 0\}|}$$

$$P_{KN}(w_i|w_{i-1}) = \frac{\max(count(w_{i-1}, w_i) - d, 0)}{count(w_{i-1})} + \lambda(w_{i-1}) P_{\text{CONTINUATION}}(w_i)$$

15.3 Backoff and Interpolation

Sometimes it helps to use less context. Intuitively we weight context less for contexts you have not learned much about. Two ways to use less context (in e.g. trigram model). **Backoff**: Use trigram counts if you have good evidence; otherwise, bigram; otherwise, unigram. **Interpolation**: Always mix unigram, bigram, trigram statistics.
Interpolation works better. State-of-the-art is extended interpolated Kneser-Ney smoothing.
Linear interpolation is a linear combination of unigram, bigram and trigram probabilities.
Simple interpolation:

$$\hat{P}(w_i|w_{i-1}, w_{i-2} = \lambda_1 P(w_i|w_{i-1}, w_{i-2}) + \lambda_2 P(w_i|w_{i-1}) + \lambda_3 P(w_i)$$
$$\text{with} \quad \sum_i \lambda_i = 1$$

Better: λ_i depends on specific context

$$\hat{P}(w_i|w_{i-1}, w_{i-2}) = \lambda_1(w_{i-2}^{i-1}) P(w_i|w_{i-1}, w_{i-2}) + \lambda_2(w_{i-2}^{i-1}) P(w_i|w_{i-1}) + \lambda_3(w_{i-2}^{i-1}) P(w_i)$$

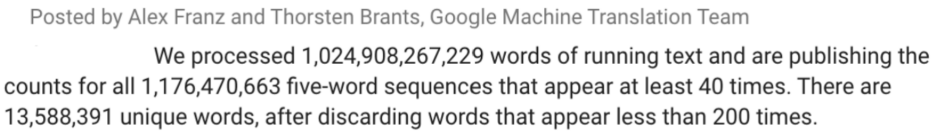
15.4 Unseen vs Unknown words

What if you see at test time a word you did not know is in the vocabulary? We divide this into two separate tasks. **Closed vocabulary task**; we know all possible words in advance; V is fixed. **Open vocabulary task**; we may encounter out-of-vocabulary (OOV) words.
A common solution is to create an unknown word token: $< UNK >$

- Choose a fixed vocabulary of size V in advance.
- Convert any token in training set, which can not be found in V to $< UNK >$.
- Estimate probabilities of $< UNK >$ just like any other word in the training set.

Another solution is using a sub-word representation (e.g. FastText word embeddings)
skiing = {skiing, ski, skii, kiin, iing, ing}

15.5 Data size matters


Posted by Alex Franz and Thorsten Brants, Google Machine Translation Team

We processed 1,024,908,267,229 words of running text and are publishing the counts for all 1,176,470,663 five-word sequences that appear at least 40 times. There are 13,588,391 unique words, after discarding words that appear less than 200 times.

https://ai.googleblog.com/2006/08/all-our-n-gram-are-belong-to-you.html(Google N-grams)

Some computing tricks for large data **Pruning**:

- Only store n-grams with count > threshold
- Entropy-based pruning of prune n-grams that aren't informative

Computational efficiency:

- Efficient data structures, such as tries.
- Bloom filters: approximate language models.
- Huffman coding more efficient than strings: stuff many words into two bytes.
- Quantization: store probabilities in 4-8 bits rather than 32 bit float (mapping continuous infinite values to a smaller set of discrete values).

Some open-source statistical language modelling tools:
http://www.speech.sri.com/projects/srilm/
https://kheafield.com/code/kenlm/

16 Neural Language Models

Here a Neural Network Language Model from 2001

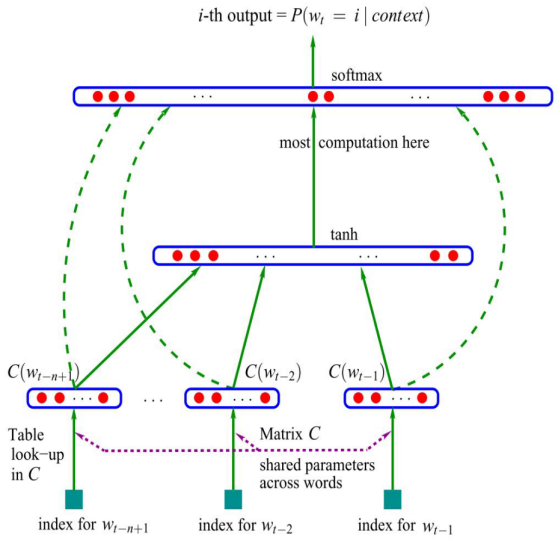
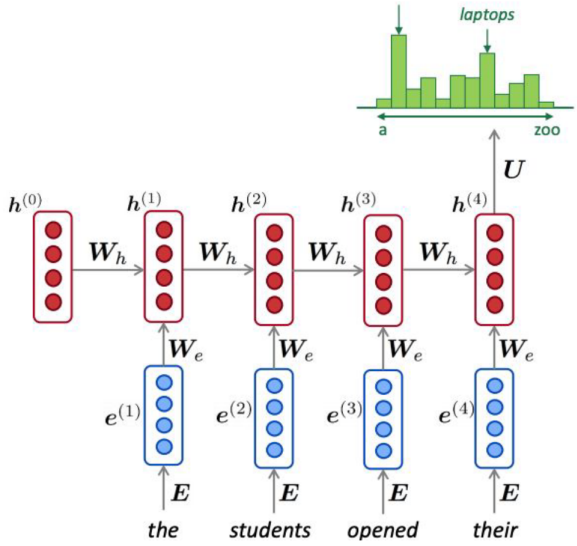


Figure 18: Bengio, Y., Ducharme, R., & Vincent, P. (2001). A Neural Probabilistic Language Model. Proceedings of NIPS.

