Human Language Technologies

Federico Matteoni

A.A. 2021/22

Index

0.1 Introduction

Prof. Giuseppe Attardi
Prerequisites are: proficiency in Python, basic probability and statistics, calculus and linear algebra and notions of machine learning.

What will we learn Understanding of and ability to use effective modern methods for **Natural Language Processing**. From traditional methods to current advanced ones like RNN, Attentions...
Understanding the difficulties in dealing with NL and the capabilities of current technologies, with experience with **modern tools** and aiming towards the ability to build systems for some major NLP tasks: word similarities, parsing, machine translation, entity recognition, question answering, sentiment analysis, dialogue system...

Books Speech and Language Processing (Jurafsky, Martin), Deep Learning (Goodfellow, Bengio, Courville), Natural Language Processing in Python (Bird, Klein, Loper)

Exam Project (alone or team of 2-3 people) with the aim to experiment with techniques in a realistic setting using data from competitions (Kaggle, CoNLL, SemEval, Evalita...). The topic will be proposed by the team or chosen from a list of suggestions.

Experimental Approach

1. Formulate hypothesis
2. Implement technique
3. Train and test
4. Apply evaluation metric
5. If not improved:

   Perform error analysis

   Revise hypothesis
6. Repeat!

Motivations Language is the most distinctive feature of human intelligence, **it shapes thought**. Emulating language capabilities is a scientific challenge, a **keystone for intelligent systems** (see: Turing test)

Structured vs unstructured data The largest amount of information shared with each other is unstructured, primarily text. Information is mostly communicated by e-mails, reports, articles, conversations, media... and attempts to turn text to structured (HTML) or microformat only scratched the surface.
Problems: requires universal agreed **ontologies** and additional effort. Entity linking attempts to provide a bridge.

0.2 State of the Art

Early History During 1950s, up until AI winter.

Resurgence in the 1990s Thanks to statistical methods, novelty, to study language. Challenges arise: NIST, Netflix, DARPA Grand Challenge...
During 2010s: deep learning, neural machine translation...

Statistical Machine Learning Supervised training with **annotated** documents.
The paradigm is composed of the following:

Training set $\{x_i, y_y\}$

Representation: choose a set of features to represent data $x \mapsto \phi(x) \in R^D$

Model: choose an hypothesis function to compute $f(x) = F_\Theta(\phi(x))$

Evaluation: define the cost function on error with respect to examples $J(\Theta) = \sum_i (f(x_i) - y_i)^2$

Optimization: find parameters Θ that minimize $J(\Theta)$

It's a generic method, applicable to any problem.

Traditional Supervised Learning Approach Freed us from devising algorithms and rules, requiring the creation of annotated training sets and imposing the tyranny of feature engineering.
Standard approach for each new problem:

Gather as much labeled data as one can

Throw a bunch of models at it

Pick the best

Spend hours hand engineering some features or doing feature selection/dimensionality reduction

Rinse and repeat

Technological Breakthroughs Improved ML techniques but also large annotated datasets and more computing power, provided by GPUs and dedicated ML processors (like the TPU by Google).
ML exploits parallelism: stochastic gradient descent can be parallelized (asynchronous stochastic gradient descent). No need to protect shared memory access, and low (half, single) precision is enough.

Deep Learning Approach Was a big breakthrough.

Design a model architecture

Define a loss function

Run the network letting the parameters and the data representations **self-organize** as to minimize the loss

End-to-end learning: no intermediate stages nor representation

Feature representation Use a vector with each entry representing a feature of the domain element
Deep Learning represents data as vectors. Images are vectors (matrices), but words? **Word Embeddings**: transform a word into a vector of hundreds of dimensions capturing many subtle aspects of its meaning. Computed by the means of **language model**.
From a discrete to distributed representation. Words meaning are dense vectors of weights in a high dimensional space, with algebraic properties.
Background: philosophy, linguistics and statistics ML (feature vectors).

Language Model Statistical model which tells the probability that a word comes after a given word in a sentence.

Dealing with Sentences A sentence is a sequence of words: build a representation of a sequence from those of its words (compositional hypothesis). Sequence to sequence models.
Is there more structure in a sentence than a sequence of words? In many cases, tools forgets information when translating sentences into sequences of words, discarding much of the structure.

0.3 Language Modeling

Probabilistic Language Model The goal is to assign a probability to a sentence.

Machine Translation: P(high winds tonight) $>$ P(large winds tonight)

Spell Correction: P(about fifteen minutes from) $>$ P(about fifteen minuets from)

Speech Recognition: P(I saw a van) $>$ P(eye saw a van)

Language Identification: s from unknown language (italian or english) and Lita, Leng language models for italian and english $\Rightarrow$ Lita$(s) > L$eng(s)

Summarization, question answering...

We want to compute

$P(W) = P(w_1, w_2, \ldots, w_n)$ the probability of a sequence

$P(w_4 \mid w_1, w_2, w_3, w_4)$ the probability of a word given some previous words

The model that computes that is called the **language model**

Markov Model and N-Grams Simplify the assumption: the probability of a word given all the previous is the same of the probability of that word given just few (one, two...) previous words. So $P(w_i \mid w_{i-}, \ldots, w_1) = P(w_i \mid w_{i-1})$ (First order Markov chain).
With a N**-gram**: $P(w_n \mid w_1^{n-1}) \simeq P(w_n \mid w_{n-N+1}^{n-1})$
In general it's insufficient: language has **long distance dependencies**, but we can often get away with N-gram models. For example:

"The **man** next to the large oak tree near the grocery store on the corner **is** tall."

"The **men** next to the large oak tree near the grocery store on the corner **are** tall."

Or even semantic dependencies:

"The **bird** next to the large oak tree near the grocery store on the corner **flies** rapidly."

"The **man** next to the large oak tree near the grocery store on the corner **talks** rapidly."

So more complex models are needed to handle such dependencies.

Maximum likelihood estimate

$$P(w_n \mid w_{n-N+1}^{n-1}) = \frac{\text{count}(w_{n-N+1}^{n-1}, w_n)}{\text{count}(w_{n-N+1}^{n-1})}$$

Maximum because it's the one that maximize $P(\text{Training set} \mid \text{Model})$

Shannon Visualization Method Generate random sentences:

Choose a random bigram $(\langle s \rangle, w)$ according to its probability

Choose a random bigram (w, x) according to its probability

Repeat until we pick $\langle /s \rangle$

Shannon Game Generate random sentences by selecting the words according to the probabilities of the language models.

```
def gentext(cpd, word, length=20):
  print(word, end = ' ')
  for i in range(length):
    word = cpd[word].generate()
    print(word, end = ' ')
  print("")
```

Perils of Overfitting N-grams only word well for word prediction if the test corpus looks like the training corpus. In real life, this is often not the case.
We need to train robust models able to adapt.

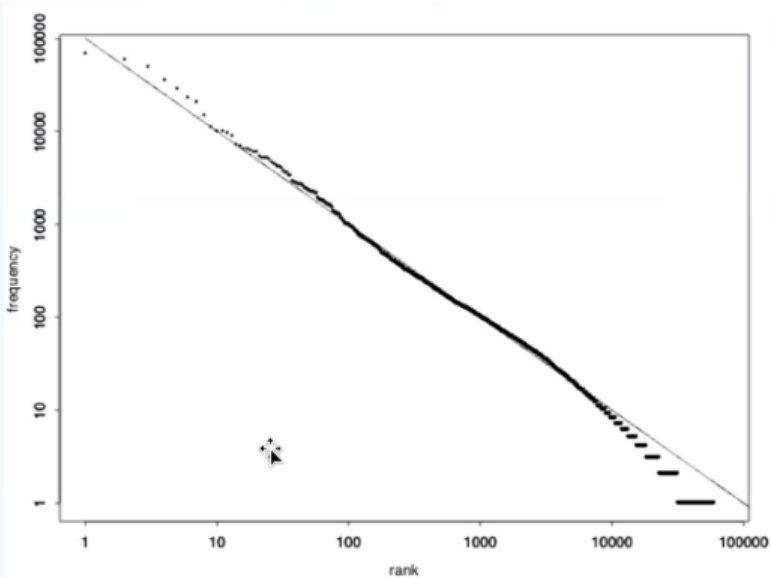
Smoothing Many rare (but not impossible) combinations of word sequences never occur in training, so MLE incorrectly assigns zero to many parameters (sparse data).
If a new combination occurs during testing, it is given a probability of zero and the entire sequence gets a probability of zero (infinite perplexity).
Parameters are **smoothed**/regularized to reassign some probability mass to unseen events (meaning removing probability from seen ones in order to maintain the joint distribution that sums to 1)

Zipf's Law A small number of events occur with high frequency, while a large number of events occur with small frequency. You can quickly collect statistics on the high frequency events, but you may have to wait an arbitrarily long time to get valid statistics on low frequency events.
The result is that our estimates are sparse. No counts at all for the vast bulk of things we want to estimate. Some of the zeroes in the table are really zeroes, but others are simply low frequency events you haven't seen yet: after all, anything can happen. How to address? By estimating the likelihood of unseen N-grams.

Word	Freq. (f)	Rank (r)	$f \cdot r$
the	3332	1	3332
and	2972	2	5944
a	1775	3	5235
he	877	10	8770
but	410	20	8400
be	294	30	8820
there	222	40	8880
one	172	50	8600
about	158	60	9480
more	138	70	9660
never	124	80	9920
Oh	116	90	10440
two	104	100	10400

Word	Freq. (f)	Rank (r)	$f \cdot r$
turned	51	200	10200
you'll	30	300	9000
name	21	400	8400
comes	16	500	8000
group	13	600	7800
lead	11	700	7700
friends	10	800	8000
begin	9	900	8100
family	8	1000	8000
brushed	4	2000	8000
sins	2	3000	6000
Could	2	4000	8000
Applausive	1	8000	8000

Result: f is proportional to $\frac{1}{r}$, meaning that there is a constant $k \mid f \cdot r = k$



0.3.1 Evaluation and Perplexity

Evaluation Train parameters of our model on a training set. For the evaluation, we apply the model on new data and look at the model's performance: **test set**. We need a metric which tells us how well the model is performing: one of such is **perplexity**.

Extrinsic Evaluation Evaluating N-gram models.

Put model A in a task (language identification, speech recognizer, machine translation system...)

Run the task, get an accuracy for A

Put model B in the task, get accuracy for B

Compare the accuracies

Language Identification Task Build a model for each language and compute probability that text is of such language.

$$lang = arg\max_{l} P_l(text)$$

Difficulty of Extrinsic Evaluation Extrinsic evaluation is time-consuming, can take days to run an experiment. Recent benchmarks like GLUE have become popular due to the effectiveness.
Intrinsic evaluation us an approximation called **perplexity**: it's a poor approximation unless the test data looks just like the training data.

Perplexity The intuition is the notion of surprise: how surprised is the language model when it sees the test set? Where surprise is a measure of "I didn't see that coming". The more surprised is, the lower the probability it assigned to the test set, and the higher the probability the less surprised it is.
So perplexity measures how well a model "fits" the test data. It uses the probability that the model assigns to the test corpus and normalizes for the number of words in the test corpus, taking the inverse.

$$PP(w) = \sqrt[N]{\frac{1}{P(w_1 \dots w_N)}}$$

Measures the weighted average branching factor in predicting the next word (lower is better).
In the chain rule

$$PP(w) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i \mid w_1 \dots w_{i-1})}}$$

And for bigrams

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

Lower perplexity $\Rightarrow$ better model

0.4 Representation of Words

Word Meaning Definition of **meaning**:

> Idea represented by a word, phrase...
>
> Idea that a person wants to express by using words, signs...
>
> Idea expressed in a work of writing, art...

