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.

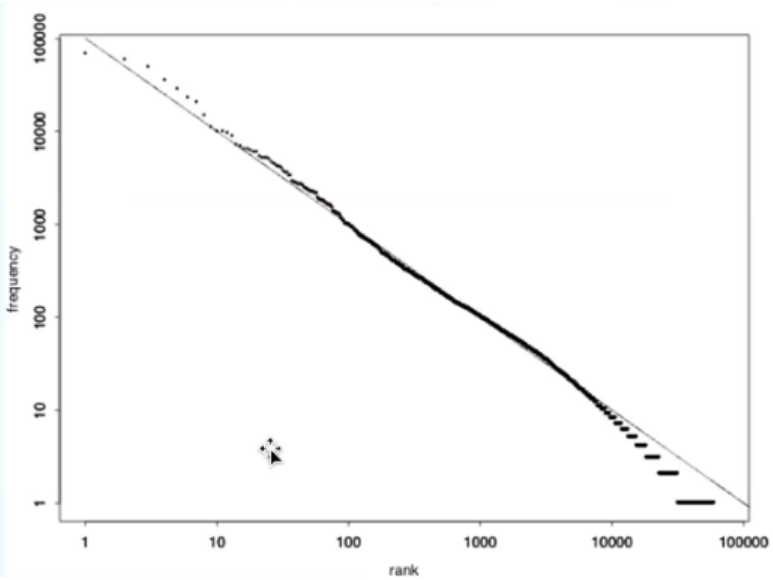
Smoothing Many rare (but not impossible) combinations of word sequences never occur in training, so MLE incorrectly assigns zero to many parameters (sparse data).
If a new combination occurs during testing, it is given a probability of zero and the entire sequence gets a probability of zero (infinite 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

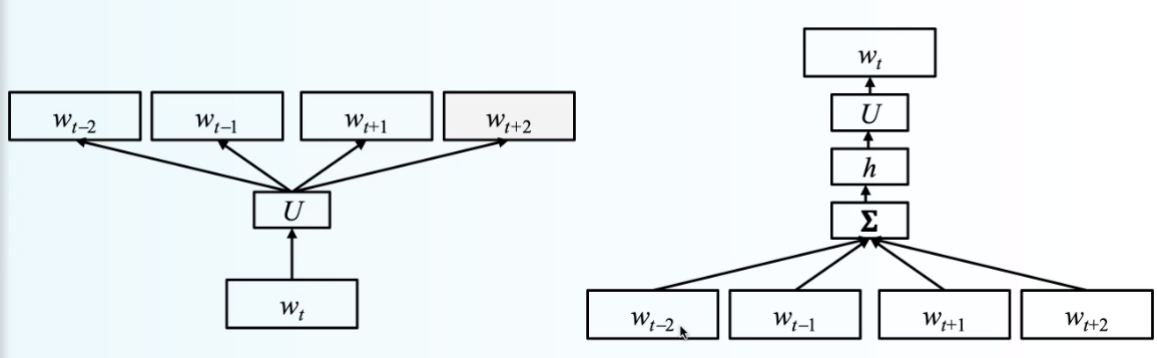
Every word in a fixed vocabulary is represented by a vector

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

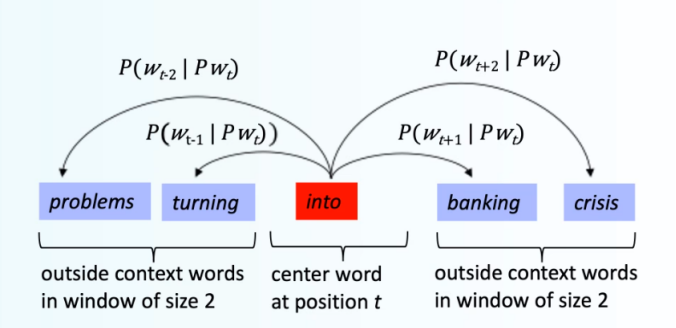
Use the similarity of the word vectors for c and o to calculate the probability of o given c

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

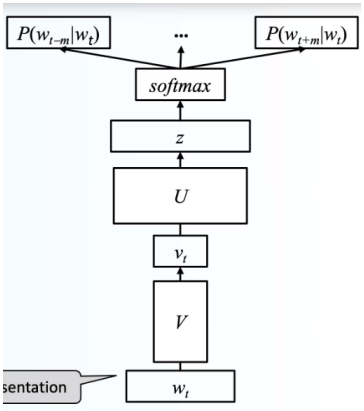
$w \in R^{|Vocabulary|}$ are high-dimensional

$v \in R^d$ 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

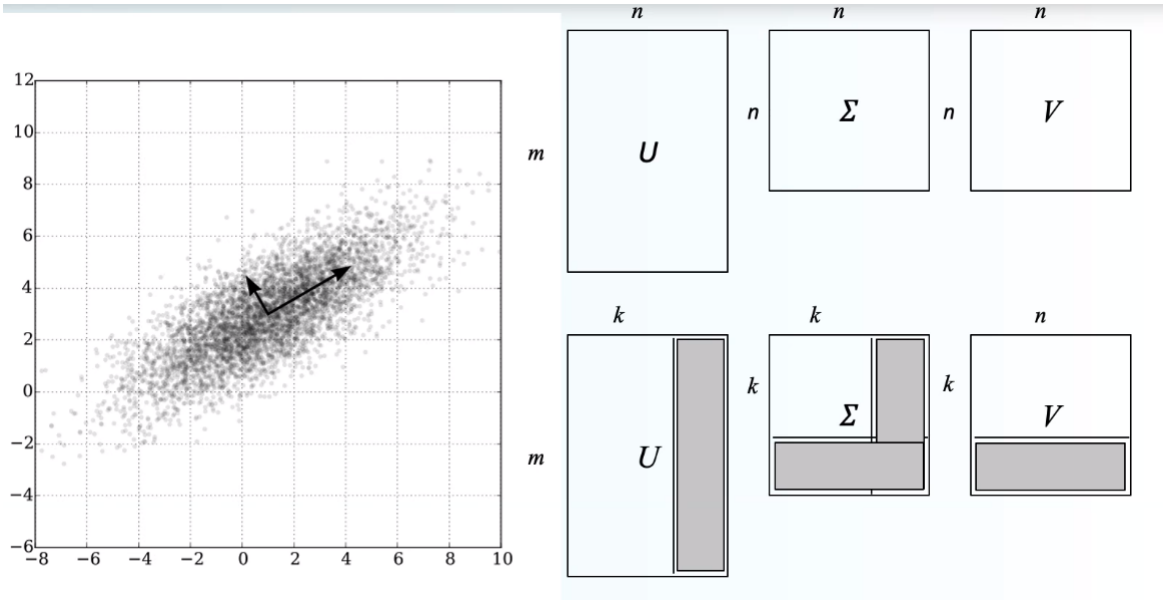
$$P(w_t, w_{t-i}, \dots, w_{t-1}) = \frac{P(w_t, w_{t-i}, \dots, w_{t-1})}{P(w_{t-i}, \dots, w_{t-1})}$$

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.

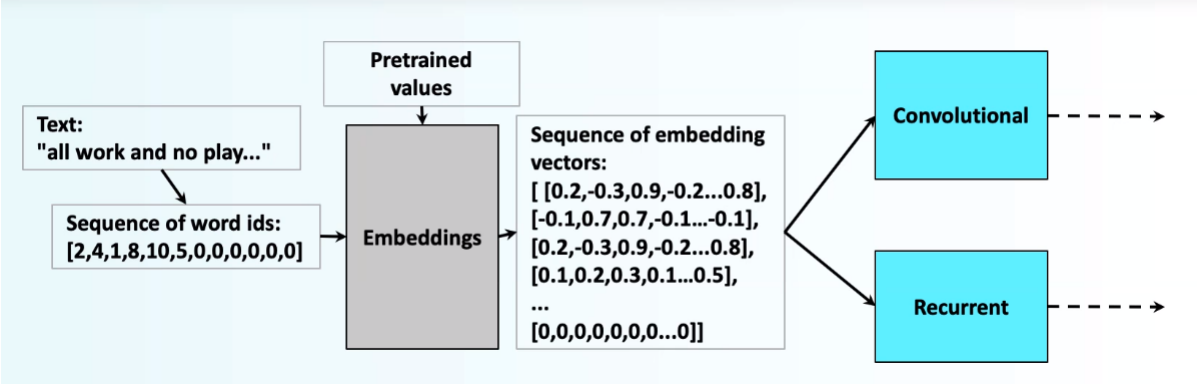
$$v_{pike} = \alpha_1 v_{pike_1} + \alpha_2 v_{pike_3} + \alpha_3 v_3$$

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**

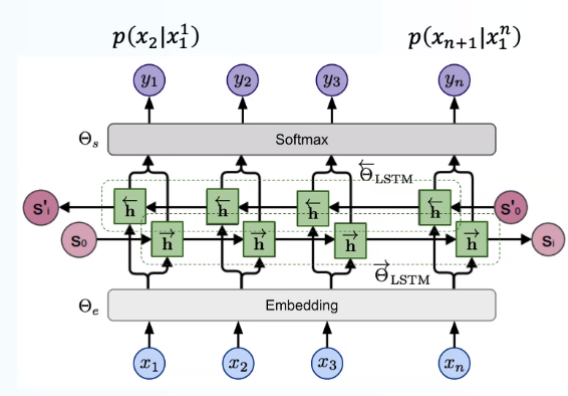
The movie was **boring**

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

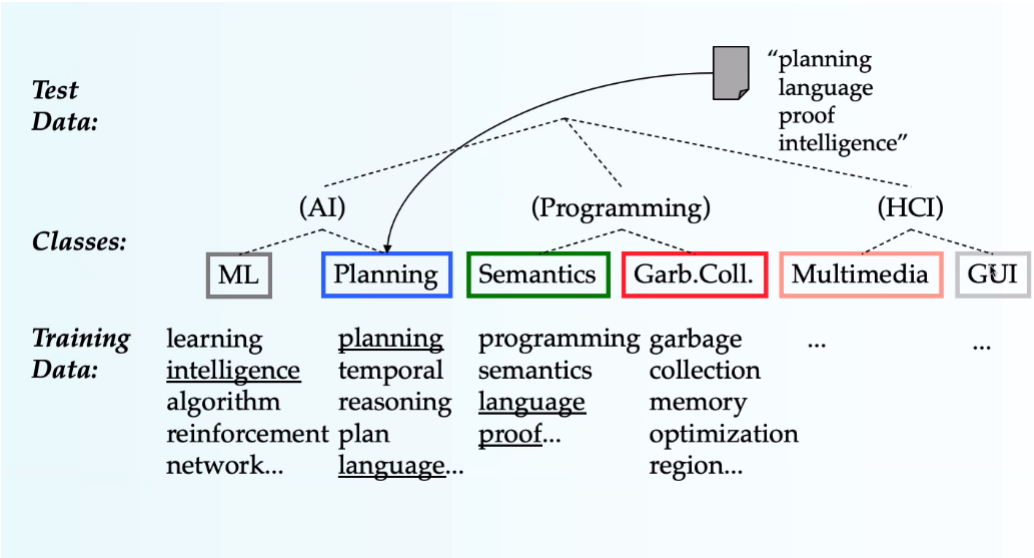
$C = \{c_1, \ldots, c_K\}$ set of classes