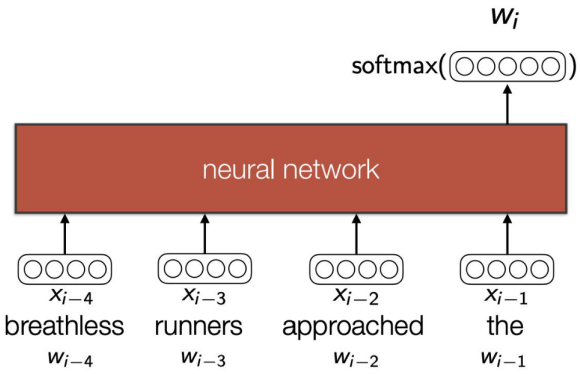
Neural Network Language Models have better empirical performance at language modeling (lower perplexity, better generalization, longer context). Neural Networks provide state-of-the-art dense word representations (used instead of word embeddings these days). But it needs a lot more computation, therefor it is slower to train and test.



The idea is to predict probabilities rather than counting. Our input is made of word embeddings $(x_1, x_2, \ldots, x_N)$. The output is $P(w_i|w_{i-1}, w_{i-2}, \ldots, w_2, w_1)$

16.1 Option 1: Feed-forward neural LM

Let's fix the window size (e.g. $k = 4$ words). As input, concatenated k word embeddings into one long vector $x = [x_{i-4}, x_{i-3}, x_{i-2}, x_{i-1}]$.



The model has a fully connected hidden layer $f(\Theta^{(x\to z)}x)$. The output is the softmax $(\Theta^{(z\to y)}z)$.

$$P(w_i = j|z) = \frac{\exp(\theta_j^{(z\to y)} \cdot z)}{\sum_{j' \in V} \exp(\theta_{j'}^{(z\to y)} \cdot z)} \tag{36}$$

How does this compare to probabilistic n-gram LMs from the last lecture?

Improvements:

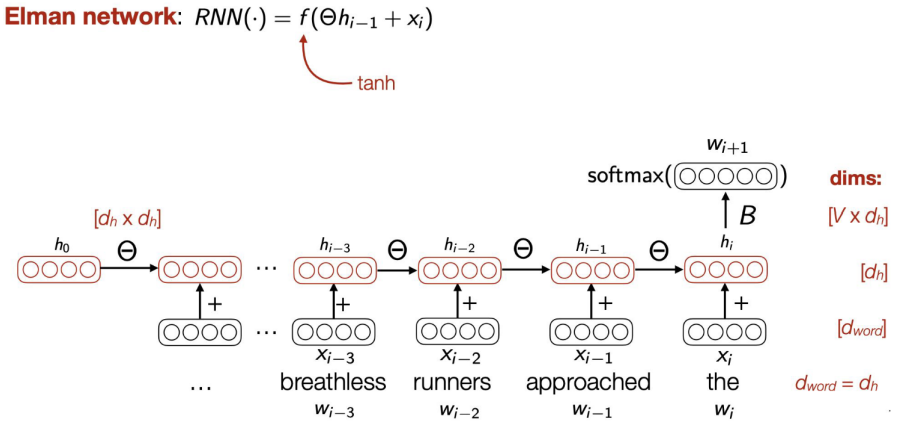
- The model size is $O(V)$, not $O(V^n)$.
- The lack of sparsity.
- Some sharing of representation across words.

Remaining challenges:

- Still not enough context. Larger context grows $\Theta^{(x \to z)}$ linear in n.
- Each x_i uses different rows of $\Theta^{(x \to z)}$. The weights are not shared across words.

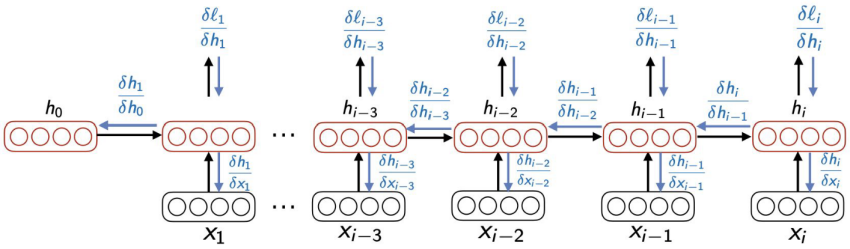
16.2 Option 2: Recurrent neural LM

2010 Recurrent Neural Networks, are being applied more and more. Maintain a context vector h. At each time-step (w_j), compose the context with the current word x_j to create a new context for the next time-step, $h_j = RNN(x_j, h_{j-1})$. These Networks are recurrent because we repeatedly apply the same function $RNN(\cdot)$ each time to consider previous step. If $d_{\text{word}} = d_h$, we can tie input X and output B word embeddings.