Signifier (symbol) $\Leftrightarrow$ **Signified** (idea or concept) = **denotation**

Linguistic Solution Dictionary or lexical resources provide word definitions as well as synonyms, hypernums, antonyms... example: **Wordnet**.

Problems with lexical resources

> Imperfect, sometimes there are errors
>
> Missing new meanings of words, hard to keep up-to-date
>
> Subjective
>
> Require human labor to create and adapt
>
> Hard to compute accurate word similarity

Vector Space Model The **VSM** is a representation for text used in information retrieval. A document is represented by a vector in a n-dimensional space

$$v(d_1) = [t_1, \dots, t_n]$$

Each dimension corresponds to a separate term.
The VSM considers words as discrete symbols.

One Hot Representation A word occurrence is represented by one-hot vectors, with the vector dimension being the number of words in a vocabulary and 1 in correspondence of the position of the represented word. Useful for representing a document by the sum of the word occurrence vector, and document similarity given by **cosine distance** (angle between vectors).

tf*idf Measure Term frequency: frequency of the word.
Inverse document frequency:

Classical VSM Vector is all zeroes with idf_t instead of 1. The vector of a document is the sum of the vectors of its terms occurrence vectors.
Doesn't capture similarity between terms.

Problems with Discrete Symbols All vectors are orthogonal, with no notion of similarity. Search engines try to address the issue using WordNet synonyms but it's better to encode the similarity in the vectors themselves.

Intuition Model the meaning of a word by "embedding" it in a vector space. The meaning of a word is a vectors of numbers (vector models are also called **embeddings**), so a **dense vector space**. Constrast: word meaning is represented in many computational

Word Vectors/Embeddings Build dense vector for each word chosen so that it is similar to vectors of words that appear in similar contexts.
Four kinds:

Sparse Vector Representations

1. Mutual-information weighted word co-occurence matrices

Dense Vector Representations

2.
3.
4.

Distributional Hypothesis A word's meaning is given by the words that frequently appear close-by. Old idea but successful just with modern statistical NLP.
When a word w appears in a text, its context is the set of words that appear nearby (within a fixed-size window). Use of the many contexts of w to build a representation of w.

Word Context Matrix Is a $|V| \times |V|$ matrix X that counts the frequencies of co-occurrence of words in a collection of contexts (i.e. text spans of a given length).

Co-Occurrence Matrix Words $\equiv$ Context words. Rows of X capture similarity yet X is still high dimensional and sparse. One row per word with counts of occurrencies of any other word.
We can compute the distance vectors between words, but neighboring words are not semantically related.

0.4.1 Word Embeddings

Dense Representations Project word vectors $v(t)$ into a low dimensional space R^k with $k << |V|$ of continuous space word representations (a.k.a. **embeddigns**)

$$Embed : R^{|V|} \to R^k$$

$$Embed(v(t)) = e(t)$$

Desired properties: remaining dimensions

Collobert Build embeddings and estimate whether the word is in the proper context using a neural network. Positive examples from text, and negative examples made replacing center word with random one. The loss for the training is

$$Loss(\Theta) = \sum_{x \in X} \sum_{w \in W} \max(0, 1 - f_\Theta(x) + f_\Theta(x^{(w)}))$$

with $x^{(w)}$ obtained by replacing the central word of x with a random word.

Word2Vec Framework for learning words, much faster to train. Idea:

Collect a large corpus of text

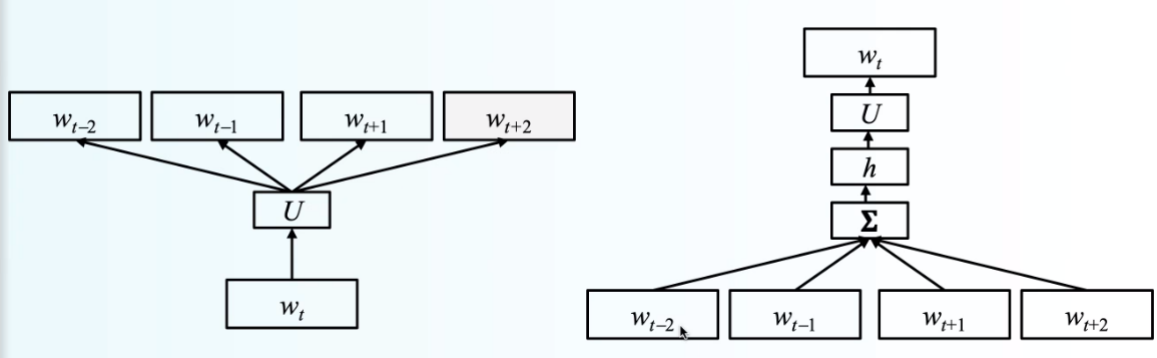
Every word in a fixed vocabulary is represented by a vector

Go through each position t in the text, which has a center word c and context ("outside") words o

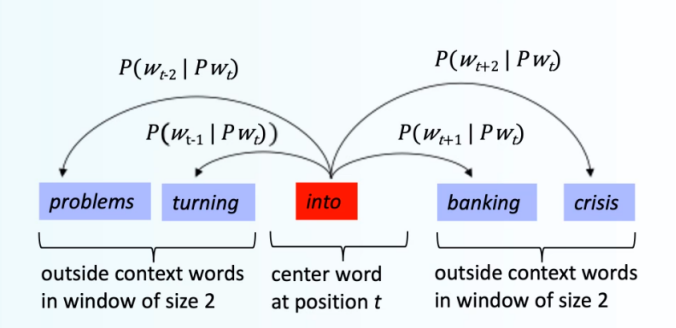
Use the similarity of the word vectors for c and o to calculate the probability of o given c

Keep adjusting word vectors to maximize this probability

Two kinds:



Skip-gram: predict context words within window of size m given the center word w_t



CBoW: predict center word w_t given context words within window of size m

Embeddings are a by-product of the word prediction task. Even though it's a prediction task, the network can be trained on any text (no need for human-labeled data!).
Usual context size is 5 words before and after. Features can be multi-word expressions. Longer windows can capture more semantics and less syntax.
A typical size for h is 200-300.

Skip-Gram

Inputs are one-hot representation of the word

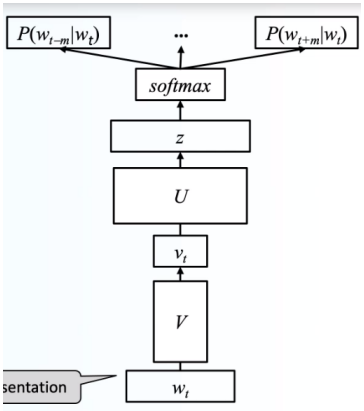
$w \in R^{|Vocabulary|}$ are high-dimensional

$v \in R^d$ is low dimensional: size of the embedding space d

$V \in R^{|Voc| \times d}$ input word matrix

row t of V is the input vector, representation for **center** word w_t

$U \in R^{d \times |Voc|}$ output matrix



column o of U is the output vector, representation for **context** word w_o

$v_t = w_t V$ embedding of word w_t
$z = v_t U$ z_i is the similarity of w_t with w_i
Softmax converts z to a probability distribution p_i

$$p_i = \frac{e^{z_i}}{\sum_{j \in V} e^{z_j}}$$

Procedure

Lookup the embedded word vector for the center word $v_c = V[c] \in R^n$

Generate score vector $z = v_c U$

Turn the score vector into probabilities $\hat{y} = \text{softmax}(z)$

$\hat{y}_{c-m}, \ldots$ Those are the estimates
So the training iterates through the whole corpus predicting sorrounding words given the center word.

Objective Function For each position $t = 1, \ldots, T$ predict context words within a windows of fixed size m given each center word w_t

$$\text{Likelihood} = L(\Theta) = \prod_{t=1}^{T} \prod_{-m \le j \le m,\ j \neq 0} P(w_{t+j} \mid w_t, \Theta)$$

While the objective function $J(\Theta)$ is the average negative log likelihood

$$J(\Theta) = -\frac{1}{T} \log L(\Theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \le j \le m,\ j \neq 0} \log(P(w_{t+j} \mid w_t, \Theta) =$$

To compute $P(o \mid c, \Theta)$ we use two vectors per word w: v_w when w center word and u_w when w context word. For a center word c and a context word o:

$$P(o \mid c) = \frac{e^{u_o v_c}}{\sum_{w \in V} e^{u_w v_c}}$$

With the dot product $u \cdot v = \sum_i u_i v_i$: larger product $\Rightarrow$ larger probability. Normalize over entire vocabulary to give probability distribution.
$J(\Theta)$ is a function of all windows in the corpus, potentially billions: too expensive to compute. The solution is the stochastic gradient descent, sampling windows randomly and update after each one.

Softmax

Soft because still assign some probability to smaller x_i

Max because amplifies the probability of largest x_i

Can we really capture the concept represented by a word? Philosophical debate.

Negative Sampling $\log \sum_{j \in F} e^{u_j}$ has lots of terms, costly to compute. The solution is to compute it only on a small sample of negative samples, i.e. $\log \sum_{j \in E} e^{u_j}$ where words E are a few (e.g. 5)

CBoW Continuous Bag of Words
Mirror of the skip-gram, where context words are used to predict target words.
h is computed from the average of the embeddings of the input context, z_i is the similarity of h with the words embedding of w_i from U.

Which Embeddings V and U both define embeddigns, which to use? Usually just V. Sometimes average pairs of vectors from V and U into a single one or append one embedding vector after the other, doubling the length.

GloVe Global Vectors for Word Representation. Insight: the ratio of conditional probabilities may capture meaning.

$$J = \sum_{i,j=1}^{V} f(X_{ij}) \dots$$

fastText Similar to CBoW, word embeddings averaged to obtain good sentence representation. Pretrained models.

Co-Occurrence Counts

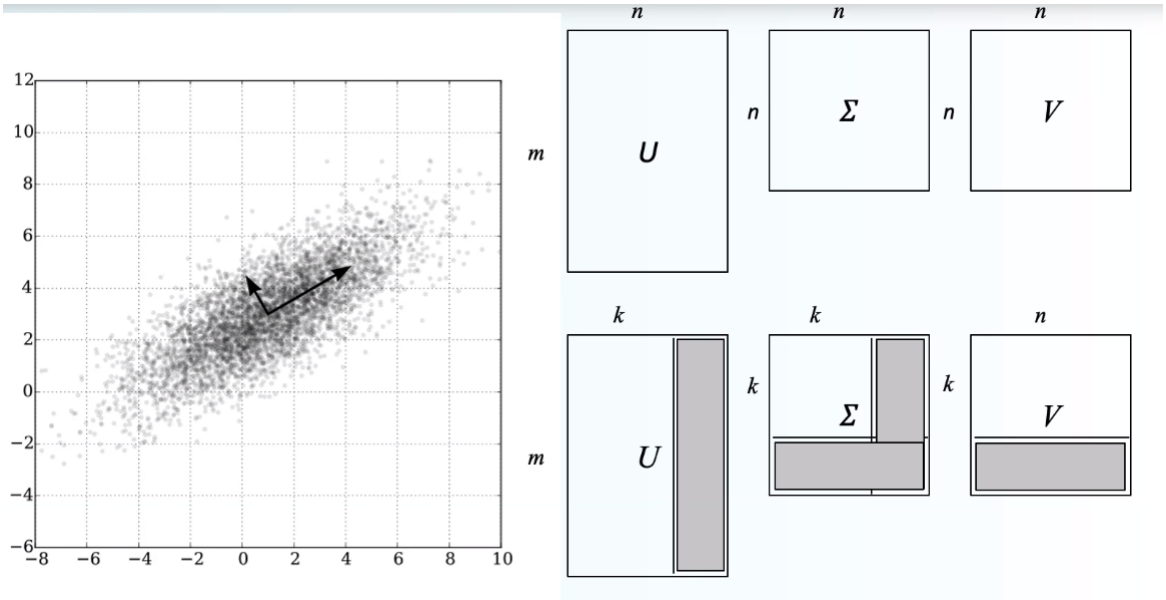
$$P(w_t, w_{t-i}, \dots, w_{t-1}) = \frac{P(w_t, w_{t-i}, \dots, w_{t-1})}{P(w_{t-i}, \dots, w_{t-1})}$$

It's a big matrix, of $|V| \times |V| \simeq 100\text{k} \times 100\text{k} \Rightarrow$ dimensionality reduction: principal component analysis, Hellinger PCA, SVD... trying to reduce to size to 100k×50, 100k×100 or something similar, assigning each word a vector of 50, 100 or similar features.