$c \in C$ 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

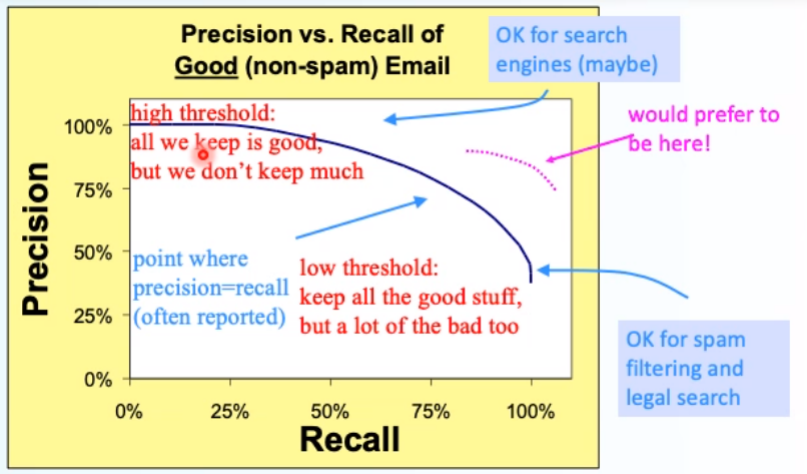
$$C_{NB} = \arg\max_{c_j \in C} \log P(c_j) + \sum_i \log P(x_i \mid c_j)$$

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

$$wx + b$$

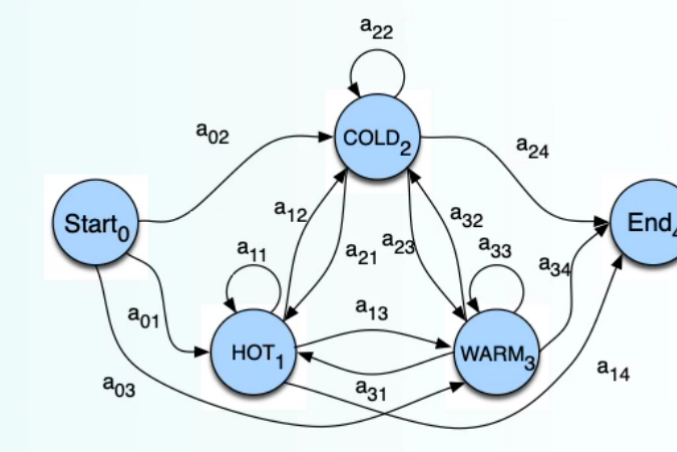
an hyperplane such that the classification rule associated with it has minimal probability of error.
Classification rule:

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

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 $+1$)

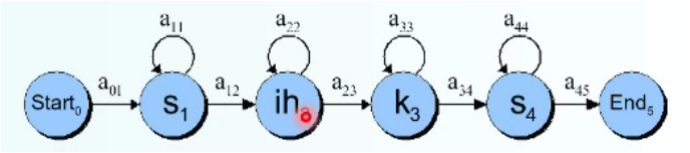
0.7.2 Hidden Markov Models

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

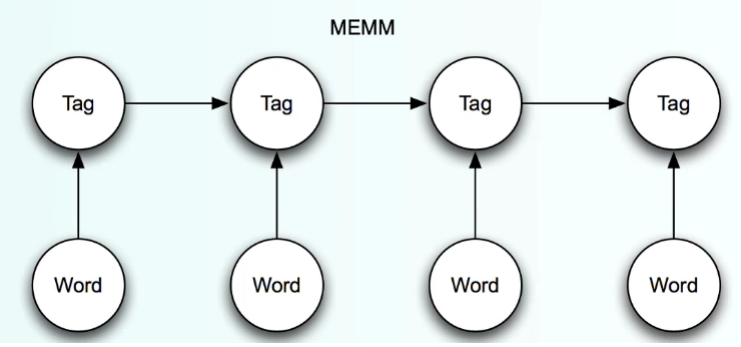
Word likelihood probabilities $P(w_i \mid t_i)$

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...

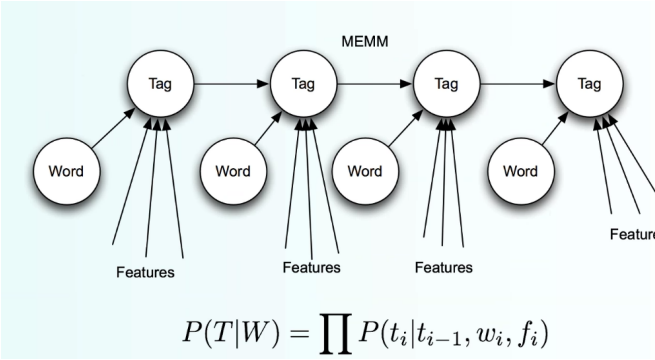
Logistic Regression Model

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

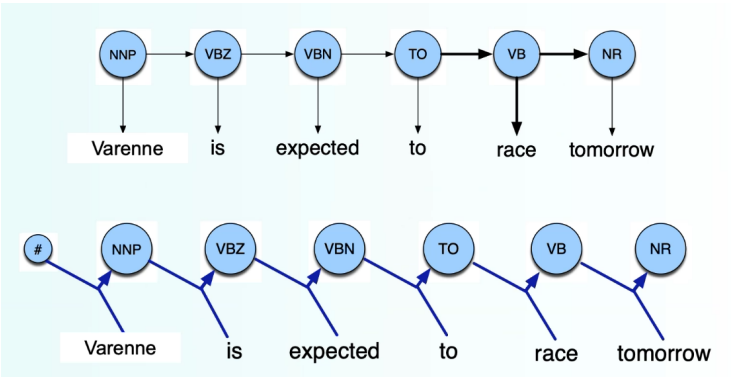


$$P(T \mid W) = \prod P(t_i \mid t_{i-1}, w_i)$$

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



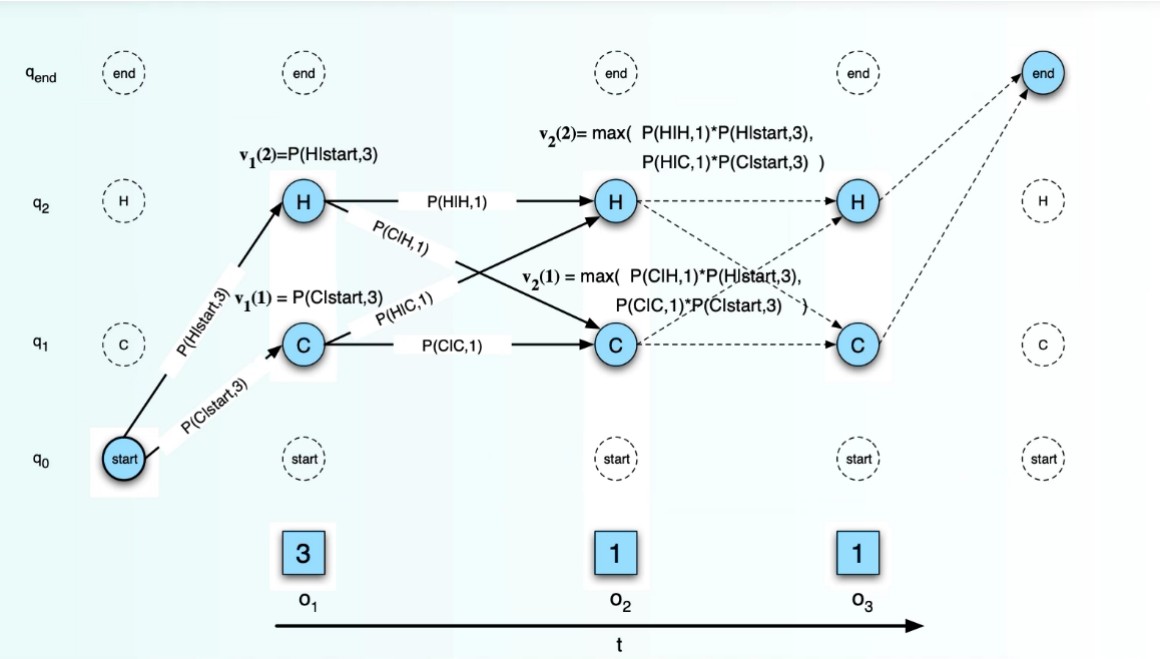
$$P(T|W) = \prod P(t_i|t_{i-1}, w_i, f_i)$$



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

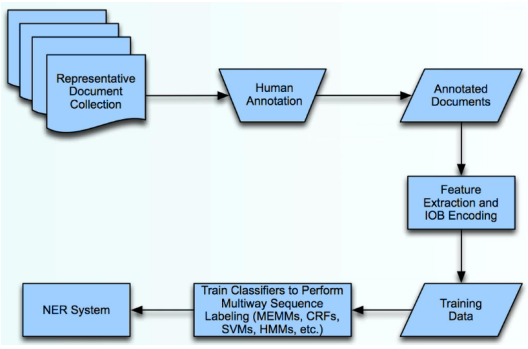
Given a text, find the entities with proper names: person names, city names...the "capitalized things". Typical approach is based on rules, might not be accurate enough. An approach based on ML needs training data, labeling them and using a classifier. Labeling may be easy: annotate each word, but an 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

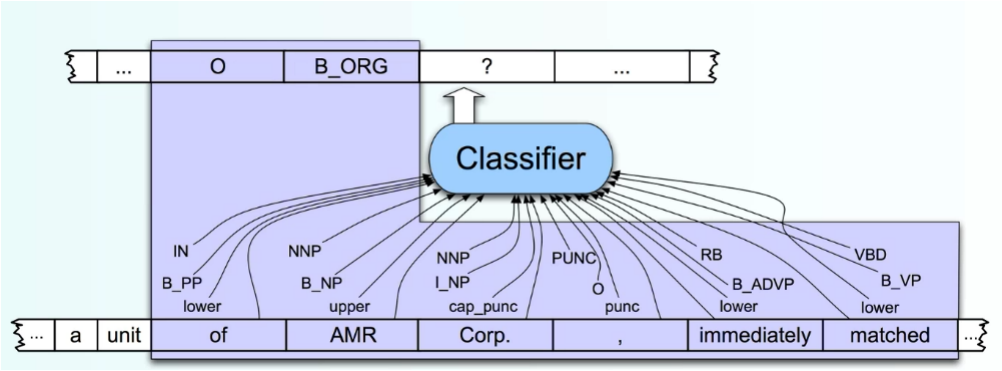
Approaches As with partial parsing and chinking, there are two basic approaches (and hybrids):

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.



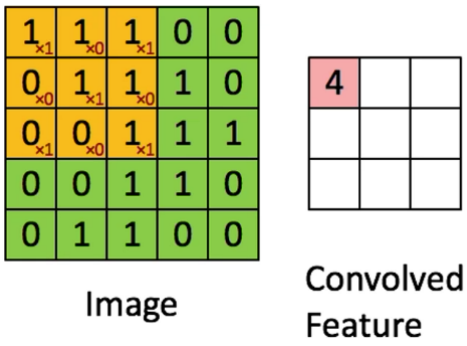
Summary of the approaches.