If we don't want to tie X with B, we can also add another projection matrix $\Theta^{(x \to h)}$.

Learning recurrent neural LMs: Backpropagation through time
In addition to each layer, now gradients for the stochastic gradient descent must be computed with respect to each time-step as well. "Unroll" the RNN and treat the parameters at each time-step as if it were another layer; apply the chain rule.
(for details, see the NLP Lecture 3 or the corresponding chapter in Jacob Eisenstein's book)



How does this compare to probabilistic n-gram LMs from the last lecture?

Improvements:

- Model size $O(V)$, not $O(V^n)$.
- The lack of sparsity.
- Sharing of representation across words.
- Models long context.

Remaining Challenges

- Softmax over large vocabulary.
- High variance/over-fitting.
- Exploding and vanishing gradients.

16.3 Softmax over large vocabulary

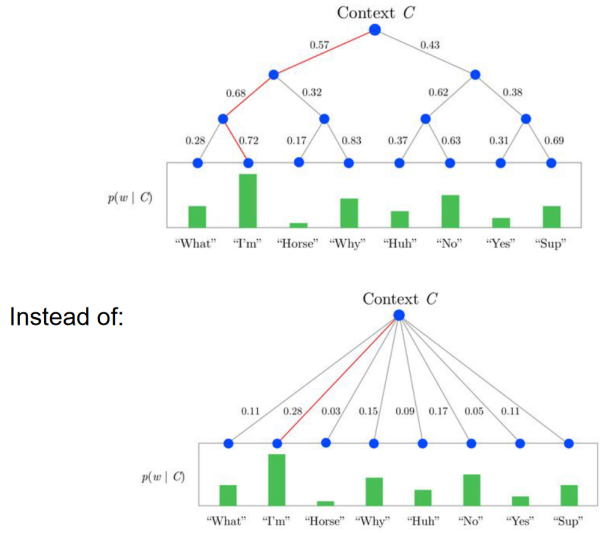
It is inconvenient to score the next word over the entire vocabulary (similar issues as in the n-gram model).

Some solutions:

- Hierarchical softmax
- Noise-contrastive estimation (NCE)
- Adaptive softmax

16.3.1 Hierarchical Softmax

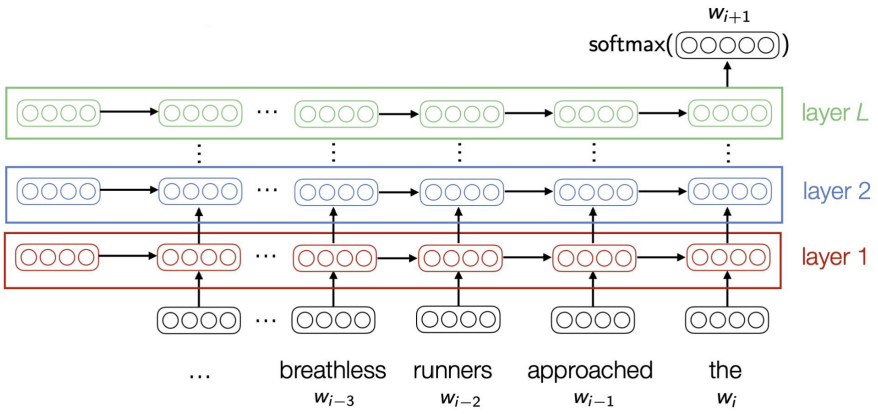
The hierarchical Softmax uses a multi-layer binary tree, where the probability of a word is calculated through the product of probabilities on each edge on the path to that node.



17 Using Neural Language Models for Word Representation

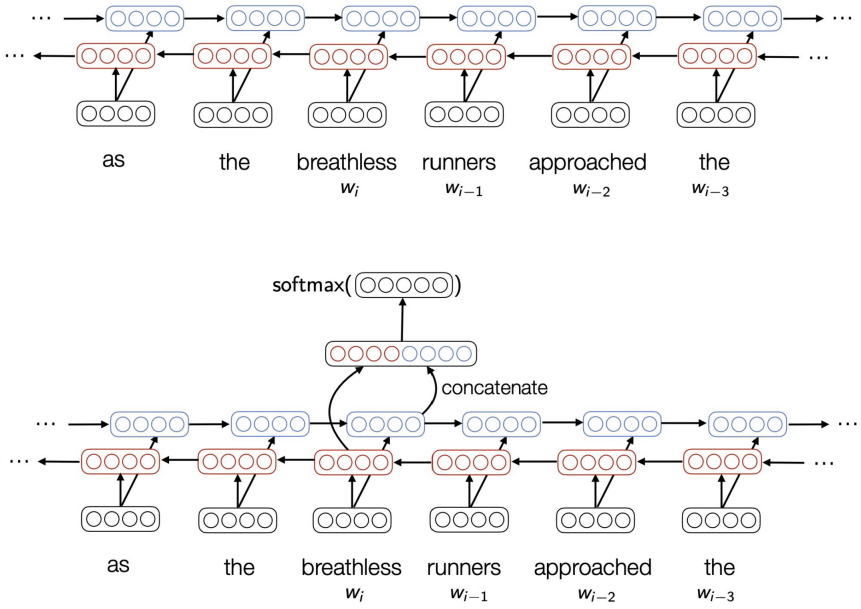
17.1 Stacking recurrent layers

Just as we can add many hidden layers to feed-forward-networks, we can add many RNN-layers (**Stacked RNN**).