Weighting Weight the counts using corpus-level statistics to reflect co-occurrence significance: **Pointwise Mutual Information**

$$PMI(w_t, w_{t-i}, \dots, w_{t-1}) = \frac{P(w_t, w_{t-i}, \dots, w_{t-1})}{\log P(w_t) P(w_{t-i}, \dots, w_{t-1})} = \log \frac{\#(w_t, w_{t-i}, \dots, w_{t-1}) \cdot |V|}{\#(w_{t-i}, \dots, w_{t-1}) \#(w_t)}$$

Skip-gram model implicitly factorizes a shifted PMI matrix.
Idea of Singular Value Decomposition:



Which One? No clear winner. Parameters play a relevant role in the outcome of each method. Both SVD and SGNS performed well on most tasks, never underperforming significantly.
SGNS is suggested to be a good baseline: faster to compute and performs well.

Parallel word2vec How to synchronize access to V and U, given multicore CPU to run SGD in parallel? No synchronization is good, because computation is stochastic hence it is approximate anyhow. Parameters are huge: low likelihood of concurrent access to the same memory cell. The effect is a very fast training.

Computing embeddings The training cost of word2vec is linear in the size of the input. The training algorithm works well in parallel, given sparsity of words in contexts and use of negative sampling. It can be halted and restarted at anytime.

Gensim Cython

Fang Uses PyTorch

0.4.2 Evaluation

Polysemy Word vector is a linear combination of its word senses.

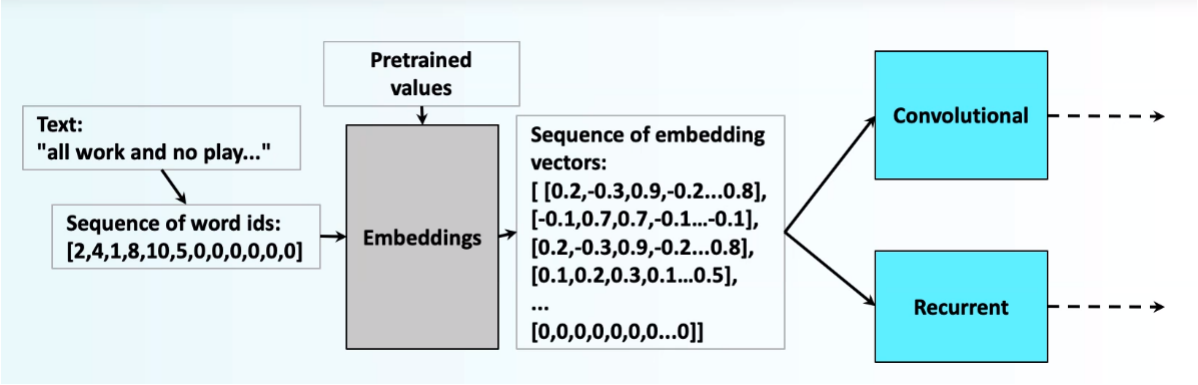
$$v_{pike} = \alpha_1 v_{pike_1} + \alpha_2 v_{pike_3} + \alpha_3 v_3$$

with $\alpha_i = \frac{f_i}{f_1+f_2+f_3}$ for the frequencies f_i.
It's intrinsic evaluation.

Extrinsic Vector Evaluation The proof of the pudding is in the eating. Test on a task, e.g. NER (Named Entity Recognition)

Embeddings in Neural Networks

An embedding layer is often used as fist layer in a neural network for processing text.
It consists of a matrix W of size $|V| \times d$ where d is the size of the embedding space. W maps words to dense representations.
It is initialized either with random weights...



Limits of Word Embeddings

Polysemous words

Limited to words (neither multi words nor phrases)

Represent similarity: antinomies often appear similar.
Not good for sentiment analysis or polysemous words. Example:

The movie was **exciting**

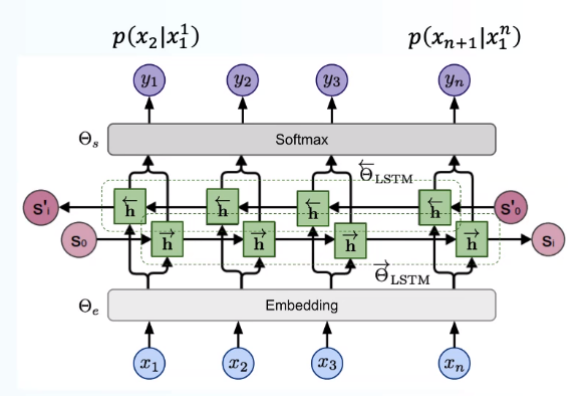
The movie was **boring**

Word Senses and Ambiguity

Sentiment Specific

Context Aware Word Embeddings

ELMo Embeddings from Language Model



Given a sequence of n tokens $(x_1, \ldots, x_n)$

OpenAI GPT-2

BERT Semi-supervised training on large amounts of text, or supervised training on a specific task with a labeled dataset.

0.5 Text Classification

For example: positive/negative review identification, author identification, spam identification, subject identification...

Definition The classifier $f : D \to C$ with

$d \in D$ input document

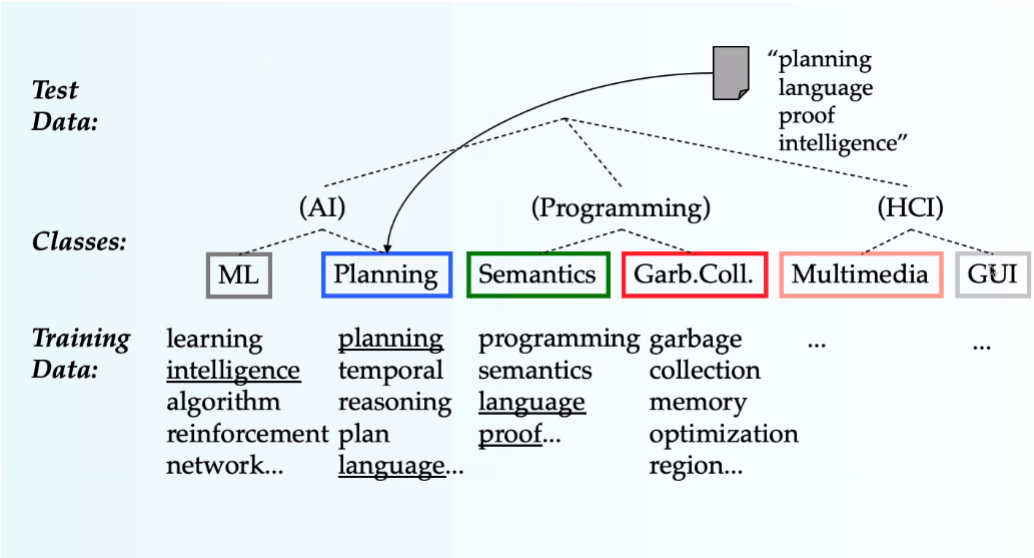
$C = \{c_1, \ldots, c_K\}$ set of classes

$c \in C$ predicted class as output

The learner has

Input: a set of N hand-labeled documents $T = \{(d_1, c_1), \ldots, (d_N, c_N)\}$

Output: a learned classifier $f : D \to C$



Hand-Coded Rules Often very high accuracy, but building and maintaining these rules is expensive. For example: assign category if a document contains a given boolean combination of words (e.g. a blacklist of words for spam classification).

Supervised Machine Learning Input

A document $d \in D$

A fixed set of classes $C = \{c_1, \dots, c_K\}$

A training set of N hand-labeled documents $T = \{(d_1, c_1), \dots, (d_N, c_N)\}$

As output

A learned classifier $\gamma : D \to C$

0.5.1 Naive Bayes

A method based on the Bayes rule, relying on simple document representation (bag of words)

Bag of words representation From a text, we count the frequency of each word.

Bayes Rule Allows to swap the conditioning, useful because sometimes is easier estimating one kind of dependence than the other.

$$P(B \mid A) = \frac{P(A \mid B)P(B)}{P(A)}$$

Applied to documents $d \in D$ and classes $c \in C$

$$P(c, d) = P(c \mid d)P(d) = P(d \mid c)P(c)$$

$$P(c \mid d) = \frac{P(d \mid c)P(c)}{P(d)}$$

Text classification problem Using a supervised learning method, we want to learn a classifier $\gamma : X \to C$. The supervised learning method is denoted with $\Gamma(T) = \gamma$: it takes the training set T as input and returns the learned classifier γ that can be applied to the test set.

Naive Bayes Classifiers

We represent an instance D based on some attributes $D = (x_1, \dots, x_n)$
Task: classify a new instance D based on a tuple of attribute values into one of the calsses $c_j \in C$

$$C_{MAP} = \arg\max_{c_j \in C} P(x_1, \dots, x_n \mid c_j)P(c_j)$$

Naive Bayes Assumption

$P(c_j)$ can be estimated from the frequency of classes in the training examples

$P(x_1, \dots, x_n \mid c_j)$ has $O(|X|^n \cdot |C|)$ parameters and could only be estimated if a very very large number of training examples was available.

The **Naive Bayes Conditional Independence Assumption** is to assume that the probability of observating the conjunction of attributes is equal to the product of the individual probabilities $P(x_i \mid c_j)$. This means that features are independent of each other given the class

$$P(x_1, \dots, x_n \mid c_j) = P(x_1 \mid c_j) \cdot \dots \cdot P(x_n \mid c_j)$$

Multinomial Naive Bayes Text Classification

$$C_B = \arg\max_{c_j \in C} P(c_j) \prod_i P(x_i \mid x_j)$$

Still too many possibilities. Assume the classification is independent of the position of the words, and use the same parameters for each position. The result is a **bag of words model** (over tokens, not types).

Learning the Model Maximum likelihood estimate: simply use the frequencies in the data

$$\hat{P}(c_j) = \frac{\text{doccount}(C = c_j)}{\text{doccount}(T)}$$

$$\hat{P}(x_i \mid x_j) = \frac{\text{count}(X_i = x_i, C = c_j)}{\text{count}(C = c_j)}$$

Zero probabilities cannot be conditioned away, no matter the other evidence!

$$l = \arg\max_c \hat{P}(c) \prod_i \hat{P}(x_i \mid c)$$

Smoothing to Avoid Overfitting For example adding 1 to the counts so that it would never be zero (**Laplace**)

$$\hat{P}(x_i \mid c_j) = \frac{\text{count}(X_i = x_i, C = c_j) + 1}{\text{count}(C = c_j) + k}$$

with $k = \#$ values of X_i
Other ways: for example Bayes Unigram Prior

$$\hat{P}(x_{ik} \mid c_j) = \frac{\text{count}(X_i = x_{ik}, C = c_j) + mp_{ij}}{\text{count}(C = c_j) + m}$$

With mp_{ik} overall fraction in data where $X_i = x_{ik}$ and m extent of "smoothing".