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

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

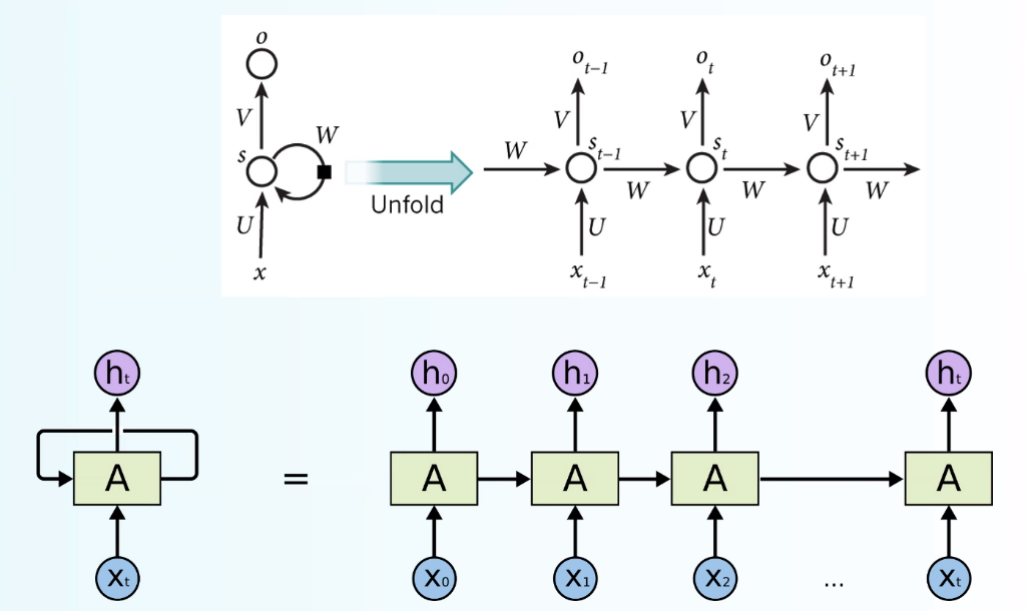
Prevents overfitting. Not used at test time, scaling final vector by probability p. Usually yields an accuracy increase of 2-4%.

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.

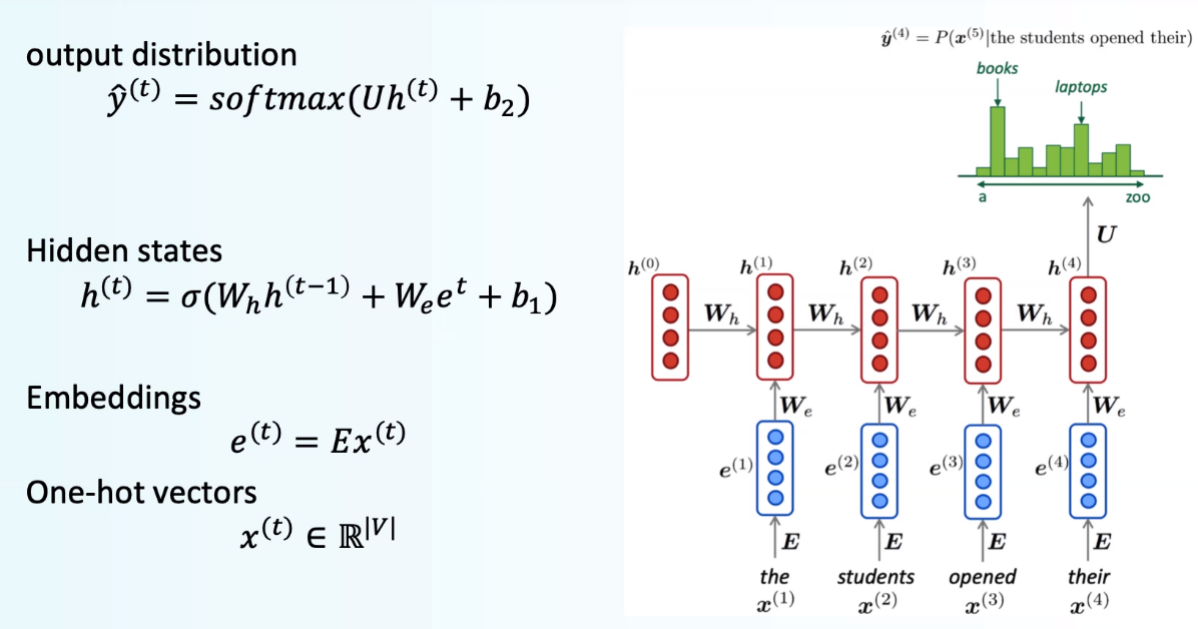


Hidden Units The hidden state s_t represents the memory of the network: it captures information about what happened in all the previous time steps.
The output o_t is computes solely based on the memory at time t.
s_t typically can't capture information from too many time steps ago. Unlike traditional 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.

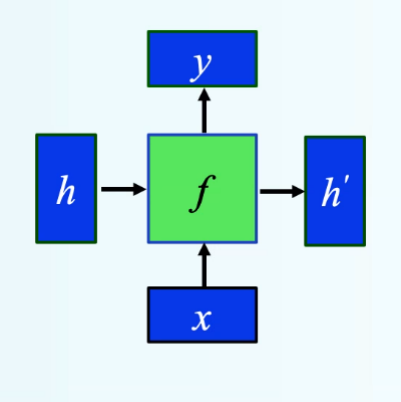
Advantages We can process input of any length and model size doesn't increase with longer input. The computation for step t in theory can use information from many steps back.
Weights are shared across timesteps, so representations are shared too.

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.

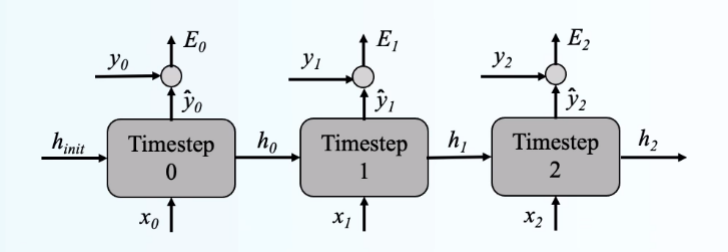
Training Tasks:

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 $x^{(t+1)}$)

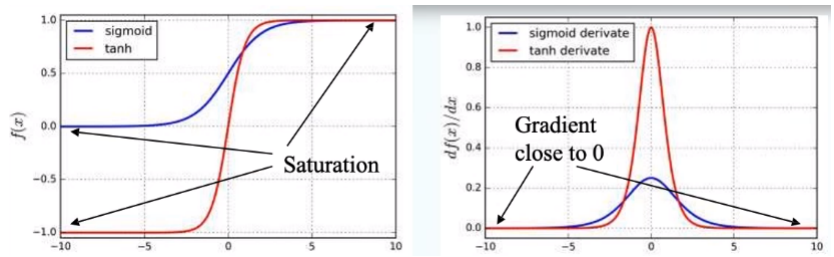
$$J^{(t)}(\Theta) = \text{CrossEntropy}(\hat{y}^{(t)}, y^{(t)}) = -\sum_{w \in V} y_w^{(t)} \log \hat{y}_w^{(t)} = -\log \hat{y}_{x_{t+1}}^{(t)}$$

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

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

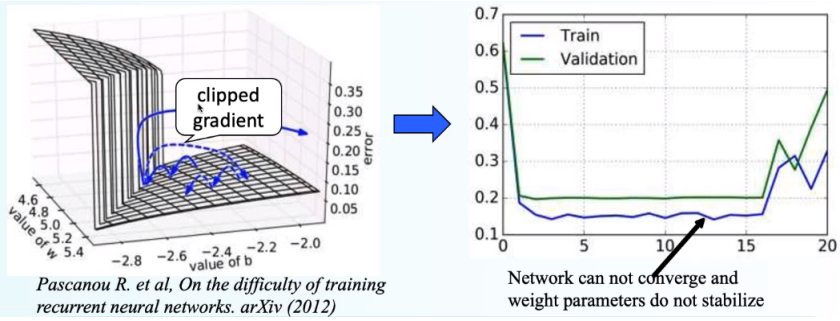
In practice: do this for each sentence and repeat through stochastic gradient descent.

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.

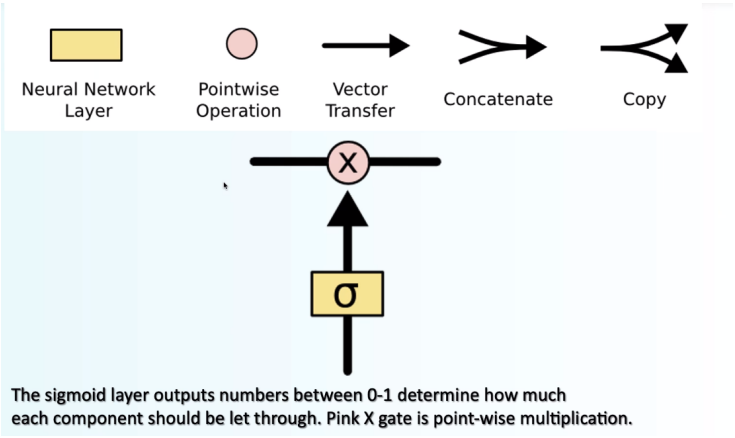
Exploding Gradients The opposite: large increase in the norm, causing NaNs or large fluctuations in cost functions.