17.2 Bidirectional RNN

In language it is also often useful to model past and future context. We can also run an RNN in the backward direction (reverse reading order). Combining both directions works best.



17.3 Using LMs as word representations

The issue with word2vec is, there is one representation for all senses and part-of-speech of the same word. This leads to complications highlighted by the following example.

"The new-look **play** area is due to be completed by early spring 2020."
"Gerrymandered congressional districts favor representatives who **play** to the party base."
"The freshman then completed the three-point **play** for a 66-63 lead."

Homonyms:

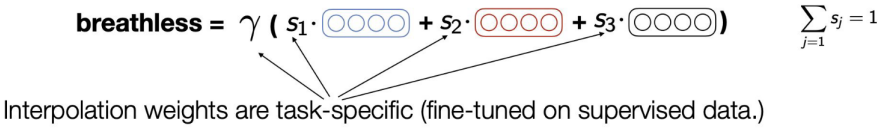
She **left** her book in her locker last night.
My **left** foot really hurts.

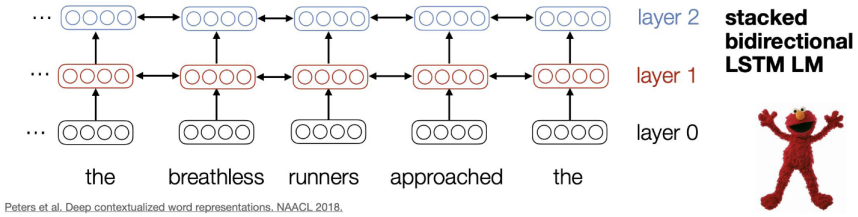
We want **contextualized** word embeddings instead. Under contextualization we understand representations of a word that differs depending on the context, but still leverage the unsupervised training on massive data. A solution is to use the hidden layer representations from a neural language model.

17.4 ELMo: Deep contextualized word embeddings

ELMo stands for **Embeddings from Language Models**. https://arxiv.org/pdf/1802.05365.pdf The key idea is context depending embeddings

for each word interpolates representations for that word from each layer.





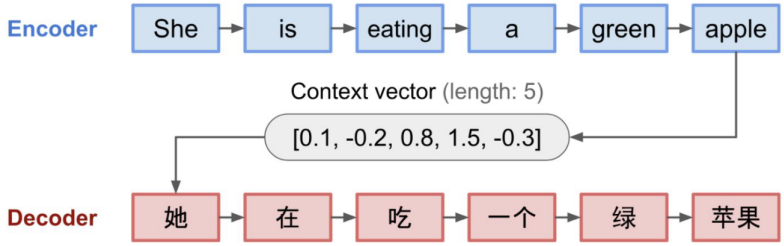
	TASK	BASELINE	ELMO + BASELINE
question answering	SQuAD	81.1	85.8
natural language inference	SNLI	88.0	88.7 ± 0.17
semantic role labeling	SRL	81.4	84.6
coreference	Coref	67.2	70.4
named entity recognition	NER	90.15	92.22 ± 0.10
sentiment analysis	SST-5	51.4	54.7 ± 0.5

18 Using Attention-Based Neural Language Models for Word Representation

Some words in the sentence are more important when trying to interpret new words.

We remember, sequence to sequence modeling (e.g. machine translation) with an encoder-decoder RNN architecture. An encoder processes the input sequence and compresses the information into a vector (sentence embedding) of a fixed length. This representation is expected to be a good summary of the meaning of the whole source sequence.

A decoder is initialized with the context vector to emit the transformed output. The early work only used the last state of the encoder network as the decoder initial state. The problem lies with capturing long contexts (forgetting).



Instead of building a single context vector out of the encoder's last hidden state, create shortcuts between the context vector and the entire source input:

- Encoder hidden states
- Decoder hidden states
- Alignment between source and target

The weights of these shortcut connections are customizable for each output.