Classifying Return the most likely category for a given document

$$C_{NB} = \arg\max_{c_j \in C} P(c_j) \prod_i P(w_i \mid c_j)$$

Preventing Underflow Log space
Multiplying lots of prob can result in floating point underflow. It's better to perform computations by summing logs of probabilities since $\log(xy) = \log x + \log y$
Class with highest final unnormalized log probability is still the most probable

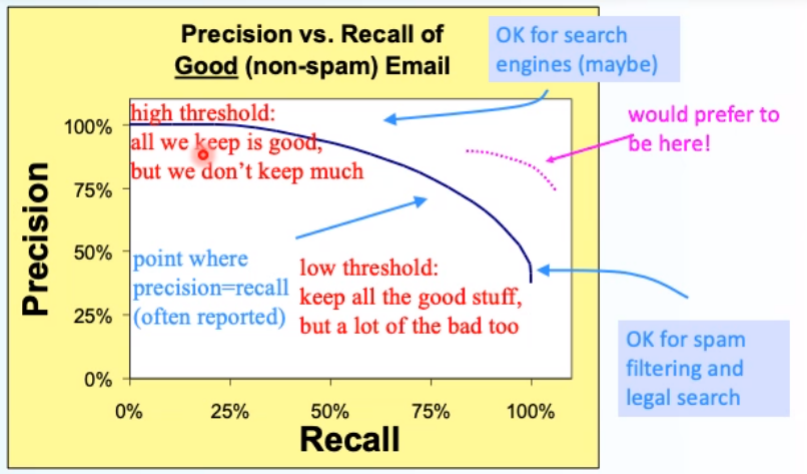
$$C_{NB} = \arg\max_{c_j \in C} \log P(c_j) + \sum_i \log P(x_i \mid c_j)$$

Note: the model is now just a max sum of weights.

Generate We can use naive bayes models to generate text, by using the probabilities of the words.

Naive Bayes and Language Modeling Not the same thing, in naive we want to generalize and can use any sort of features. But if we only use word features and use all the words in a text, then Naive Bayes bears similarity to language modeling.

Evaluating Categorization Must be done on data independent of the training data. Accuracy is $\frac{c}{n}$ where n is the total number of test instancens and c in the number of instances correctly classiefied.



Contingency table

	Correct	Incorrect
Selected	True Positive	False Positive
Not selected	False Negative	True Negative

Micro vs Macro Averaging Macro: performance for each class.
Micro: decision for all classes, compute contingecy table and evaluate

Multiclass Classification More than 2 class, a binary classifier to dinstiguish belongin to a class and *not belonging* to it.

Training Size The more the better, usually.

Violation of Naive Bayes Assumptions Conditional and positional independence.
Naive Bayss is not so naive. Among state of the art algorithms, being robust to irrelevant features (cancel each others).
A good baseline for text classification, but not the best.
Optimal if the independence assumptions hold. Also is very fast, low storage requirements and **online learning algorithm** (incremental training, on new examples).

Example: SpamAssassin Naive Bayes widely used in spam filtering.

0.6 •

Regular Expressions Formal language for specifying text strings. Letters inside square brackets, or specified ranges, like [] and [A-Z], or negations.

Tokenization To do before analysis, for information retrieval and extraction, and spell-checking. Three tasks:

1. Segmenting/tokenizing words in running text
2. Normalizing word formats
3. Segmenting sentences in running text

What's a Word? Not easy. Babbling, in spoken language, or "are *cat* and *cats* the same word?".
Terminology:

> **Lemma**: a set of lexical forms having the same stem, major part of speect, and rough word sense. What you would find in a dictionary.
> *Cat* and *cats* = same lemma
>
> **Wordform**: full inflected surface form.
> *Cat* and *cats* = different wordform
>
> **Type**/Form: element of the vocabulary
>
> **Token**

How many words? N tokens and V vocabulary, set of types (of size $|V|$)
$|V| > O(N^{\frac{1}{2}})$
Google N-grams has $N = 1$ trillion and $|V| = 13$ million.

Stanza Tokenizer Toolkit, ternary classifier to distinguish between: normal character, end of token and end of sentence.

Clitics Some languages have composite words: lascia-mi, lascia-me-lo... Splitting clitics is important for parsing, since clitics incorporate relevant syntactic components (e.g. pronouns corresponding to an object of the verb).
Train 4-class tokenizer: normal character, end of token, end of sentence, start of clitic.

0.7 Classification

1. Define classes/categories
2. Label text
3. Extract features
4. Select classifier: Naive Bayes Classifiers, Decision Trees, SVMs, Neural Networks...
5. Train it
6. Use it to classify new examples

The data is easier to handle if it's linearly separable. Naive Bayes is slightly more general than DTs.

Naive Bayes Simple model, can scale easily to millions of training examples, efficient and fast in training and classification.
A major limitation is the independence assumption It's an inappropriate assumption if there are strong conditional dependencies between the variables.

Decision Trees Capable to generate understandable rules, and perform classification without requiring much computation. Can handle continuous and categorical variables and provide clear indication of the important features.
But it's prone to errors in classification problems with many classes and small numbers of training examples. Also can be computationally expensive to train: need to compare all possible splits, and pruning can be expensive.

Linear vs non-linear algorithms We find out if data is linearly or non linearly separable only empirically.
Linear algorithms when data is linearly separable Non linear when data is not, more accurate but more parameters (e.g. Kernel methods)

Perceptron algorithm

0.7.1 Linear Binary Classification

Data: $\{(x_i, y_i)\}$ for $i = 1, \ldots, n$

$x \in R^d$

$y \in \{-1, +1\}$

Question: find a linear decision boundary

$$wx + b$$

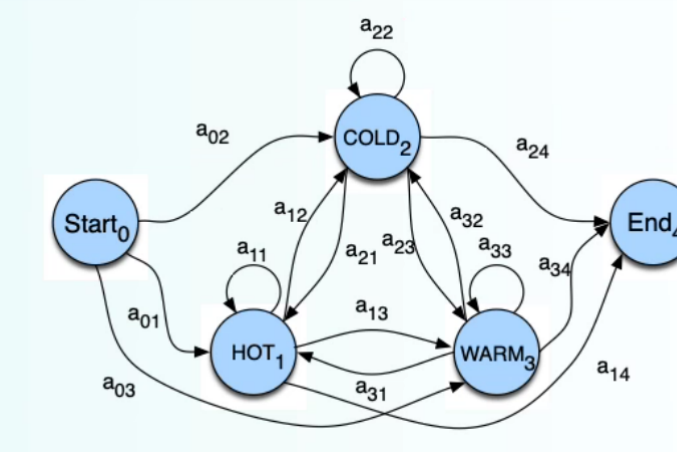
an hyperplane such that the classification rule associated with it has minimal probability of error.
Classification rule:

$$y = sign(wx + b)$$

Perceptron Solves if linearly separable. Basic idea: go through all existing data patterns whose label is known, if correct continue. If not, add to the weights a quantity proportional to the product of the input pattern with y (-1 or $+1$)

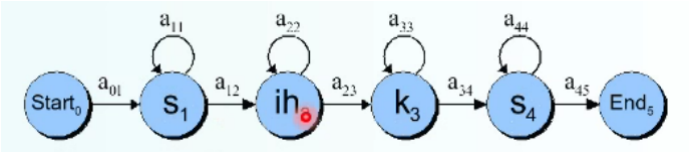
0.7.2 Hidden Markov Models

Markov Chain Stochastic model describing a sequence of possible events in which the probability...



Hidden Markov Model For the chains, the output symbols are the same as the states. In named-entity of part-of-speech tagging (and speech recognition) the output symbols are **words** and the hidden states are **something else**: tags

Example: speech Observed outputs are phones (speech sounds) and hidden states are phonemes (units of sound).
Loopbacks because a phone is circa 100ms but phones are captured every 10ms, so each phone repeats 10 times (simplifying greatly).



$Q = q_1 q_2 \ldots q_N$	a set of N **hidden states**
$A = a_{11} a_{12} \ldots a_{n1} \ldots a_{nn}$	a **transition probability matrix** A, each a_{ij} representing the probability of moving fronm state i to state j, s.t. $\sum_{j=1}^{n} a_{ij} \forall i$
$O = o_1 o_2 \ldots o_T$	a sequenxe of T **observations**, each one drawn from a vocavulary $V = v_1, v_2, \ldots, v_V$
$B = b_i(o_t)$	a sequence of **observation likelihoods**, also called **emission probabilities**, each expressing the probabilitu of an observation o_t being generated from a state i
q_0, q_F	a special **start state** and an **end state** that are not associated with observations, together with transition probabilities $a_{01} a_{02} \ldots a_{0n}$ out of the start state and $a_{1F} a_{1F} \ldots a_{nF}$ into the end state.

Markov Assumption

$$P(q_i \mid q_1, \ldots, q_{i-1} = P(q_i \mid q_{i-1})$$

Output-independence assumption

$$P(o_t \mid O_1^{t-1}, q_1^t) P(o_t \mid q_t)$$

Three basic problems

Evaluation: given the observation sequence $O = (o_1, \ldots, o_T)$ and a HMM model $\Phi = (A, B)$, how to efficiently compute $P(O \mid \Phi)$ the probability of the observation sequence given the model?

Decoding

Learning: how do we adjust the model parameters $\Phi = (A, B)$ (transition and emission probabilities) to maximise $P(O \mid \Phi)$?

Computing the likelihood

function FORWARD(*observations* of len T, *state-graph* of len N) **returns** *forward-prob*

create a probability matrix *forward*$[N+2,T]$
for each state s **from** 1 **to** N **do** ; initialization step
  $forward[s,1] \leftarrow a_{0,s} * b_s(o_1)$
for each time step t **from** 2 **to** T **do** ; recursion step
  for each state s **from** 1 **to** N **do**

$$forward[s,t] \leftarrow \sum_{s'=1}^{N} forward[s',t-1] * a_{s',s} * b_s(o_t)$$

$$forward[q_F,\mathrm{T}] \leftarrow \sum_{s=1}^{N} forward[s,T] * a_{s,q_F}$$; termination step

return $forward[q_F, T]$

Decoding Given an observation and a HMM, the task of the decoder is to find the best hidden state sequence.
Given the observation sequence $O = (o_1, \ldots, o_T)$, and a HMM $\Phi = (A, B)$, how to choose a corresponding state sequence $Q = (q_1, \ldots, q_T) \ldots$?
One possibility: for each hidden state sequence Q, compute $P(O \mid Q)$ and pick the highest, but N^T possibilities.
Instead: **Viterbi algorithm**, dynamic programming algorithm that uses similar trellis as the Forward algorithm.

Viterbi Algorithm

Training Baum-Welch algorithm (Expectation Maximization), no details.

Part of Speech Tagging

The parts-of-speech, or lexical categories/word classes/lexical tags/POS, are nouns, verbs, adjectives, prepositions... we'll use the term POS the most. Examples:

N, noun

V, verb

ADJ, adjectives

ADV, adverbs

...