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

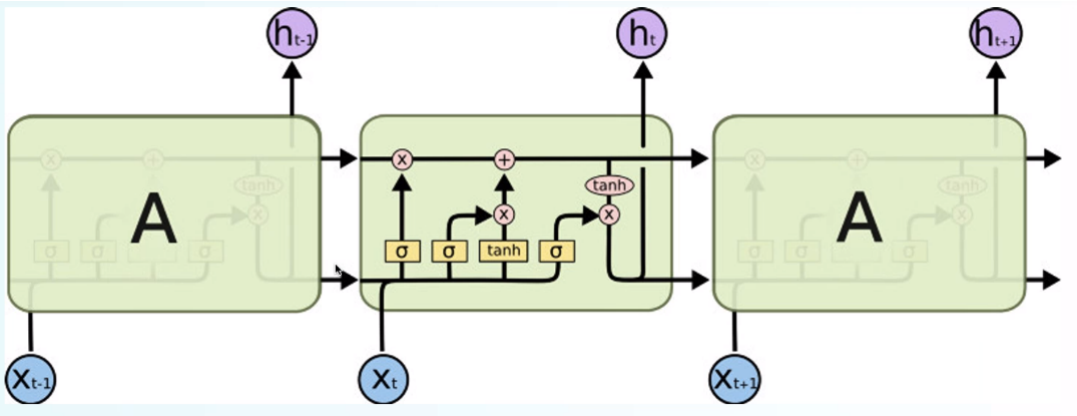
0.9.1 Specializations

Notation



LSTM

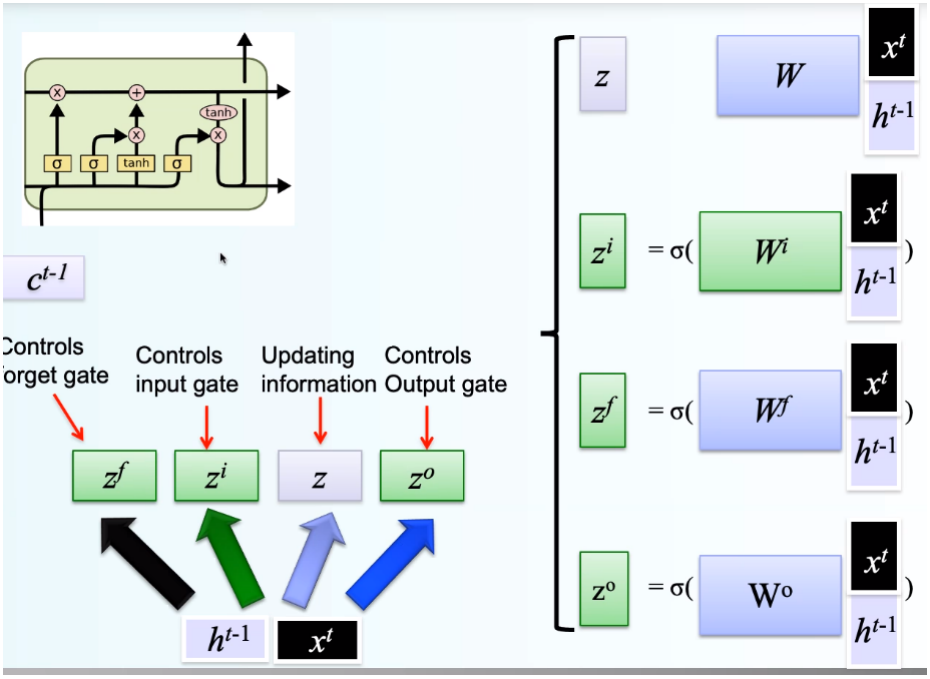
Long Short-Term Memory

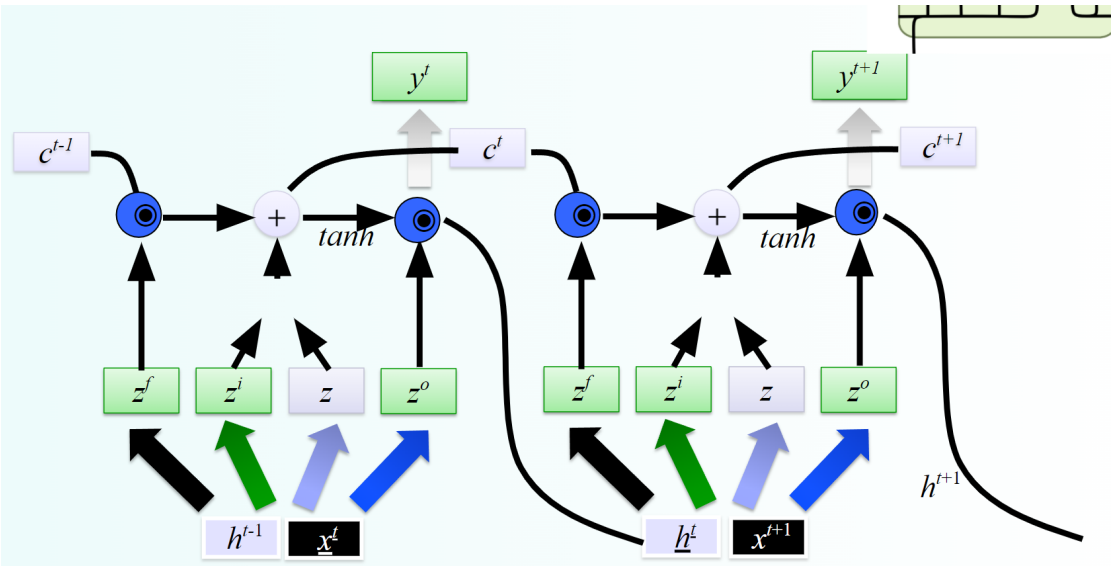


The core idea is this cells state C_t is changed slowly with only minor interactions. Very easy for information to flow unchanged.

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

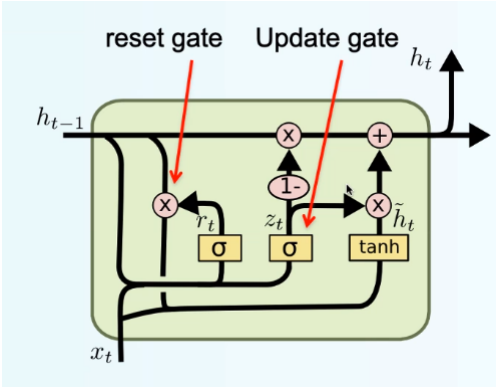
The first sigmoid goes to the **forget gate**, determines how much information goes through.
the second sigmoid is the **input gate** and decides how much input is added in the cell state (so in the next pass).
The **output gate** of the third sigmoid controls what goes in the output.
Why sigmoid or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





GRU

Gated Recurrent Units



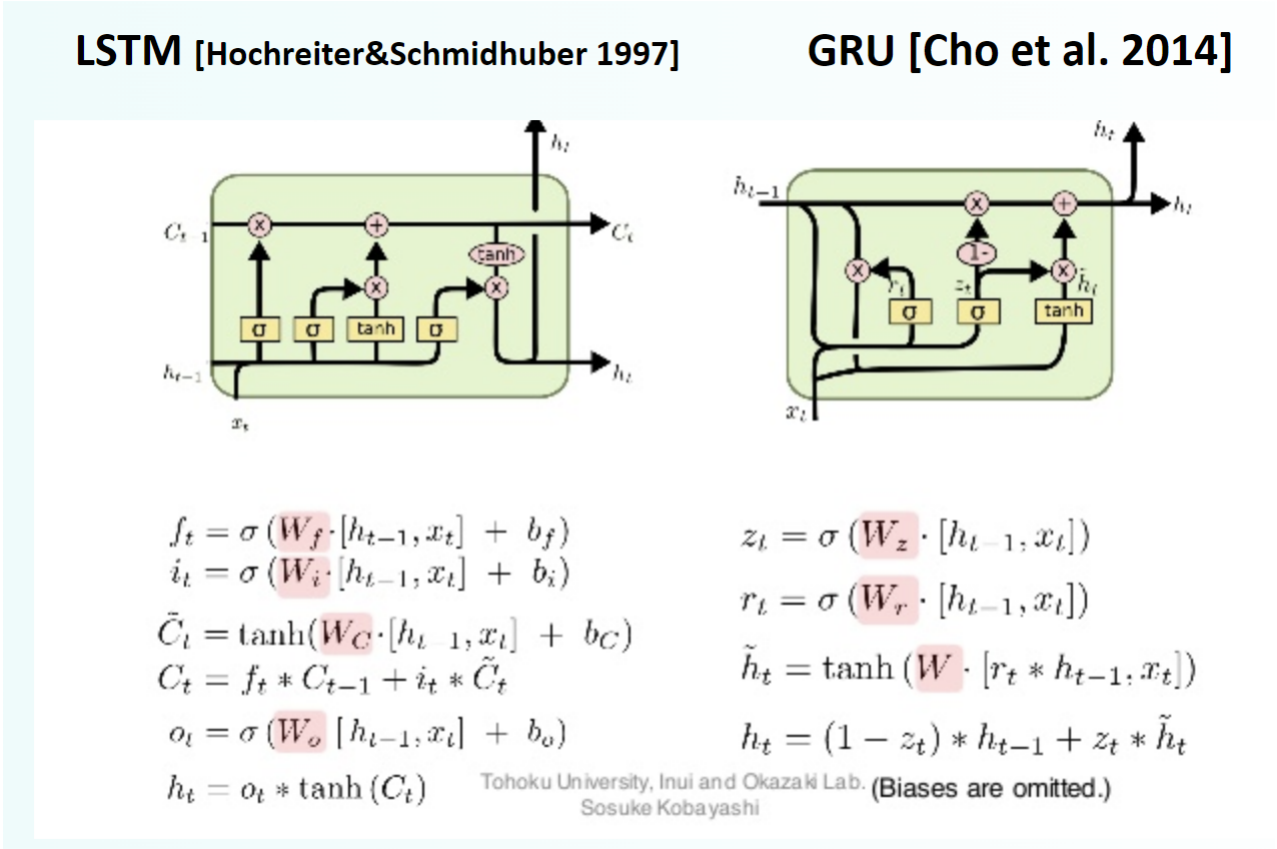
$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$