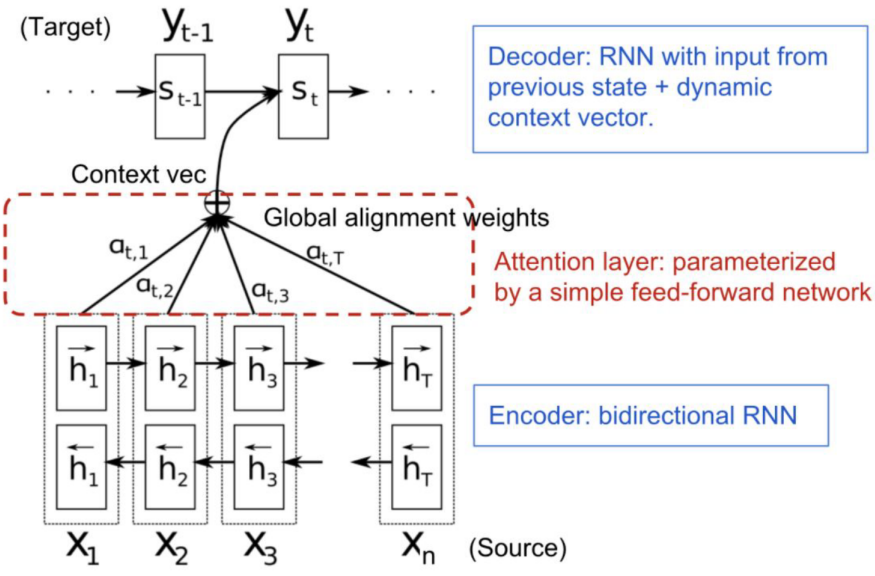
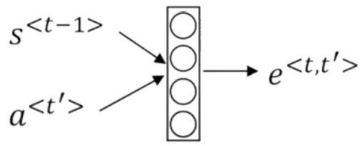
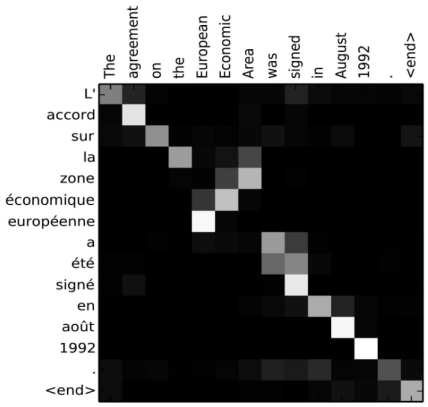


Figure 19: https://lilianweng.github.io/posts/2018-06-24-attention/

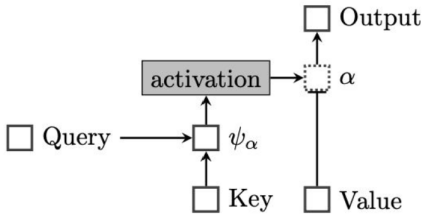
To compute the attention, learn the parameters alpha (the attention weights). For example just by adding a fully connected DNN with one hidden layer. The goal is to predict how well an input word aligns with an output word (an alignment score e t, t').

$$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t'=1}^{T_x} \exp(e^{<t,t'>})}$$





Neural attention mechanism is a weighted average over (word) representations, where the weights are learned, computed as a function of the input. Additive attention ψ is a feed-forward layer with concatenated $[k; q]$ as input. Multiplicative attention ψ is a dot product of linear projections of k, q.

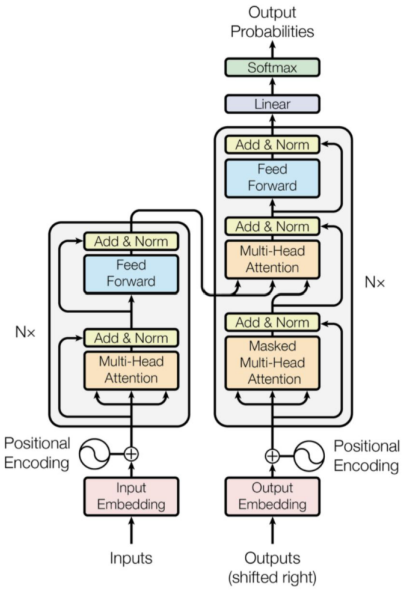


Self-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the same sequence. Self-attention is the method the Transformer uses to integrate the "understanding" of other relevant words into the one we're currently processing.

19 The Transformer

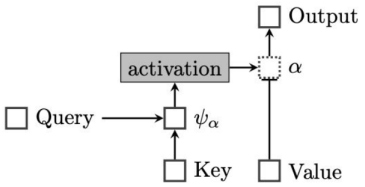
Like LSTM, Transformer is an architecture for transforming one sequence into another one with the help of two parts (Encoder and Decoder). But it differs from the previously described sequence-to-sequence models because it does not use any Recurrent Networks. Both Encoder and Decoder are composed of stacked modules that consist mainly of **Multi-Head Attention**

and Feed Forward layers.



19.1 Transformer self-attention

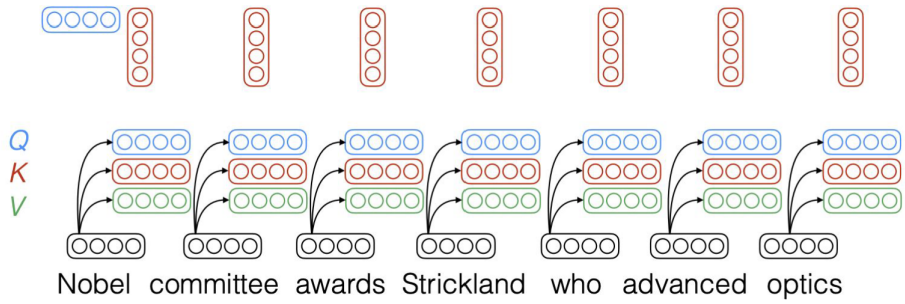
Q is a matrix that contains the query (vector representation of one word in the sequence). K are all the keys (vector representations of all the words in the sequence). V are the values, which are again the vector representations of all the words in the sequence.