For example, "the koala put the keys on the table" → "DET N V..."
But words often have more than one POS: "back" can be ADJ, ADV, N, V. POS tagging problem is determining the POS tag for a particular instance of a word.
We want, out of all sequences of n tags $t_1, \ldots, t_n$, the single tag sequence such that $P(t_1, \ldots, t_n \mid w_1, \ldots, w_n)$ is the highest

$$\hat{t}_1^n = \arg\max_{t_1^n} P(t_1^n \mid w_1^n)$$

We can compute it using the Bayes rule to transform it into a set of other probabilities that are easier to compute.

$$\hat{t}_1^n = \arg\max_{t_1^n} P(w_1^n \mid t_1^n) P(t_1^n)$$

Excluding the denominator because we're taking the maximum. It's composed by likelihood and prior

Likelihood $P(w_1^n \mid t_1^n) \simeq \prod_{i=1}^{n} P(w_i \mid t_i)$ with the naive bayes assumption

Prior $P(t_1^n) \simeq \prod$ with the markov assumption

Two kinds of probabilities

Tag transition probabilities $P(t_i \mid t_{i-1})$

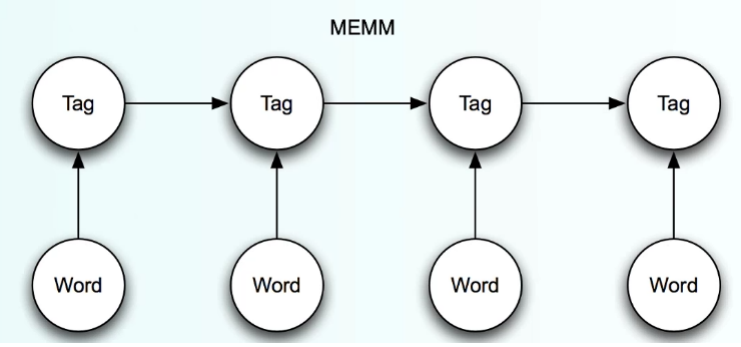
Word likelihood probabilities $P(w_i \mid t_i)$

Sequence Tagging

For example classifying each token independently using information about the surrounding tokens (sliding window) as input features.
For sequence tagging, sequence models work better: HMMs, MEMMs, conditional random fields, convolutional neural networks...

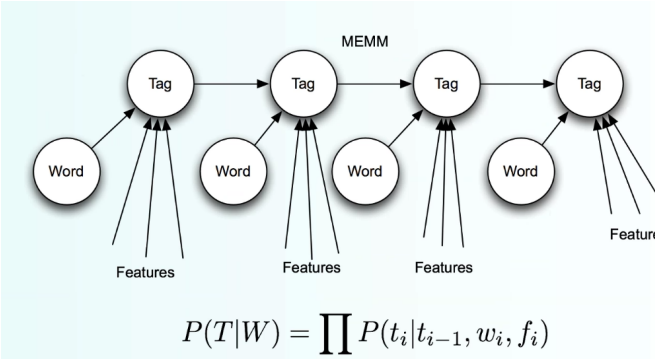
Logistic Regression Model

MEMM HMM works backwards from the tags to the outputs, while MEMM works backwards: from the words to the probabilities of the tags.

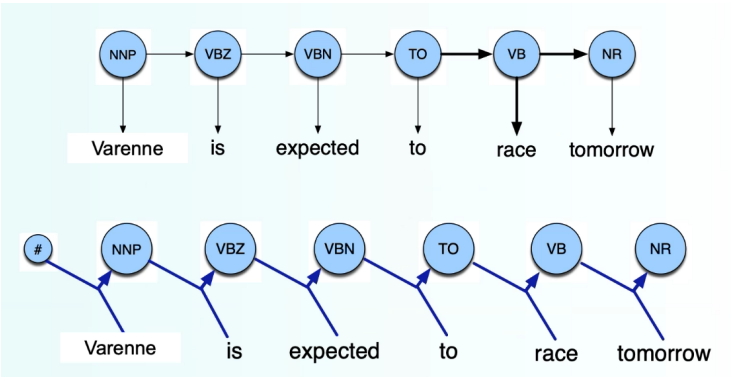


$$P(T \mid W) = \prod P(t_i \mid t_{i-1}, w_i)$$

We can also add **features** and use them in computing the probabilities.



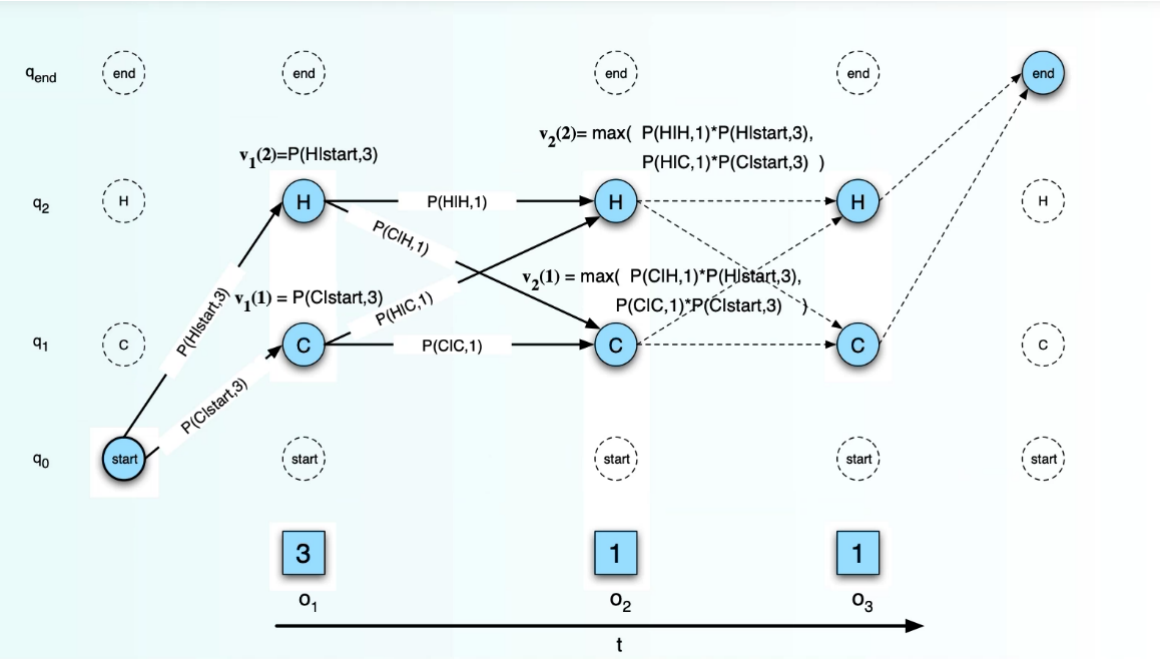
$$P(T|W) = \prod P(t_i|t_{i-1}, w_i, f_i)$$



An example of HMM vs MEMM

Viterbi algorithm can be used to select sequence of tags optimal given the whole sentence. In MEMM, decoding via Viterbi is

$$v_t(j) = \max_{i=1} v_{t-1}(i) P(q_j \mid q_i, o_t) \quad \text{for } 1 \le j \le N, 1 < t \le T$$



Named Entity Tagging

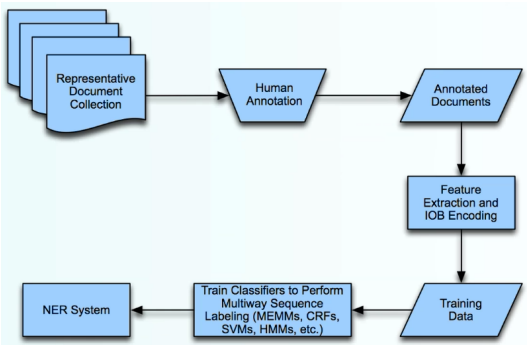
Given a text, find the entities with proper names: person names, city names...the "capitalized things". Typical approach is based on rules, might not be accurate enough. An approach based on ML needs training data, labeling them and using a classifier. Labeling may be easy: annotate each word, but an entity may span more words (name and surname, for example) or be non-continguous, overlapping with other words and perhaps other named entities (examples in biomedical entities).
Typical, named entities systems handle very few categories:

Type	Tag	Sample Categories
People	PER	Individuals, fictional characters, small groups
Organizations	ORG	Companies, agencies, sport teams, parties, religious groups
Location	LOC	Physical extents, mountains, lakes, seas
Geo-political Entities	GPE	Countries, states, provinces, counties
Facility	FAC	Bridges, buildings, airports
Vehicles	VEH	Planes, trains, automobiles

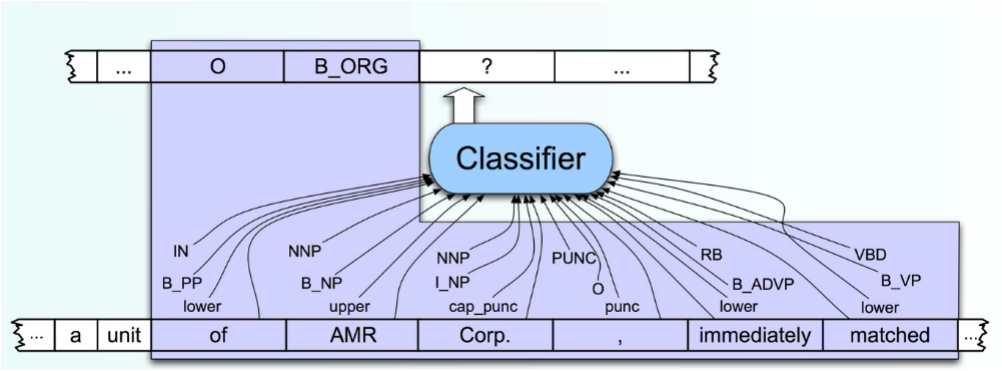
Approaches As with partial parsing and chinking, there are two basic approaches (and hybrids):

Rule-based: patterns to mach things that look like names and environments that classes of names tend to occur in (regular expressions)

ML-based: get annotated data, extract features and train systems to replicate the annotation. Typical approach today



For N classes we have $2N + 1$ tags, with IOB encoding: an I and a B for each tag.



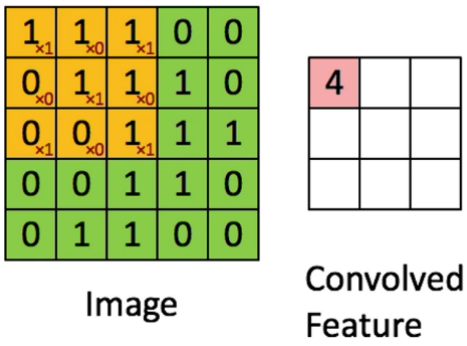
Summary of the approaches.

0.8 Convolutional Neural Networks for NLP

The main CNN idea is to compute vectors for every poissible word subsequence of a certain length, regardless of whether the phrase is grammatially correct, and group them afterwards. Not very linguistically nor cognitively plausible.
Convolution is classically used to extract features from images

$$(f * g)(n) = \sum_{m=-M}^{M} f(n-m)g(m)$$

Much like a "sliding window" across the data.



A convolutional layer in a NN is composed by a set of filetrs: combines a local selection of input values into an output value, sweeping across all input.
During training each filter specializes into recognizing some kiind of relevant combination of features. CNNs work well on stationary features (independent from position). Filters have additional parameters that define:

Behavior at the start/end of documents (**padding**)

Size of the sweep step (**stride**)

Possible presence of holes in the filter window (**dilation**)

Distant Supervision Use the convolutional neural network to further refine the embeddings: word embeddigns from plain text are completely clueless about their sentiment behavior. Collect 10M tweets containing positive emoticons, used as distantly supervised labels to train sentiment-aware embeddings.

Sentiment Specific Word Embeddings The idea is to build sentiment specific word embeddings where we return also the polarity of the word, positive, neutral or negative.

Sentiment Classification from a Single Neuron A char-level LSTM with 4096 units has been trained on 82M reviews from Amazon, only to predict the next character. After training, one of the units had a very high correlation with sentiment, resulting in **state-of-the-art accuracy when used as a classifier**. The model can also be used to generate text: by setting the value of the sentiment unit, one can control the sentiment of the resulting text.