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

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

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

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

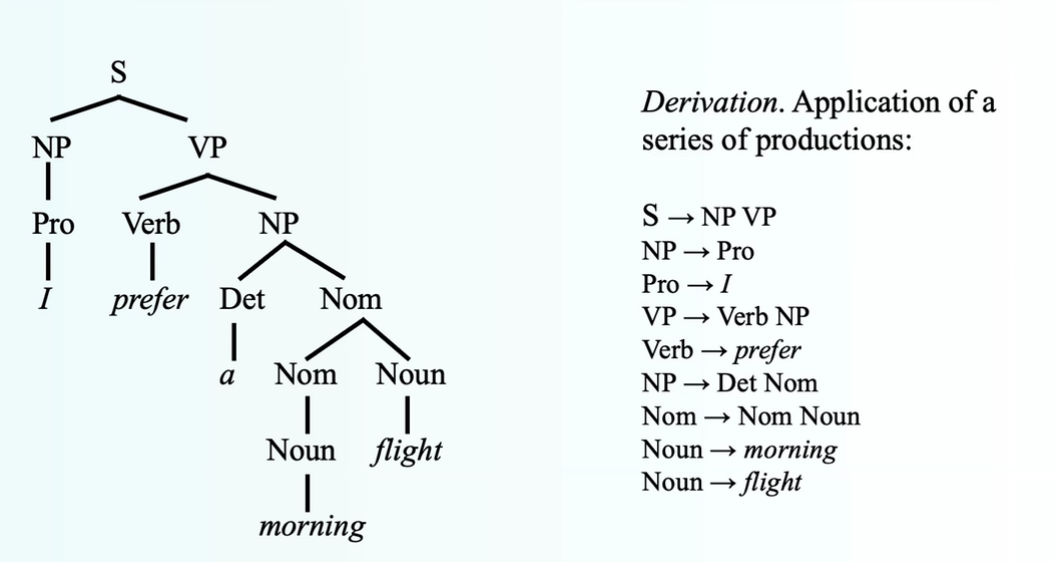
Question Answering

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

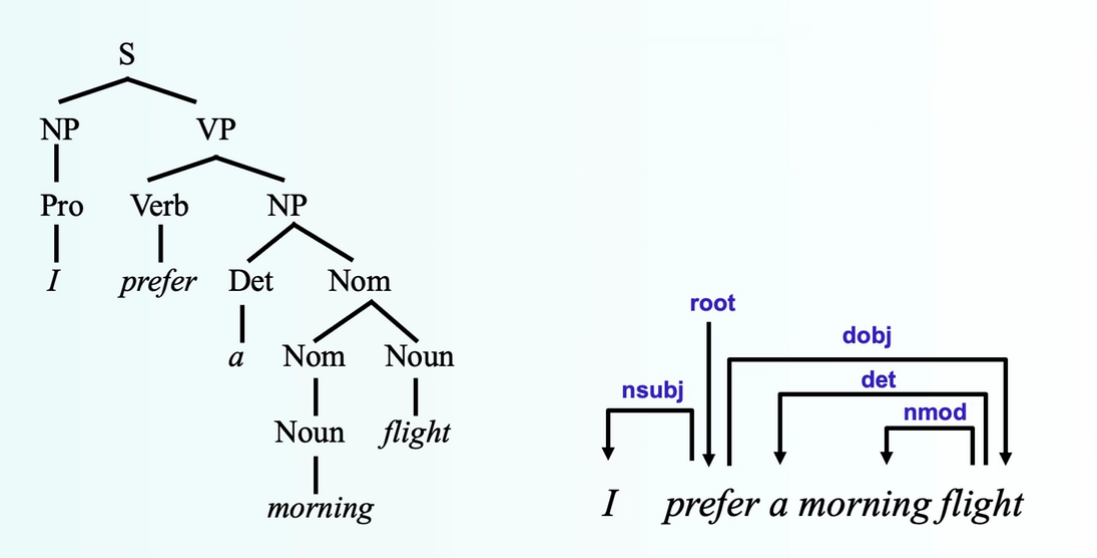
Training by giving a set of sentences X and the set of possible outputs (trees) Y to learn a function $F_W : X \to Y$. Parsing is choosing the highest scoring tree

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

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 Traditional statistical parsers are trained directly on the task of tagging a sentence. Instead, a shift-reduce parser **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.

Emulate the parser and at each parser state extract a context representation of the state, in terms of features.

Provide the features as input and the suggested action as output to the classifier.

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.

Problems with Oracles Only suggest the correct path. If a parser makes mistakes, it finds itself in a state never seen in training and doesn't know how to recover, causing error propagation.

Graph-Based Parsing

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

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

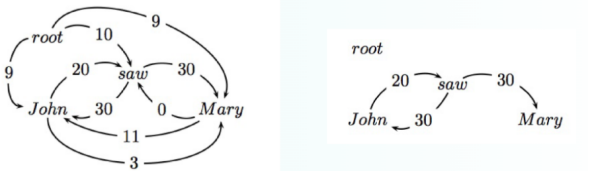
$A_x = \{(i, j, k) \mid i, j \in V \text{ and } k \in L\}$

A key observation is that valid dependency trees for x are **directed spanning trees of** G_x.
A spanning tree is a tree that contains all the vertexes of the original tree and a subset of the arcs, only those needed to connect all the vertexes with one and only one path.
The score of the dependency tree T is given by the score of its arcs:

$$s(T) = \sum_{i,j,k \in T} s(i, j, k)$$

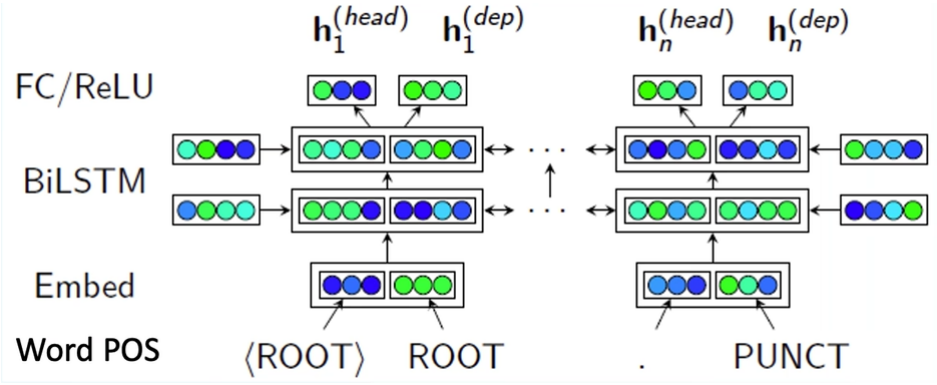
The learning is about the scoring functions $s(i, j, k)$ for each arc (i, j, k). Inference is the search for the maximum spanning tree T of G_x given $s(\,)$.

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. Then recalculate 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 although slower than neural dependency-based parsers.
Bidirectional LSTMs over word/tag embeddings.



Parser Two separate FC ReLU (Fully Connected Rectified Linear Units) layers:

One representing each token as a dependent trying to find its head.

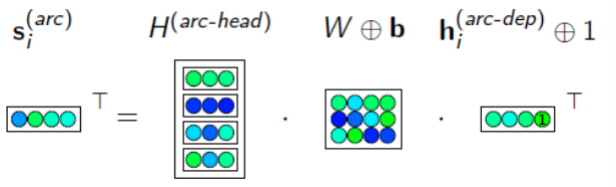
One representing each token as a head trying to find its dependents.

Dependency Relations Two separate FC ReLU layers:

One representing each token as a dependent trying to determine its label.

One representing each token as a head trying to determine its dependents labels.

Self-Attention Biaffine self-attention layer to score a possible heads for each dependent. A $n \times n$ matrix score:

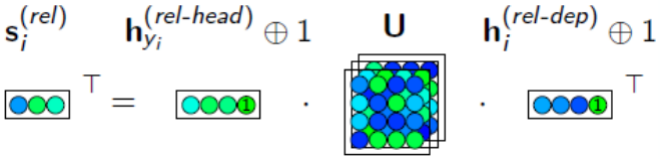


$$s_i = H^{(\text{arc-head})} \cdot \left(W h_i^{(\text{arc-dep})} + b \right)$$

$$H^{(\text{arc-head})} = \left[h_1^{(\text{arc-head})} \ \ldots \ h_n^{(\text{arc-head})} \right]$$

Train with cross-entropy and apply a spanning tree algorithm at inference time.

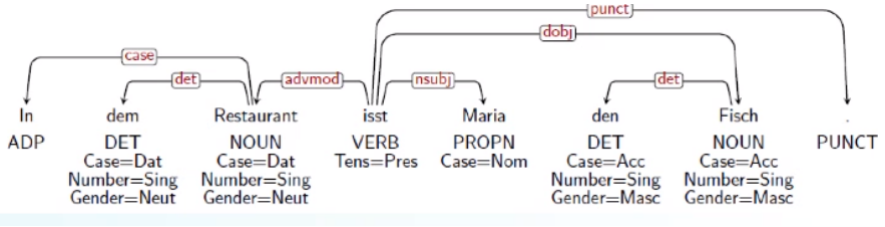
Classifier for Labels Biaffine layer to score possible relations for each best-head/dependent pair, $n \times c$



Train with softmax cross-entropy, added to the loss of the unlabeled parser.

0.11 Universal Dependencies

Treebank annotation schemes vary across languages, hard to compare results across them. Also hard to use to build multilingual.



Goal Facilitate consistent notation of similar constructions across languages. Support multilingual NLP and linguistic research. Build on common usage and existing de-facto standards. Complement and not replace language-specific schemes. **Community effort**.

Guiding Principles Allow parallelism across languages:

Don't annotate same thing in different ways

Don't make different things look the same

Don't annotate things that are not there

Use a universal pool of categories

Allows language-specific axioms.

Design Principles

Dependency: widely used in practical NLP systems, available in treebanks for many languages.

Lexicalism: the basic annotation units are syntactic words, they have morphological properties and can enter into syntactic relations.

Recoverability: transparent mapping from input text to word segmentation.

Morphological Annotation

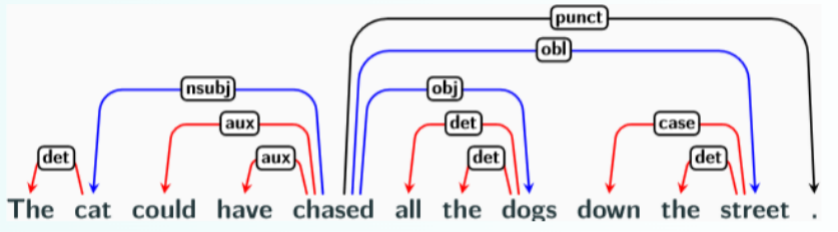
Le La DET Definite=Def Gender=Masc Number=Sing	chat chat NOUN Gender=Masc Number=Sing	chasse chasser VERB Mood=Ind Number=Sing Person=3	les le DET Definite=Def Gender=Masc Number=Sing	chiens chien NOUN Gender=Masc Number=Plur	. . PUNCT

Lemma represent the semantic content of a word

Part-of-Speech tag represent its grammatical class

Morphological Features represent lexical and grammatical properties of the lemma or the specific word form

Syntactic Annotation

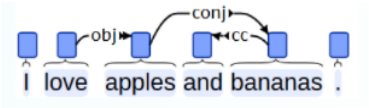


Content Words are used as heads of dependency relations

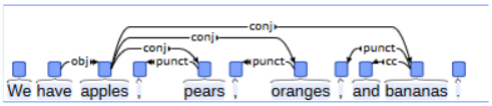
Function Words attach to the content word they modify

Punctuation attach to the head of phrase or clause

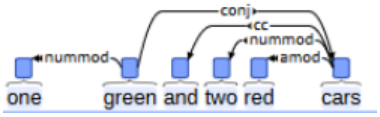
Coordination The first conjunct is the head of all following conjuncts.



Attach coordinating conjunctions and punctuation to the immediately succeeding conjunct.



Except for the right headed constructions.

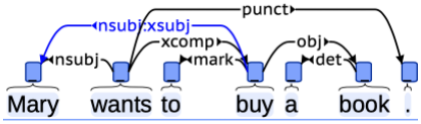


Enhanced Dependencies Making some of the implicit relations between words more explicit, to facilitate relation extractions

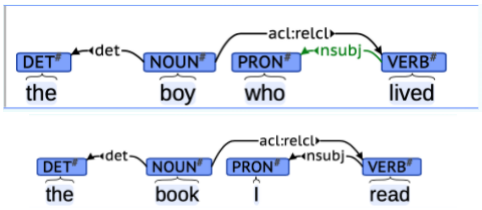
Propagation of conjuncts

Conjoined verb and phrases

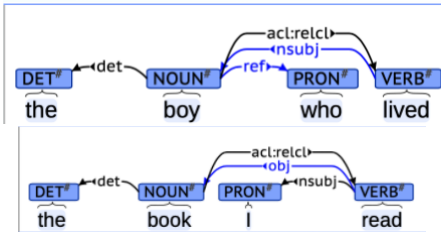
Subject from controller phrases



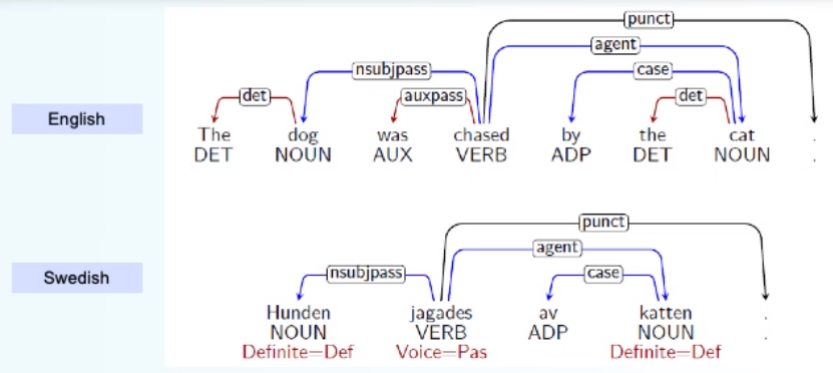
Basic



Enhanced



Dependency Structure The same concept expressed in different languages is represented in similar ways.