$$Q = W^Q X;\ K = W^K X;\ V = W^V X$$

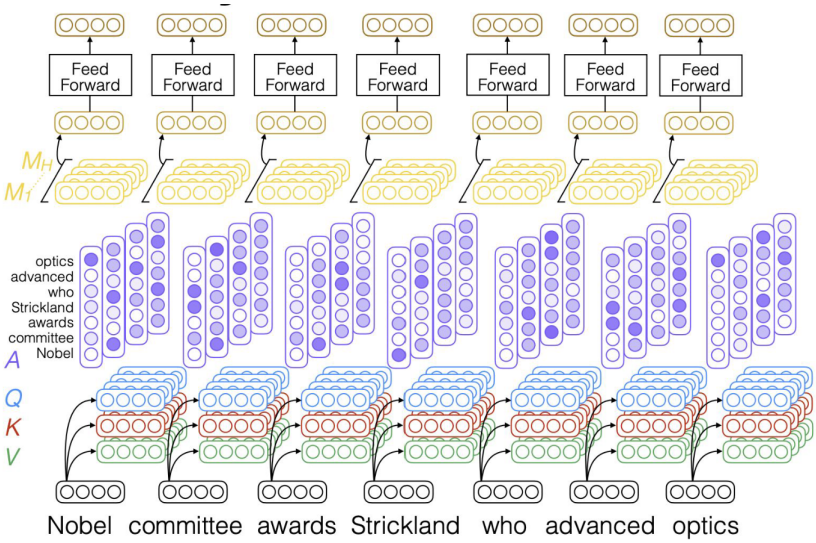
$$a = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

$$SelfAttention(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



19.2 Transformer multihead self-attention

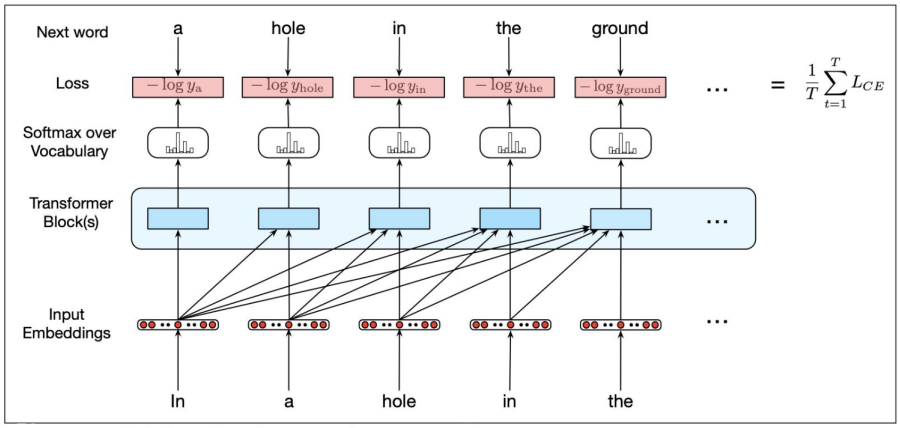
Different words in a sentence can relate to each other in many different ways at once. It is difficult to capture all of the different kinds of relations. Transformers address this issue with multihead self-attention layers (sets of self-attention layers, called heads, that reside in parallel at the same depth in a model, each with its own set of parameters).
This way each head can learn different aspects of the relationships among inputs at the same level of abstraction.
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 feed-forward layers, residual connections and normalizing layers.



These blocks can then be stacked just as was the case for stacked RNNs.

19.3 Training a transformer LM

This is a somewhat similar setup as RNN, but can be much easier parallelized (everything happens simultaneously, no recurrence). Generally, requires lots of computing power to train. The GPT-3 model requires 175 billion parameters and 22 graphic processors that would cost somewhere between $4.6 million and $12 million to train. BERT large has 345 million parameters (https://github.com/google-research/bert). For ELMo, the LSTM has about 100 million parameters and the softmax layer 400 million parameters.

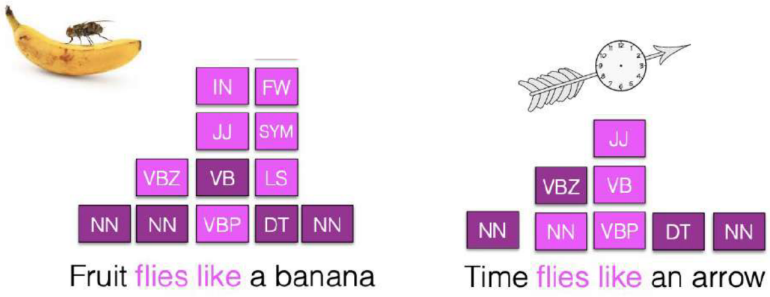


19.4 Additional Resources

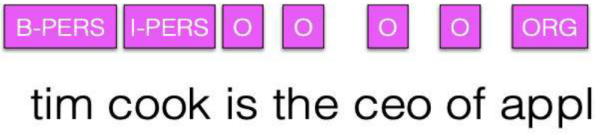
https://web.stanford.edu/~jurafsky/slp3/9.pdf [Neural Language Models (chapter)]
https://arxiv.org/abs/1510.00726 [Introduction to Neural Networks in NLP]
IntroductiontoNeuralNetworksinNLP [ELMo (original paper)]
https://ruder.io/nlp-imagenet/ [Elmo explained]
https://arxiv.org/abs/1706.03762 [Transformers (original paper)]
https://jalammar.github.io/illustrated-transformer/ [Transformers explained]
https://arxiv.org/abs/1810.04805 [BERT Transformer]
https://www.aclweb.org/anthology/2020.emnlp-demos.6.pdf [Transformers performance overview and library]