0.8.1 Regularization

We can use **dropout**: creating a masking vector r of Bernoulli random variables with probability p (hyperparameter) of being 1 and delete features during training

$$h = W(r \otimes z) + b$$

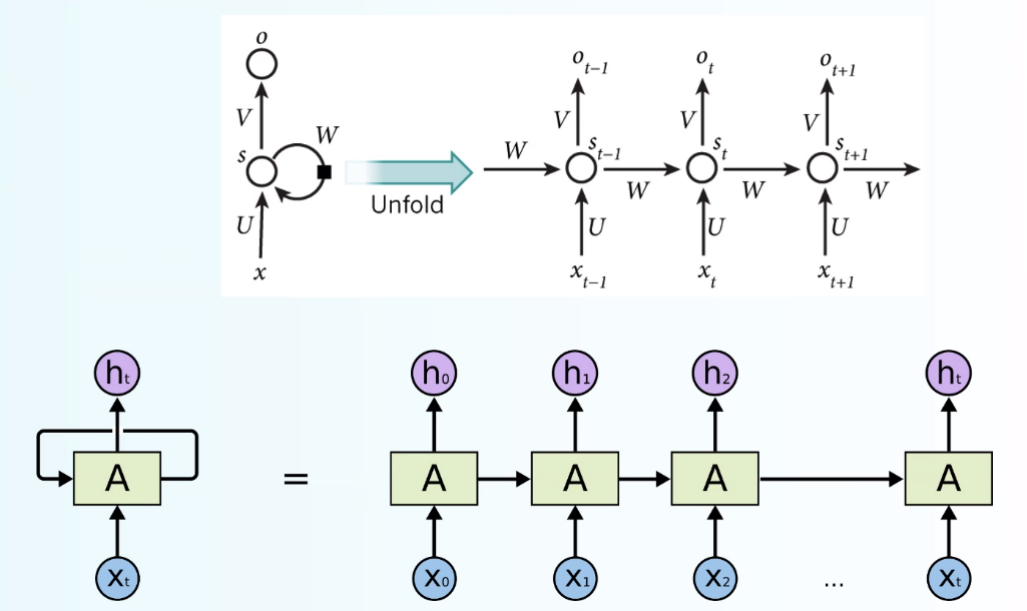
Prevents overfitting. Not used at test time, scaling final vector by probability p. Usually yields an accuracy increase of 2-4%.

0.9 Recurrent Neural Networks

Up until now, whether we grouped words or not each word/group would be handled independently from the others.

Recap A language model assigns to each sentence W a probability $P(W) = P(w_1, \dots, w_n)$. Alternatively we want to compute $P(w_n \mid w_1, \dots, w_{n-1})$.
The model that computes either probability is called **Language Model**, and language modeling is the task of estimating a language model.
The Markov assumption is that w_n depends only on the preceding $n-1$ words. n-gram models have sparsity (out-of-vocabulary words and never-seen-prefixes) and storage problems (counts for every n-gram): bigger n makes the sparsity problem words, typically $n < 5$.
Neural Language Models improves over n-gram language models: no sparsity and no need to store all observed n-grams, but still problems: fixed window is too small, never large enough, and no symmetry in the input where each word is still treated independently and multiplied by completely different weights. We need a **neural architecture that can process any length input**.

Recurrent Because they perform the same process for every element where **the output depends on the previous elements**. RNNs have a "memory" which captures information about what has been calculated so far. In theory, RNNs can make use of information in arbitrarily long sequences.

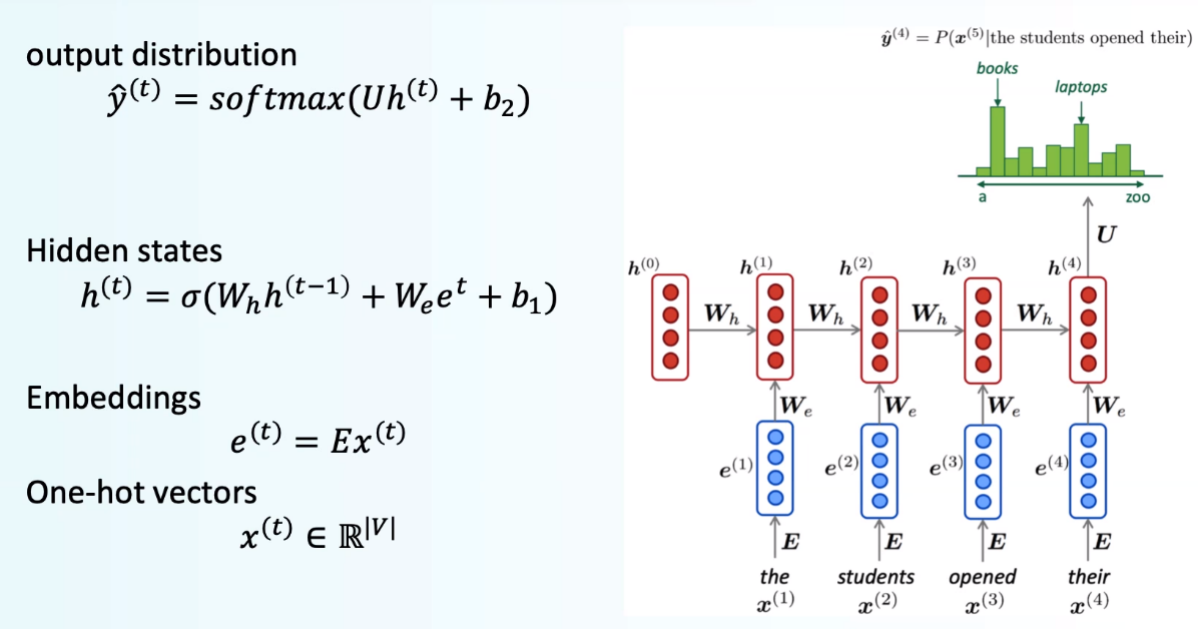


Hidden Units The hidden state s_t represents the memory of the network: it captures information about what happened in all the previous time steps.
The output o_t is computes solely based on the memory at time t.
s_t typically can't capture information from too many time steps ago. Unlike traditional deep networks, which uses different parameters at each layer, a RNN shares the same parameters across all steps (U, V and W above). This reflects the fact that we are performing the same task at each step, just with different inputs, greatly reducing the total number of parameters we need to learn.

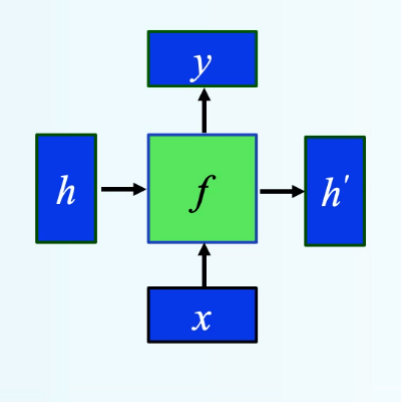
Advantages We can process input of any length and model size doesn't increase with longer input. The computation for step t in theory can use information from many steps back.
Weights are shared across timesteps, so representations are shared too.

Disadvantages Recurrent computation is really slow. In practice, is difficult to access information from many steps back.

Simple RNN Language Model



Vanilla RNN



$$h_t = \sigma\left(Wh_{t-1} + Wx_t + b\right)$$
$$y_t = \sigma\left(Vh_t\right)$$

Notice that y is computed from the current h' only.

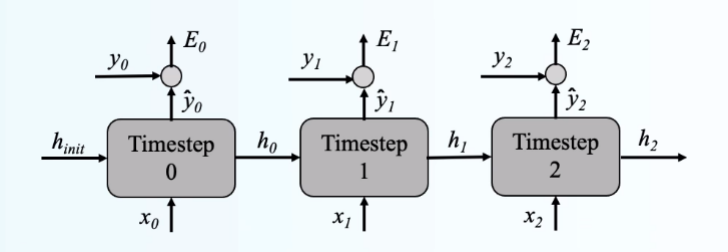
Training Tasks:

for each timestep of the input sequence x we predeict the output y synchronously

for the input sequence x we predict the scalar value of y (e.g. at the end of the sequence)

Main method: **backpropagation**, reliable and controlled convergence, supported by most ML frameworks. Other methods: evolutionary methods, expectation maximization, particle swarm...

Backpropagation through time



Applying the chain rule

$$\frac{\partial h_2}{\partial h_0} = \frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial h_0}$$

For time 2:

$$\frac{\partial E_2}{\partial \Theta} = \sum_{k=0}^{2} \frac{\partial E_2}{\partial \hat{y}_2}\frac{\partial \hat{y}_2}{\partial h_2}\frac{\partial h_2}{\partial h_k}\frac{\partial h_k}{\partial \Theta}$$

Training RNN Language Model Get a big corpus of text and feed into the RNN-LM, computing the output distribution for every step t.
Loss function on step t is cross-exntropy between the predicted probability distribution $\hat{y}$ and the true next word $y^{(t)}$ (one-hot for $x^{(t+1)}$)

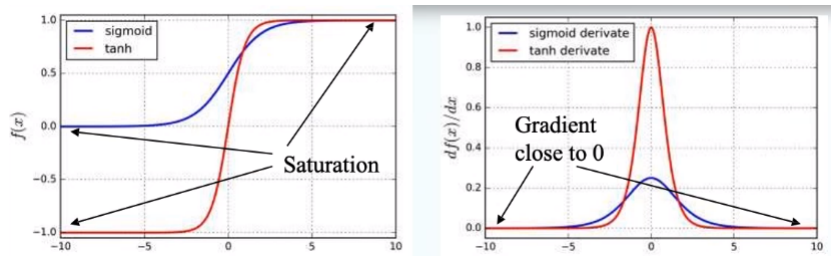
$$J^{(t)}(\Theta) = \text{CrossEntropy}(\hat{y}^{(t)}, y^{(t)}) = -\sum_{w \in V} y_w^{(t)} \log \hat{y}_w^{(t)} = -\log \hat{y}_{x_{t+1}}^{(t)}$$

Average this to get the overall loss for the entire training set

$$J(\Theta) = \frac{1}{T}\sum_{i=1}^{T} J^{(t)}(\Theta) = \frac{1}{T}\sum_{i=1}^{T} -\log \hat{y}_{x_{t+1}}^{(t)}$$

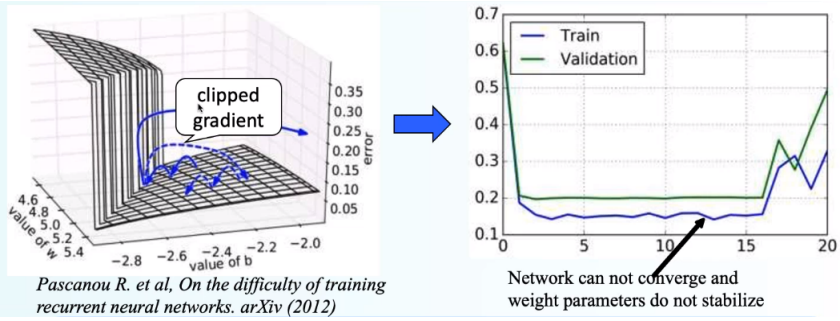
In practice: do this for each sentence and repeat through stochastic gradient descent.

Vanishing Gradients Known problems: the gradients decay exponentially and networks stops learning without updating, making impossible to learn correlations between temporally distant events. A solution is to use ReLU instead of sigmoids.



Smaller weights initialization leads to faster gradient vanishing, and very big initialization make the gradient diverge fast.