Keeping the content words as heads promotes parallelism across languages. The main grammatical relations involving a passive verb, a nominal subject and an oblique agent are the same.

Parsing Can train a parser, also possible to train on multiple languages.

0.12 Machine Translation

Main task that let to the creation of the field. MT is translating a text from one language to another.

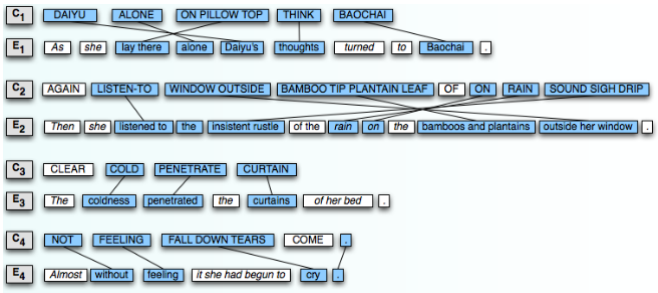
Issues

Sentence segmentation (e.g. 4 English sentences to 1 Chinese sentence)

Grammatical differences

Stylistic and cultural differences

Alignment



Not just literature! For example, in the European Parliament there are more official languages so all official documents have to be produced in all official languages.

MT Already Good for...

Tasks for which a rough translation is fine: extracting informations, web pages, email...

Tasks for which the MT can be post-edited: MT as first-pass, computer-aided human translation...

Tasks in sublanguage domain where high-quality MT is possible: FAHQT (Fully Automatic High Quality Translation)

MT Not Yet Good Enough for...

Really hard stuff: literature, natural spoken speech...

Really important stuff: medical translations in hospitals, emergency phone calls...

0.12.1 Language Similarities and Divergences

Some aspects of human language are universal or near-universal, others diverge greatly.

Typology Systematic study of these similarities and divergences. What are the dimension along which human languages vary?

Morphology

Morpheme: minimal meaningful unit of language.

Word: Morpheme + Morpheme + ...

Stems root plus derivational morphemes
Hope+ing ⇒ **hop**ing **Lemma**: also called base form, root, lexeme
Hoping ⇒ **Hope Affixes**:

Prefixes

Suffixes

Infixes

Circumfixes

Morphological Variation In isolating languages a single word generally have one morpheme, while in polysynthetic languages single words can have many morphemes.
In agglunative languages morphemes have clean boundaries, in fusion languages a single affix can have many morphemes.

So there's a wide range of synthesis: from vietnamese (isolating) to english to russian to onedia (synthetic).
The other range is fusion: from swahili (agglunative) to russian and oneida (fusion).

Segmentation Variation Not every writing system has word boundaries marked between words, also some languages have very long sentences.
Some cold languages require the hearer to do more "figuring out", while in hot languages for example the subject of the sentence is always required.

Lexical Gaps For example, Japanese doesn't have a word for "privacy", or English lacks the word for the Japanese "Oyakoko".

Event-To-Argument Divergences In verb-framed languages we mark the direction of motion on verb, while in satellite-framed languaged we mark the direction in the satellite.

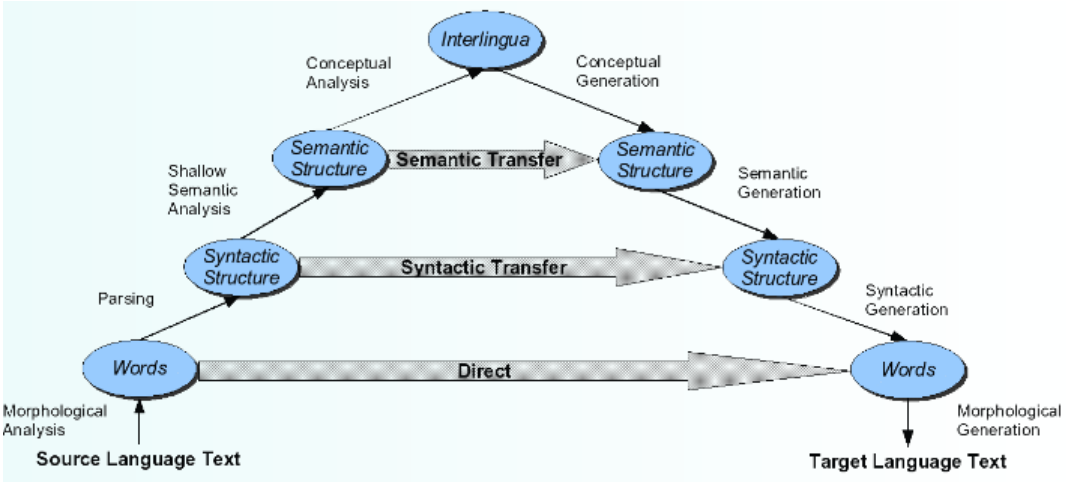
0.12.2 Classical Techniques

Three classical ones:

Direct

Transfer

Interlingua



Direct Translation Proceed word-by-word in the text translating each, without intermediate structures except morphology. Knowledge is in the form of a huge bilingual dictionary and some word-to-word translation information. The dictionary can be more specific for each word, specifying cases and different translation for each case.
The main problem is that we don't "translate" the different syntactic structures and maintain the order of the words of the original language which is often different in the target language.

Pros Fast, simple, cheap, no translation rules hidden in the lexicon.

Cons Unreliable, not powerful, rule proliferation, need major restructuring after lexical subsitution.

Transfer Model We use a set of **transfer rules** that restructure the parse tree. But we need hard to obtain lexical transfer rules. We apply contrastive knowledge: knowledge about the difference between two languages.

Analysis: syntactically parse source language

Transfer: rules to turn this parse into parse tree for target language

Generation: generate target sentence from parse tree

Lexical Transfer Transfer-based systems also need lexical transfer rules, bilingual dictionaries. Can be a list or a word disambiguation system.

Systram Combination of direct and transfer.

Analysis:

Morphological analysis, POS tagging
Chunking of NPs, PPs, phrases
Shallow dependency parsing

Transfer:

Translation of idioms
Word sense disambiguation
Assigning prepositions based on governing verbs

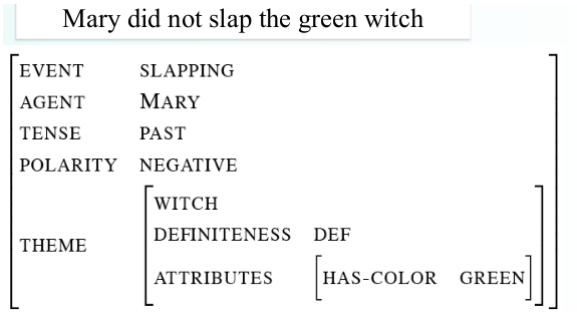
Synthesis:

Apply rich bilingual dictionary
Deal with reordering
Morphological generation

N^2 set of transfer rules, grammar and lexicon full of language-specific stuff, hard to build and maintain.

Interlingua Instead of language-to-language rules, we abstract the meaning of the sentence and translate that.

1. Translate source sentence into meaning representation
2. **Generate** target sentence from meaning



The idea is that some of the MT work that we need to do is part of other NLP tasks: disambiguating eng:book-ita:libro from eng:book-ita:prenotare. So we could have concepts like BOOKVOLUME and PRENOTARE and solve this problem once for each language.

Pros Avoids the N^2 problem, and easier to write rules.

Cons Semantics is hard, useful information lost (because we paraphrase).

0.12.3 Statistical Machine Translation

Example Start from a parallel corpus: a set of sentences in a language and the corresponding sentences in the other language.
Have a sentence to translate into an unknown target sentence into the target language. For each word in the sentence, look for that into the parallel corpus, trying to figure out the translation.

What Makes a Good Translation Two facts to maximize:

Faithfulness or fidelity: how close is the meaning of the translation to the meaning of the original.
Even better: does the translation cause the reader to draw the same inferences as the original would have?

Fluency or naturalness: how natural the translation is, just considering its fluency in the target language.

Faithfulness and fluency are formalized: best-translation $\hat{T}$ of a source sentence S

$$\hat{T} = \arg\max_T \{\text{Fluency}(T) \cdot \text{Faithfulness}(T, S)\}$$

Called the **IBM model**. It's the Bayes rule

$$\hat{T} = \arg\max_T P(T)P(S \mid T)$$

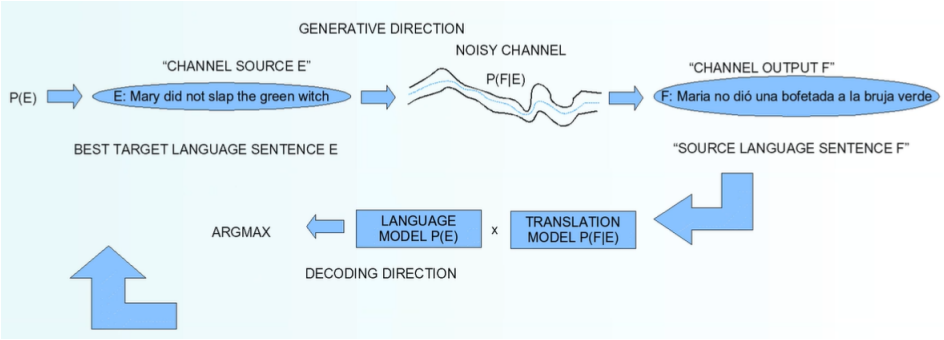
More formally: assume we want to translate from a foreign language sentence F to an English language sentence E

$$F = f_1 \dots f_m$$

We want to find the best English sentence

$$\overline{E} = e_1 \dots e_n = \arg\max_E P(F \mid E)P(E)$$

With $P(E)$ from the language model and $P(F \mid E)$ from the translation model.
Also known as noisy model



Fluency $P(T)$ How to measure that a sentence is more fluent than another? E.g. "That car was almost crash onto me" is less fluent than "That car almost hit me".
The answer is language models, n-grams! For example, $P(\text{hit} \mid \text{almost}) > P(\text{was} \mid \text{almost})$.
But can use any other more sophisticated model of grammar. Advantage: it's monolingual knowledge.

Faithfulness $P(S \mid T)$ How to quantify? Intuition: degree to which words in one sentence are plausible translations of words in the other sentence. Product of probabilities that each word in target sentence would generate each word in source sentence.
Need to know, for every target language word, probability of it mapping to every source language word. How to learn this? Parallel texts: lots of times we have two texts that are translations of each other. If we knew which word in Source text mapped to each word in Target text, we could just count.