20 Sequence Labeling

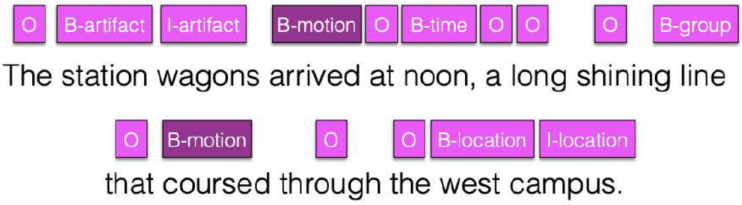
One sequence classification problems is **Part-of-Speech tagging** (e.g. recognize verbs in a sentence).



Another sequence classification problem is **Named Entity Recognition** (e.g. find company names in tweets).



Furthermore there is **Supersense tagging** (e.g. find all words describing an animal or a fruit), **Semantic labeling** (e.g. find who in the sentence does what to whom), **Slot classification for chatbots** (e.g. find a song title in a play command from the user) and many more.



1	person	7	cognition	13	attribute	19	quantity	25	plant
2	communication	8	possession	14	object	20	motive	26	relation
3	artifact	9	location	15	process	21	animal		
4	act	10	substance	16	Tops	22	body		
5	group	11	state	17	phenomenon	23	feeling		
6	food	12	time	18	event	24	shape		

For a set of inputs x with n sequential time steps, we want one corresponding label y_i for each x_i.

20.1 Part-of-Speech Tagging

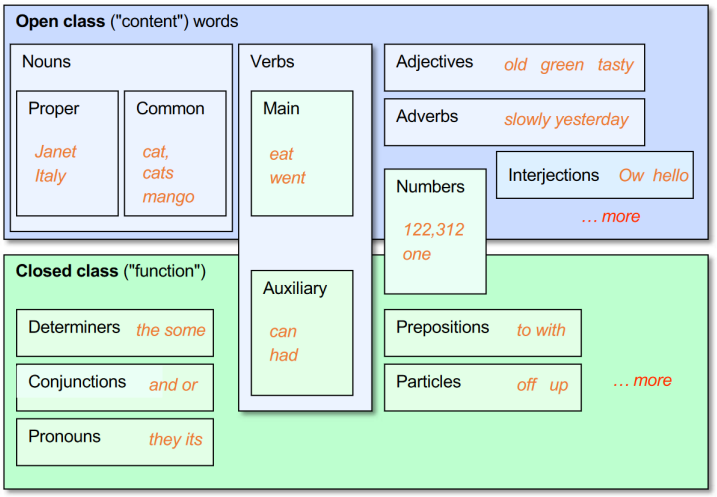
From the earliest linguistic traditions (Aristoteles), the idea was that words can be classified into grammatical categories (called part of speech, word classes, POS, POS tags, ...), such as noun, verb, pronoun, preposition, adverb, conjunction, participle and article. There are open and closed word classes. Closed class words rarely change, these are usually function words; short, frequent words with grammatical function:

- Determiners: a, an, the
- Pronouns: she, he, I
- Prepositions: on, under, over, near, by, ...

Open class words often get extended:

- Usually content words: Nouns, Verbs, Adjectives, Adverbs

- Plus interjections: oh, ouch, uh-huh, yes, hello
- New nouns and verbs like iPhone or to fax



Part-of-speech (POS) Tagging, is assigning a part-of-speech label to each word in a text. Words often have more than one POS, so we need to disambiguate these for practical applications (translation, meaning analysis, pronunciation...). For example "book" has multiple meanings depending on the context.

- VERB: (Book that flight)
- NOUN: (Hand me that book)

Universal Dependencies tag-set (on the picture), can be used across many languages.

	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

There/PRO were/VERB 70/NUM children/NOUN there/ADV ./PUNC
Preliminary/ADJ findings/NOUN were/AUX reported/VERB in/ADP
today/NOUN 's/PART New/PROPN England/PROPN Journal/PROPN
of/ADP Medicine/PROPN

Why is this useful? Language-analytic computational tasks, have the need to control for POS when studying linguistic change like creation of new words, or meaning shift, or in measuring meaning similarity or difference. Use cases are

- Syntactic parsing: better understanding of the structure of the sentence tree.
- Machine Translation: reordering of adjectives and nouns (say from Spanish to English).
- Sentiment or affective tasks: may want to distinguish adjectives or other POS.
- Text-to-speech (how do we pronounce "lead" or "object"?).

20.1.1 Part-of-Speech Tagging Features

Given the example

Janet will back the bill
AUX/NOUN/VERB? NOUN/VERB?