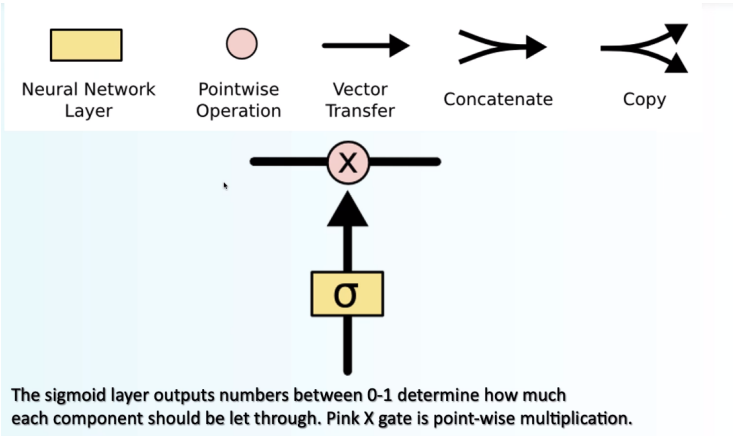
Exploding Gradients The opposite: large increase in the norm, causing NaNs or large fluctuations in cost functions.



Solutions: gradient clipping, reducing learning rates or changing loss function by setting constraints on weights (L1 or L2 norms).

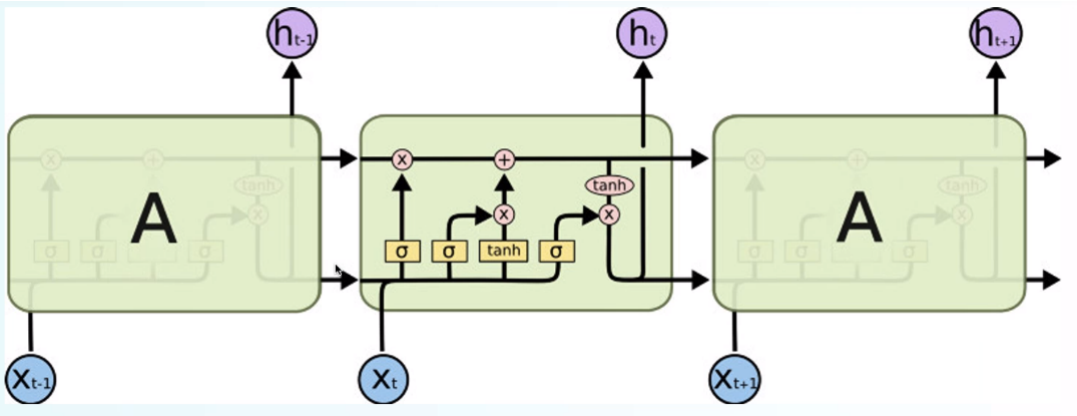
0.9.1 Specializations

Notation



LSTM

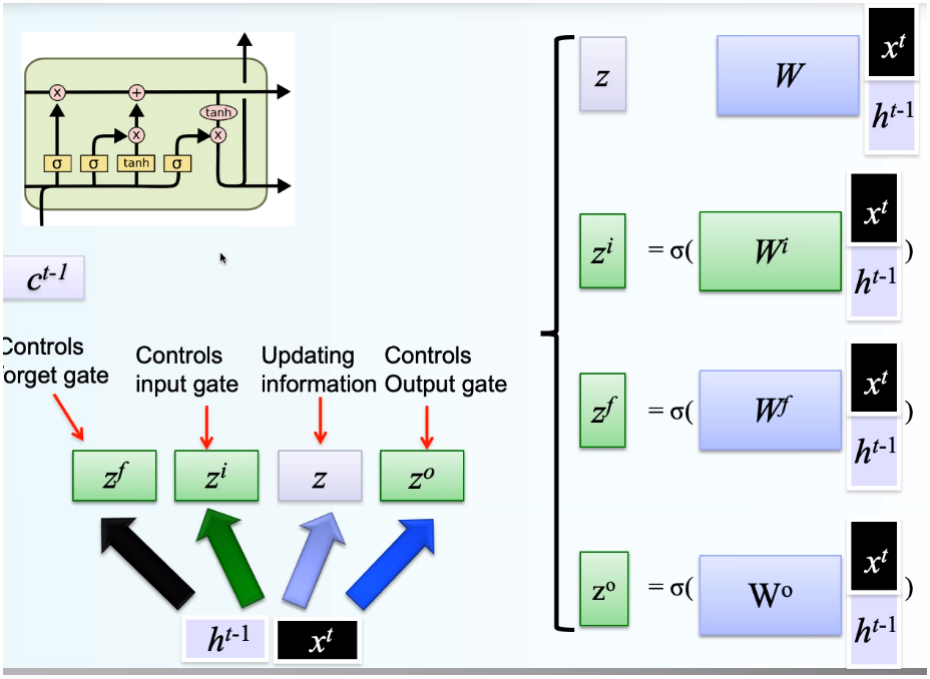
Long Short-Term Memory

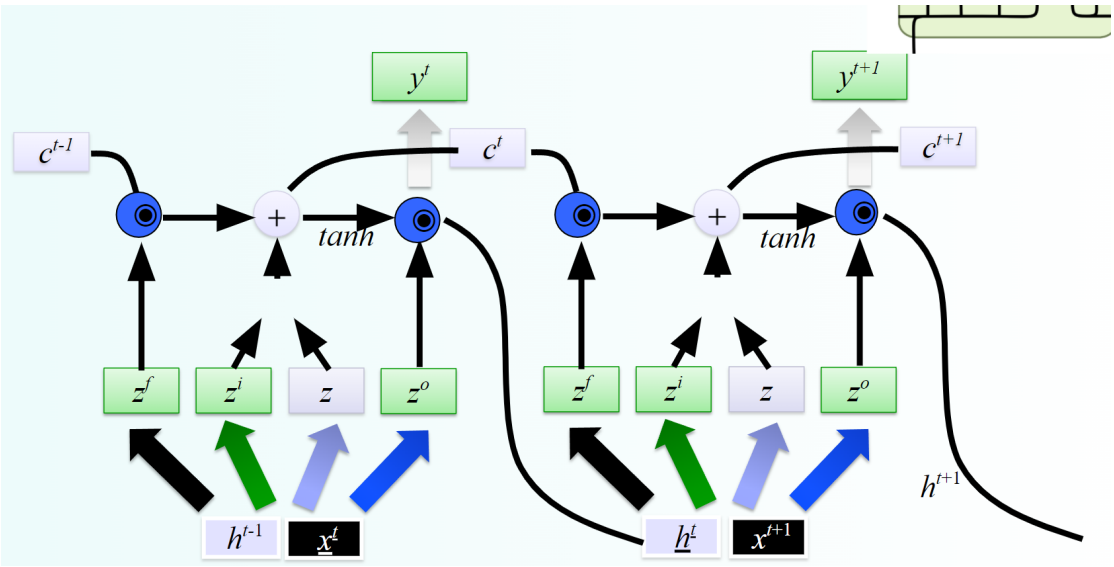


The core idea is this cells state C_t is changed slowly with only minor interactions. Very easy for information to flow unchanged.

$$C_t = f_t C_{t-1} + i_t \hat{C}_t$$

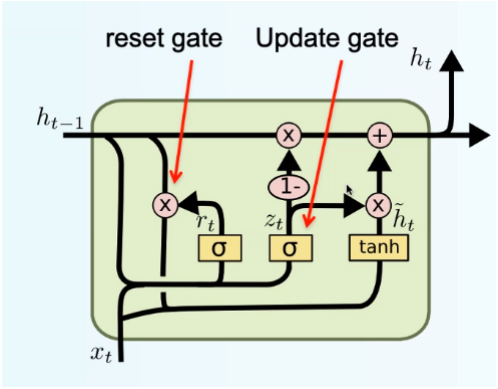
The first sigmoid goes to the **forget gate**, determines how much information goes through.
the second sigmoid is the **input gate** and decides how much input is added in the cell state (so in the next pass).
The **output gate** of the third sigmoid controls what goes in the output.
Why sigmoid or tanh: sigmoid are used as $0/1$ switches, and the vanishing gradients are already handled in the LSTMs.





GRU

Gated Recurrent Units



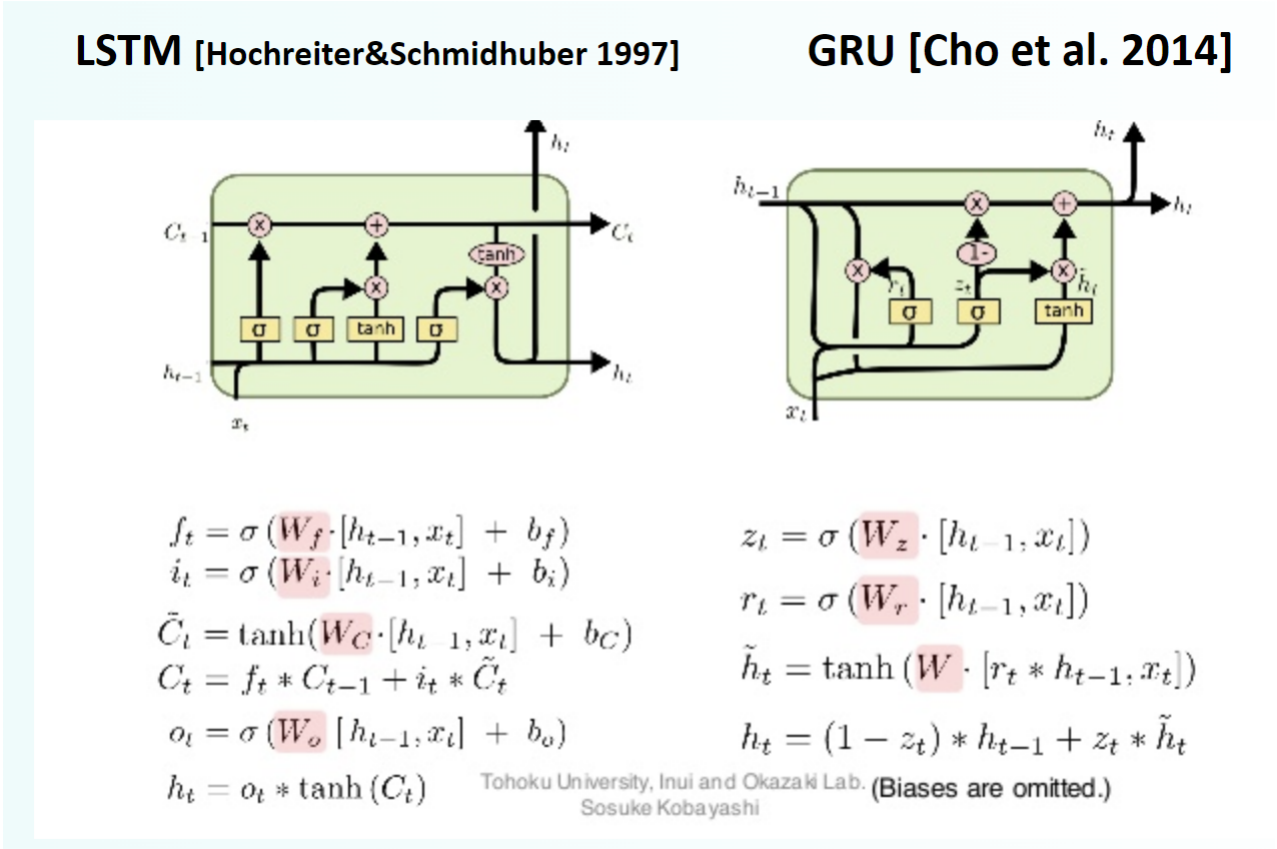
$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$

$$\hat{h}_t = \tanh(W \cdot [r_t \cdot h_{t-1}, x_t])$$

$$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \hat{h}_t$$

Combines forget and input gate into a single update gate, also merging cell state and hidden state. Simpler than LSTM.



0.10 Parsing

Dealing with Text

Bag Of Words representation: enough for classification and information retrieval

N-Grams for language modeling, POS tagging...