Sentence Alignment Figuring out which source language sentence maps to which target language sentence.

Word Alignment Figuring out which source language word maps to which target language word.

The faithfulness model $P(S \mid T)$ just models a "bag of words" which words come from e.g. English to Italian.
$P(S \mid T)$ doesn't have to worry about internal facts about target word order, that's the job of $P(T)$.
$P(T)$ can do bag generation: put the following words in order.

Three Problems for Statistical Machine Translation

Language Model: given an English string e, assigns $P(e)$ by a formula such that:

Good English string $\Rightarrow$ high $P(e)$
Random word sequence $\Rightarrow$ low $P(e)$

Can be trained on large, unsupervised mono-lingual corpus for the target language, and could use more sophisticated PCFG language model to caputre long-distance dependencies. Terabytes of web data used to build large 5-gram models.

Translation Model: given a pair of strings $\langle f, e \rangle$, assigns $P(f \mid e)$ by a formula such that:

$\langle f, e \rangle$ look like translations $\Rightarrow$ high $P(f \mid e)$
$\langle f, e \rangle$ don't look like translations $\Rightarrow$ low $P(f \mid e)$

Decoding Algorithm: given a language model, a translation model and a new sentence f, it finds the translation e that maximize $P(e)P(f \mid e)$

0.12.4 Phrase Based Machine Translation

Follows three steps

1. Group words into phrases
2. Translate each phrase
3. Move the phrases around

$P(F \mid E)$ is modeled by translating phrases in E to phrases in F. First segment E into a sequence of phrases $\overline{e}_1, \ldots, \overline{e}_I$, then translate each $\overline{e}_i$ into $\overline{f}_i$ based on **translation probability** $\phi(\overline{f}_i \mid \overline{e}_i)$. Then, reorder translated phrases based on **distortion probability** $d(i)$ for the $i$the phrase.

Translation Probabilities Assuming a phrase aligned parallel corpus is available or constructed that show matching between phrases in E and F. Then compute (MLE) estimate of ϕ based on simple frequency counts

$$\phi(\overline{f}, \overline{e}) = \frac{\text{Count}(\overline{f}, \overline{e})}{\sum_f \text{Count}(f, \overline{e})}$$

Distortion probability The probability that a phrase in position X in the original sentence moves to position Y in the translation.
Distortion is the measure of distance between positions of corresponding phrases in the 2 languages. Distortion of phrase i as the distance between the start of the foreign phrase generated by $\overline{e}_i$ (a_i and the end of the foreign phrase generated bu the previous phrase $\overline{e}_{i-1}$ (b_{i-1}). Typically we assume the probability of a distortion decreases exponentially with the distance of the movement

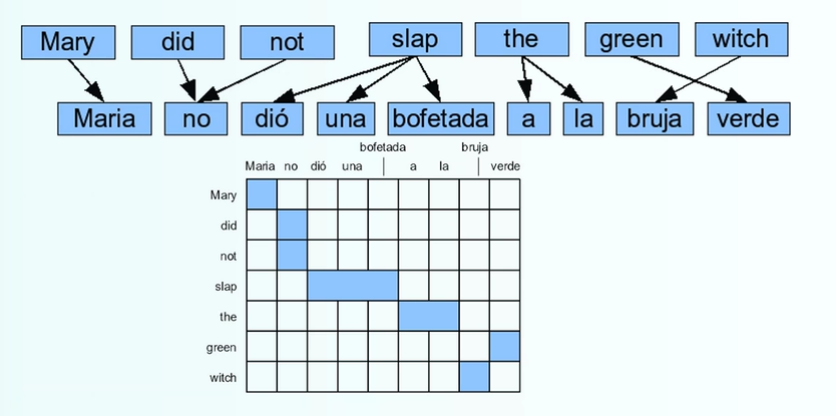
$$d(i) = c\alpha^{|a_i - b_{i-1}|}$$

Set $0 < \alpha < 1$ base on fit to phrase-aligned training data. Then set c to normalize $d(i)$ so that it sums to 1.

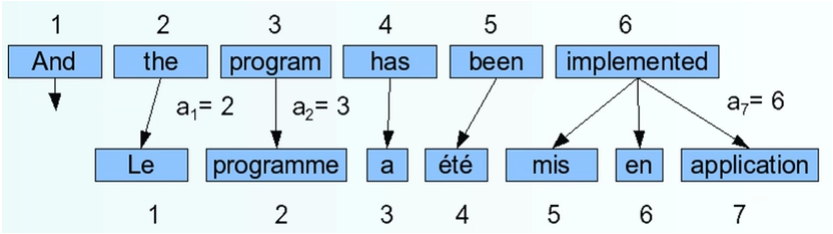
Training $P(F \mid E)$ What we mainly need to train is $\phi(f_j \mid e_i)$
Suppose we had a large bilingual training corpus: a **bitext**. Suppose also to know exactly which phrase in the target language was the tranlsation of which phrase in the source language: **phrase alignment**.
If we had this, we could just count-and-divide

$$\phi(\overline{f}, \overline{e}) = \frac{\text{Count}(\overline{f}, \overline{e})}{\sum_f \text{Count}(f, \overline{e})}$$

But we don't have phrase alignments. We have **word alignments**.



Actually we have more restrictive word alignments. Word alignments are mapping between source and target words in parallel sentence. Restriction: each foreign word come from exactly one source word.



Advantage: represent alignment by the index of the English word that the french word comes from. The above alignment thus is 2,3,4,5,6,6,6.
Spurious words are words in the foreign sentence that does not align with any word in the source language. We assume one to many alignment: each word in F aligns with 1 word in E, with some words in F coming from NULL.

Computing Word Alignments IBM-Model 1. For phrase-based machine translation we need a word-alignment to extract a set of phrases.
A word alignment model gives us $P(F, E)$, and we want this to train our phrase probabilities $\phi(f_j \mid e_j)$ as part of $P(F \mid E)$.
A word-alignment model allows to compute the translation probability $P(F \mid E)$ by summing the probabilities of all possible $(l+1)^m$ "hidden" alignments A between F and E

$$P(F \mid E) = \sum_A P(F, A \mid E)$$

IBM-Model 1 assumes the following simple generative model of producing F from $E = e_1, \ldots, e_l$:

1. Choose length J of F sentence: $F = f_1, \ldots, f_J$
2. Choose a 1 to many alignment $A = a_1, \ldots, a_J$
3. For each position j in F, **generate** a word f_j from the aligned word e_{a_j} in E

Its goal is to find the most probable alignment given a prameterized model

$$\hat{A} = \arg\max_A P(F, A \mid E) = \arg\max_A \prod_{j=1}^{J} t(f_j, e_{a_j})$$

Since translation choice for each position j is independent, the product is maximized by maximizing each term

$$a_j = \arg\max_{0 \le i \le l} t(f_j, e_i) \quad 1 \le j \le l$$

Training Alignment Probabilities

Get a parallel corpus, e.g.: Europarl, Hansards...

Sentence alignment. Intuition: use length in words or chars, together with dynamic programming. Ora a simpler MT model.

Use Expectation Maximization to train word alignments.
We can bootstrap alignment from a sentence-aligned bilingual corpus using the EM algorithm.
$P(A \mid E, F)$ is the probability of the alignment A given a translated pair of sentences E and F

$$P(A \mid E, F) = \frac{P(F, A \mid E)}{P(F \mid E)} = \frac{P(F, A \mid E)}{\sum_{A'} P(F, A' \mid E)}$$

Inherent hidden structure is revealed by EM training.

```
Randomly set model parameters, making sure they represent legal distributions
Until convergence (i.e. parameter no longer change) do:
        E Step: compute the probability of all possible alignments
                of the training data using the current model
        M Step: use these alignment probability estimates to re-estimate values
                for all of the parameters
end
```

Phrase-Based Translation Model

Major components of a phrase-based model:

Phrase translation model $\phi(f \mid e)$

Reordering model $\Omega(f \mid e)$

Language model $P_{LM}(e)$

Bayes rule

$$\arg\max_e P(e \mid f) = \arg\max_e P(f \mid e) P(e) = \arg\max_e \phi(f \mid e) P_{LM}(e) \Omega(f \mid e)$$

Sentences f and e are decomposed into I phrases: $\overline{f}_1^I = \overline{f}_1, \ldots, \overline{f}_I$

$$\phi(f \mid e) = \prod_{i=1}^{I} \phi(\overline{f}_1, \overline{e}_i) d(a_i - b_{i-1})$$

Phrase Alignment Alignment algorithms produce one-to-many word translations, and we know that words do not map one-to-one in translations. Better to map "phrases", sequences of words, to phrases and probabilistically reorder them in translation.
Combine $E \to F$ and $F \to E$ word alignments to produce a phrase alignment.

Decoding

Evaluation Human subjective evaluation is best but time consuming and expensive. Automated evaluation comparing the output to multiple human reference translations is cheaper and correlates with human judgments.
Better: computer-aided translation evaluation. **Edit cost**: measure the number of changes that a human translator must make to correct the MT output (number of words changed, amount of time taken, number of keystrokes...).

Scores Based on similarity to the reference translations:

BLEU (Bilingual Evaluation Understudy): determine the number of n-grams of various sizes that the MT output shares with the reference translations. Compute a modified precision measure of the n-grams in MT result, averaging n-gram precision over all n-grams up to size N (typically 4) using the geometric mean:

$$p_n = \frac{\sum_{C \in \text{Corpus}} \sum_{n\text{-gram} \in C} \text{Count}_{\text{clip}}(n\text{-gram})}{\sum_{C \in \text{Corpus}} \sum_{n\text{-gram} \in C} \text{Count}(n\text{-gram})}$$

$$p = \sqrt[N]{\prod_{n=1}^{N} p_n}$$

Brevity Penalty (BP): use a penalty for the translations that are shorter than the reference translation. Define the effective reference length r for each sentence as the length of the reference sentence with the largest number of N-gram matches. With c the candidate sentence length:

$$BP = \begin{cases} 1 & c > r \\ e^{\frac{1-r}{c}} & c \leq r \end{cases}$$

The final BLEU score is

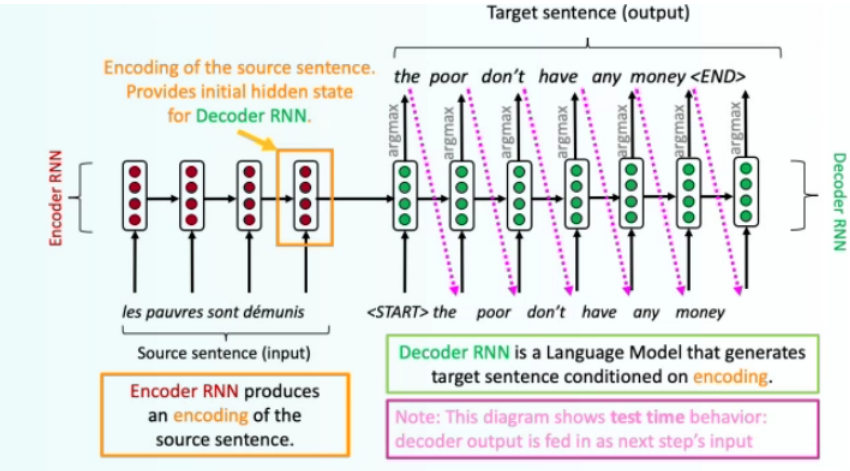
$$BLEU = BP \cdot p$$

Correlates with human evaluation when comparing outputs from different SMT systems, but doesn't correlate with human judgments when comparing SMT systems with manually developed MT or MT with human translations.