Prior probabilities of word/tag, shows "will" is usually an AUX. To tag words it can be helpful to identity neighboring words. "the" means the next

word is probably not a verb.

Morphology and word-shape:

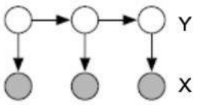
- Prefixes unable: un- → ADJ
- Suffixes importantly: -ly → ADJ
- Capitalization Janet: CAP → PROPN

20.2 Sequence Classification Models

HMM is generative, needs many samples to calculate prior probabilities and determine posterior probabilities (remember Naive Bayes, but now we have sequences). CRF is discriminative, directly predicting the posterior probabilities (remember Logistic Regression, but now we have sequences), uses features over all pairs. Other alternatives such as RNN or transformers are also possible.

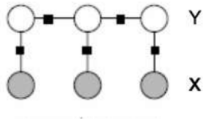
Hidden Markov Model (HMM)

$$p(\mathbf{y}, \mathbf{x}) = \prod_{t=1}^{\mathrm{T}} p(y_t|y_{t-1})p(x_t|y_t)$$



Conditional Random Fields (CRF)

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{Z(\mathbf{x})} \exp\left\{\sum_{k=1}^{K} \lambda_k f_k(y_t, y_{t-1}, \mathbf{x}_t)\right\}$$



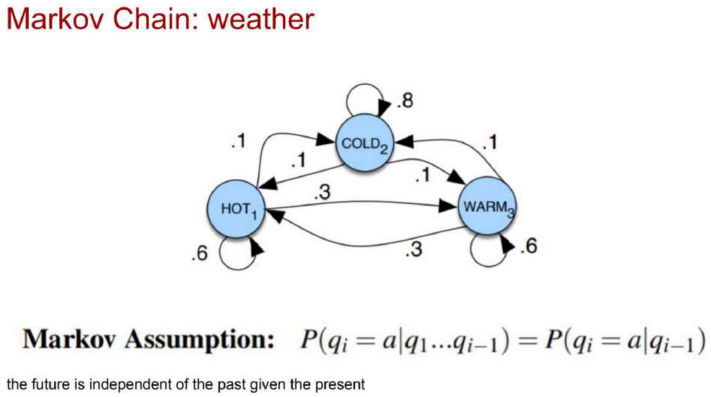
21 Hidden Markov Models

The idea is to look at both the previous labels in the sequence, and the properties of the current input. "look at" = take the conditional probability. How likely is a possible current label considering the previous label? How likely is the current input considering a possible current label?
Similarly to the language models, we are using the Markov assumption here; it is sufficient to look at one previous step $(t-1)$ as a proxy for all the other previous steps.

21.1 Markov Chain

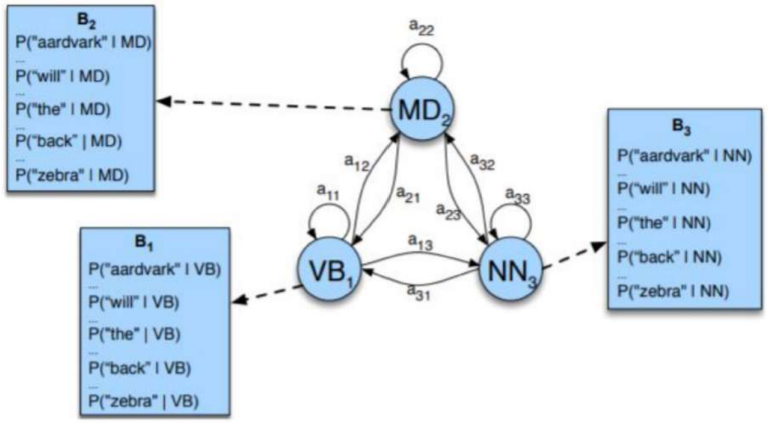
$Q = \{q_1, q_2, \ldots, q_N\}$ is a set of N **states**. $A = \{a_{1,1}, a_{1,2}, \ldots, a_{1,n}, \ldots, a_{n,n}\}$ a **translation probability matrix**. Each $a_{i,j}$ representing the probability to move from state i to state j. It holds $\sum_{j=1}^{n} a_{i,j} = 1, \forall i$. $\pi =$

$\{\pi_1, \pi_2, \ldots, \pi_N\}$ is an **initial probability distribution** over states. $\pi_\mathbf{i}$ is the probability that the Markov Chain starts in state i. Some states j may have $\pi_j = 0$, meaning that they can not be the initial state. It holds, $\sum_{i=1}^{n} \pi_i = 1$.



21.2 Hidden Markov Model

The "Hidden" Markov model extends the Markov Chain. It has all the properties of the Markov Chain. Additionally $O = \{o_1, o_2, \ldots, o_T\}$ is a sequence of T **observations**, each one draw from a vocabulary $V = \{v_1, v_2, \ldots, v_V\}$ and $B = b_i(o_t)$ is 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.



Here we make two assumptions. The probability of a word occurrence is only dependent on its own tag and is independent of neighbor words and tags. The probability of a tag is dependent only on the previous tag, rather than the entire tag sequence.
Similarly to the language models, we are using the **Markov assumption**

here; it is sufficient to look at **one** previous step $(t-1)$ as a proxy for all the other previous steps.