Sequences for neural machine translations

But we have nothing that's applicable for information extraction or question answering.

Sentence Structure Recovering the structure is needed to fully understand the language.
Syntax is the way words are arranged together into larger units, and grammar is a formalism used to describe the syntax of a language.
Structural ambiguity: prepositional attachment, coordination scope, verb phrase attachment.

Practical uses of parsing

Relation Extraction: knowledge graph enriched from relation extracted from dependency trees

Semantic Relation

Translation: helps disambiguating sentences

Sentiment Analysis: improved by dependency parsing

Negation: determining the scope of negations

Summarization: detecting relevant parts

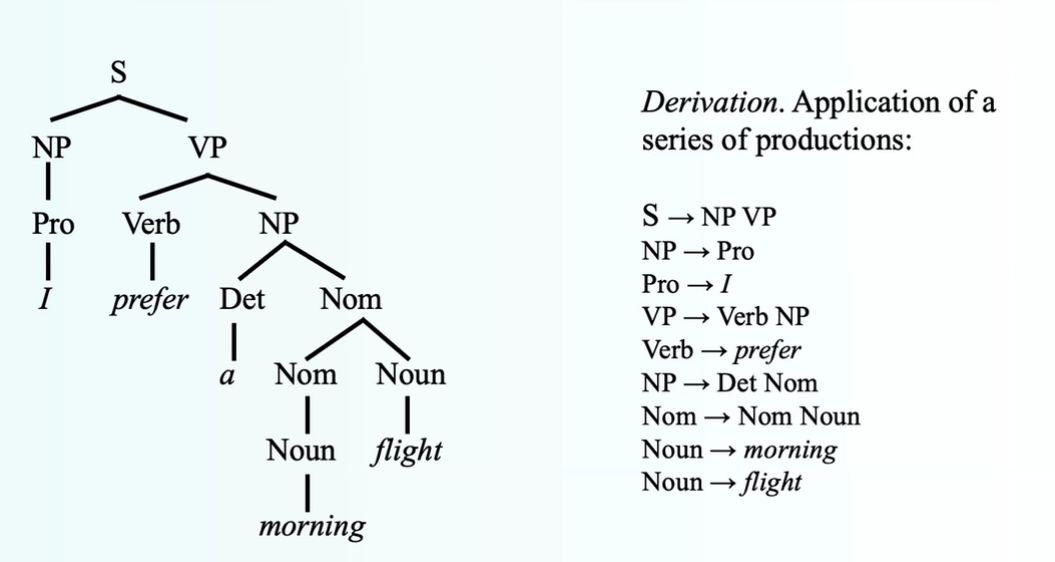
Question Answering

In NLP we've seen information extraction and finding linguistic structure. Can be cast as **learning mapping**: strings to hidden state sequences (POS tagging), strings to strings (translation), strings to trees (**parsing**), strings to relational data structures (information extraction).

0.10.1 Parsing Approaches

Constituency Grammar

AKA phrase structure grammar or **context free grammar**.



Context Free Grammars $G = (N, \Sigma, R, S)$

Set of non-terminal symbols N

Set of terminal symbols Σ disjoint from N

set of rules/productions R in the form $A \to \beta$ with A non-terminal, β string of symbols from the infinte set of strings $(\Sigma \cup N)^*$

A designated start symbol $S \in N$

$$L(G) = \{w \in \Sigma^*, S \to w\}$$

Constituency Parsing Requires phase structure grammar and produces phrase structure parse tree.

Statistical Parsing Three components

GEN is a function from a string to a set of candidate trees

Φ maps a candidate to a feature vector

W is the parameter vector

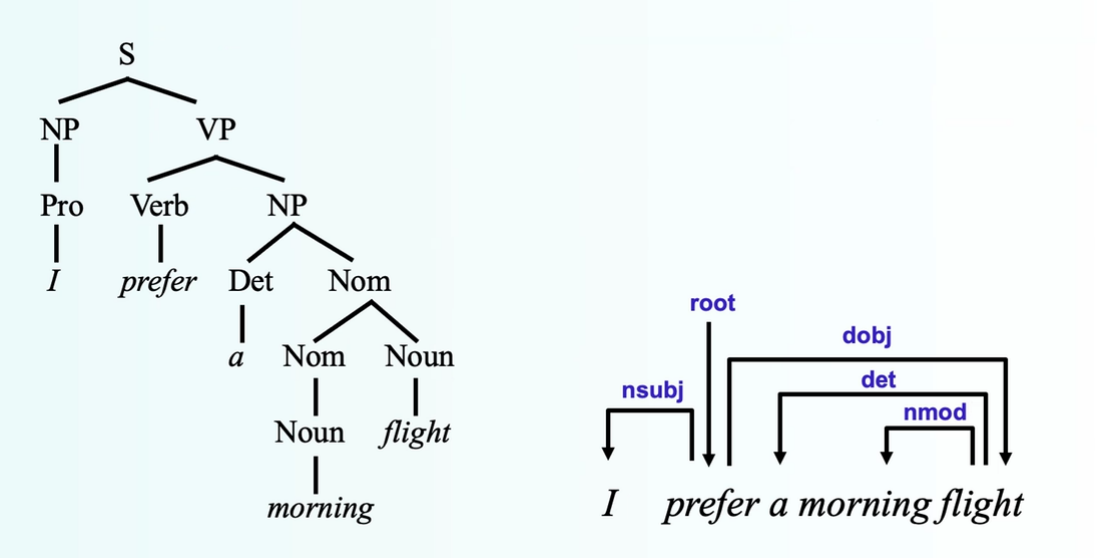
Training by giving a set of sentences X and the set of possible outputs (trees) Y to learn a function $F_W : X \to Y$. Parsing is choosing the highest scoring tree

$$F_W(x) = \arg \max_{y \in GEN(x)} \Phi(y) \cdot W$$

Dependency Grammar

Dependency Structure Shows which words depend on (modify or are arguments of) which other words. The syntactic structure of a sentence is described only in terms of the words in a sentence and a set if directed binary grammatical relations among the words.

Difference Between Constituency Tree and Dependency Trees



Criteria for a syntactic relation between a head H and a dependent D in a construction C

H determines the syntactic category of C (H can replace C)

H determines the semantic category of C (D specifies H)

H is obligatory and D may be optional

H selects D and determines whether D is obligatory

The form of D depends on H

The linear position of D is specified with reference to H

Annotation Constraints A dependency graph $D = (W, A)$ is a directed rooted tree

D is weakly connected: $i, j \in V \Rightarrow i \leftrightarrow^* j$

D is acyclic: $i \rightarrow j \Rightarrow \neg(j \rightarrow^* i)$

D obeys the **single-head constraint**: $i \rightarrow j \Rightarrow \neg(i' \rightarrow j)\ \forall\, i' \neq i$

The single-head constraints causes problems in handling certain linguistic phenomena.

Data-Driven Dependency Parsing

Graph Based: consider the possible dependency graphs and define a score selecting the best scoring one.

Transition Based: define a transition system that leads to a parse tree while analyzing a sentence one word at a time.

Constraint Satisfaction: edges are deleted that don't satisfy hard constraints.

Transition-Based Shift-Reduce Parsing Trained directly on the task of tagging a sentence. Instead, a shift-reduce parser is trained and learns the sequence of parse actions required to build the parse tree. An inductive parser doesn't require grammar, while a traditional parser requires a grammar for generating candidate trees.

Parsing as Classification Inductive dependency parsing, based on Shift/Reduce actions: Learn from annotated corpus which action to perform at each step.

Dependency Graph Let $R = \{r_1, \ldots, r_m\}$ the set of dependency types (the tags we'll put on the links).
A dependency graph for a sequence of words $W = w_1, \ldots, w_n$ is a labeled directed graph $D = (W, A)$ where

W is the set of nodes, i.e. the word tokens in the input sequence

A is a set of labeled arcs (w_i, w_j, r) with $w_i, w_j \in W$ and $r \in R$

$\forall\, w_j \in W$ there is at most one arc $(w_i, w_j, r) \in A$

The parser build such a graph. Its state at each time is a triple (S, B, A) where

S is a stack of partially processed tokens

B is a buffer of remaining input tokens

A is the arc relation for the dependency graph

$(h, d, r) \in A$ represent an arc $h - r \to d$ tagged with relation r.

Arc Standard Transitions

$$\text{Shift} \quad \frac{\langle S, n|B, A\rangle}{\langle S|n, B, A\rangle}$$

$$\text{Left-Arc}_r \quad \frac{\langle S|s, n|B, A\rangle}{\langle S, n|B, A \cup \{(n, s, r\}\rangle}$$

$$\text{Right-Arc}_r \quad \frac{\langle S|s, n|B, A\rangle}{\langle S, s|B, A \cup \{(s, n, r)\}\rangle}$$

Parser Algorithm Is fully deterministic, using a trained model to predict the next action, given a representation of the context current state.

```
Input Sentence: (w1, w2, ... , wn)
  S = <>
  B = <w1, w2, ... , wn>
  A = {}
  while B != <> do
        x = getContext(S, B, A)
        y = selectAction(model, x)
        performAction(y, S, B, A)
```

Oracle The gold tree of each sentence can be used to suggest which actions to perform in order to rebuild such gold tree. There can be more than one possible sequence to produce the same parse tree.
An Oracle is an algorithm that given the gold tree for a sentence, produces a proper sequence of actions that a parser may use to obtain that gold tree from the input sentence.
Simplest Oracle: arc standard Oracle, emulates the parser knowing what the outcome should be, returning the correct action at each step. Works but cannot handle certain situation: e.g. non-projectivity situations.

Projectivity An arc $w_i \to w_k$ is projective $\Leftrightarrow \forall j, i < j < k$ or $i > j > k$ we have $w_i \to^* w_j$, so no arc crosses that arc. A dependency tree is projective if and only if every arc is projective.
Intuitively: arcs can be drawn without intersections.

Arc-Standard Algorithm

Doesn't deal with non-projectivity

Every transition sequence produces a projective dependency tree (soundness)

Every projective tree is produces by some transition sequence (completeness)

Fast deterministic linear algorithm: parsing n words requires $2n$ transition.

Arc Eager Transitions

$$\text{Shift} \quad \frac{\langle S, n|B, A\rangle}{\langle S|n, B, A\rangle}$$

$$\text{Left-Arc}_r \quad \frac{\langle S|s, n|B, A\rangle}{\langle S, n|B, A \cup \{(n, s, r\}\rangle}$$

$$\text{Right-Arc}_r \quad \frac{\langle S|s, n|B, A\rangle}{\langle S|s, s|B, A \cup \{(s, n, r)\}\rangle}$$

(Connects without removing, to delay decision and keep words on the stack that might need further connections to children)

$$\text{Reduce} \quad \frac{\langle S|s, B, A\rangle}{\langle S, B, A\rangle}$$

Non-Projective Transitions

Learning Procedure

Go through each sentence in the treebank and extract the sequence of actions suggested by the oracle.

Dependency Shift-Reduce Parsers

CoNLL-X Shared Task Assign labeled dependency structures for a range of languages by means of a fully automatic dependency parser.

Graph-Based Parsing

For an input sequence x define a graph $G_x = (V_x, A_x)$ where

$V_x = \{0, 1, \ldots, n\}$

The basic idea is to choose the arc with the highest score from each node. But the risk is to end up with a graph and not a tree.
Another solution is the Chu-Liu-Edmonds: if it's not a tree, identify cycle and contract reculaculating arc weights in and out of the cycle. $O(n^2)$ complexity for non-projective trees (much slower than transition-based parsers).

NN Graph-Based Parser Revived graph-based dependency parsing in a neural world, with great results altough slower than neural dependency-based parsers.