0.13 Neural Machine Translation

Statistical Machine Translation is a huge research field, but requires the compilation and maintainment of extra resources (for example tables of equivalent phrases), thus lots of human effort to maintain.
In 2014 neural machine translation was introduced with an huge impact on the machine translation field.

NMT Neural Machine Translation is a way of doing MT with a **single neural network**, a sequence-to-sequence (seq2seq) one.



A seq2seq model is versatile: a neural network takes an input an produces a neural representation, which is used as input by a second network that produces a sequence as output. These models are useful for more than just MT, many natural language processing tasks can be phrased as sequence-to-sequence:

Summarization, long text to short text.

Dialogue, previous utterances to next utterance

Parsing, input text to sequence of parsing symbols

Code generation, natural language to e.g. Python code

The seq2seq model is an example of **conditional language model**:

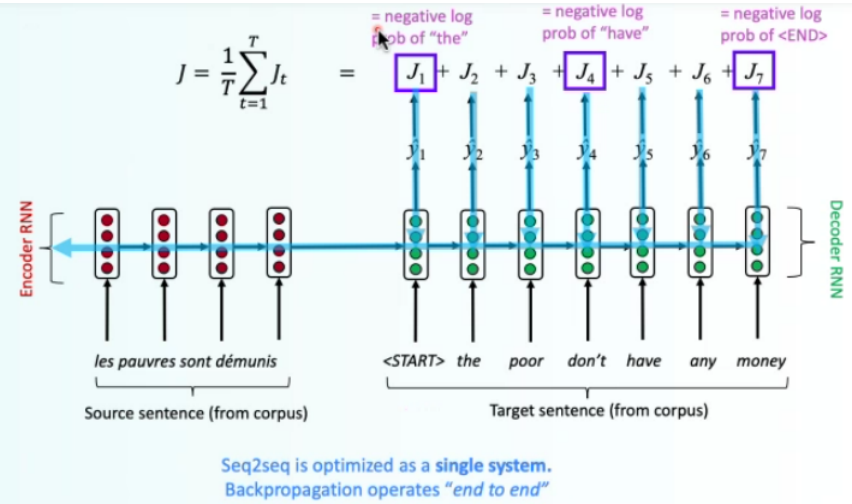
Language model because the decoder is predicting the next word of the target sentence y

Conditional because its predictions are also conditioned on the source sentence x

NMT directly calculates

$$P(y \mid x) = P(y_1 \mid x)P(y_2 \mid y_1, x) \dots P(y_T \mid y_1, \dots, y_{T-1}, x)$$

To train a NMT system, get a big parallel corpus...



Beam Search We showed how to generate (decode) the target sentence by taking the argmax on each step of the decoder. But this is greedy decoding, taking the most probable word on each step, but it has problems e.g. it has no way to undo decisions! A better option would be to use **beam search**, to explore several hypothesis and select the best one. Ideally we want to find a translation y that maximizes

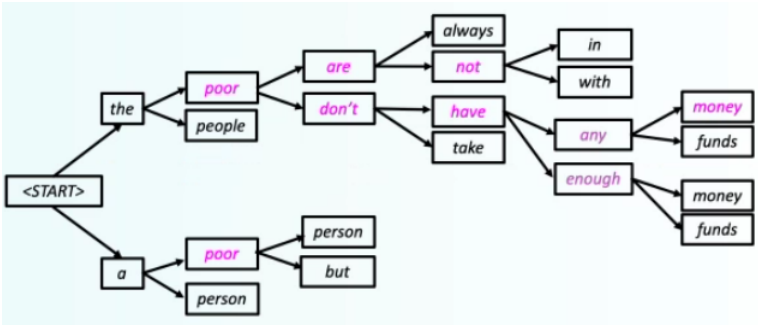
$$P(y \mid x) = \prod_{i=1}^{T} P(y_i \mid y_1, \dots, y_{i-1}, x)$$

We could try computing all possible sequences y: on each step t of the decoder we're tracking V^t possible partial translations with V vocab size. This $O(V^T)$ complexity is too expensive, though, so **beam search**: we keep track of the k most probable partial translations, with k being the beam size ($k \simeq 5, 10$ in practice).
An hypothesis has a score which is its log probability

$$\text{score}(y_1, \dots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i \mid y_1, \dots, y_{i-1}, x)$$

Scores are all negative, the higher the better. We search for high-scoring hypothesis, tracking top k on each step. Doesn't guarantee to find the most optimal solution, but it's far more efficient.

Example With $k = 2$



Then we go back finding the complete translations.

Stopping Criterion In greedy decoding, we decode until we find the END token. In beam search, different hypothesis can produce the END tokens on different timesteps: we place it aside and continue exploring other hypothesis. Usually we continue beam search until:

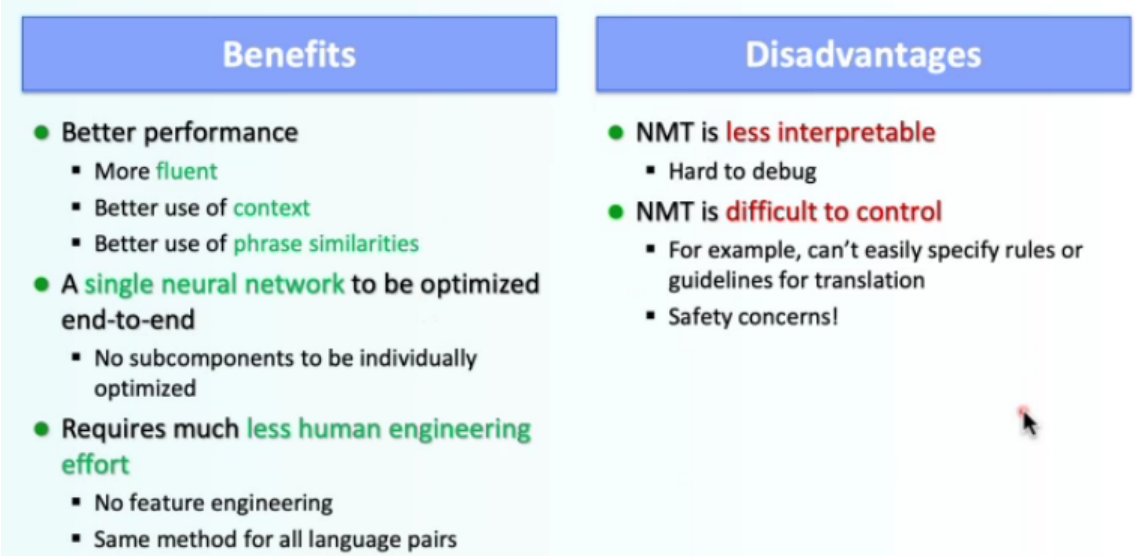
we reach a timestep T, a predefined cut-off, or

we have at least n completed hypothesis, another predefined cut-off

Finishing Up When we have the list of hypothesis, how we select the highest scoring one? Each hypothesis has its score, but longer hypothesis have lower scores so we normalize by length

$$\text{score}(y_1, \ldots, y_t) = \frac{1}{t} \sum_{i=1}^{t} \log P_{LM}(y_i \mid y_1, \ldots, y_{i-1}, x)$$

Pro and Cons



Machine translation is not a solved problem, many difficulties remain:

out-of-vocabulary words

domain mismatch between train and test data

maintaining context over longer texts

failures to accurately capture sentence meaning

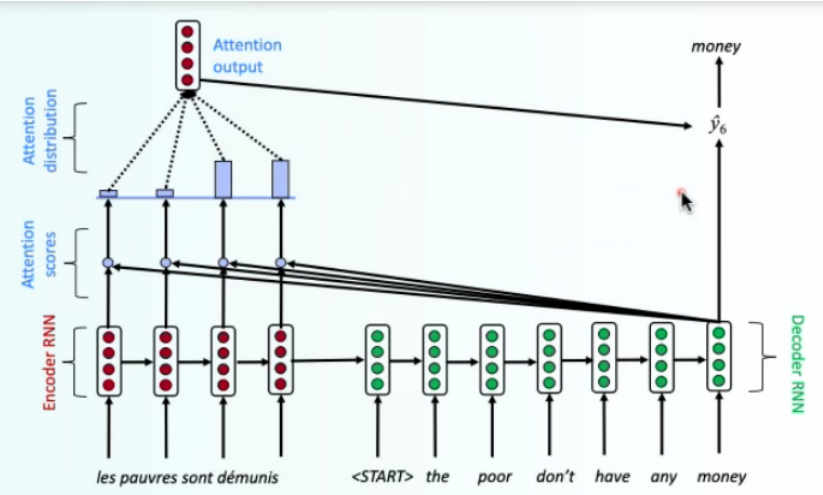
pronouns (or zero pronoun) resolution errors

morphological agreement errors

low-resource language pair

NMT is the flagship task for NLP deep learning: it has pioneered many of the recent innovations of NLP. In 2021 NMT research continues to thrive, researchers have found many improvements to the vanilla seq2seq NMT system presented thus far. But one improvement is so integral that is has become the new vanilla.

Attention Seq2seq has a bottleneck: the encoding needs to capture all the information about the source sentence, it's an **information bottleneck**.
Attention provides a solution to the bottleneck problem. The core idea is: on each step of the decoder, focus on a particular part of the source sequence.



In equations

We have the hidden states in the encoder $h_1, \ldots, h_N \in R^h$

On timestep t we have the decoded hidden state $s_t \in R^h$

We get the attention scores e^t for this step

$$e^t = [s_t^T h_1, \ldots, s_t^T h_N] \in R^N$$

We take the softmax to get the attention distribution for this step (probability distribution that sums to 1)

Attention significantly improves NMT performance, solving the bottleneck problem and helping with vanishing gradient problem. Also provides some interpretability, and we get alignment for free: we never explicitly tranied an alignment system, the network just learned alignment by itself.

Attention Variants We have some values $h_i \in R^h$ and a query $s \in R^h$. Attention involves:

Computing the attention scores $e \in R^N$

Taking softmax to get a attention distribution

$$\alpha = \text{Softmax}(e) \in R^N J$$

Using α to take the weighted sum of values

$$\alpha = \sum_{i=1}^{N} \alpha_i h_i \in R^h$$

Thus obtaining the **attention output** α (sometimes called the **context vector**)

We've seen it in translation, but this can be applied to other NLP tasks. It has becomes a general deep learning technique as well. A more general definition of attention is: given a set of vector values, and a vector query, attention is a technique to compute a weighted sum of the values dependent on the query.
We sometimes say that the attention attends to the query.
Attention is all you need. Self-attention helps to contextualize words: a limit on word embeddings was that each word had a single vector, with attention we can introduce into the representation the context into which they appears, allowing to observe other words.

0.13.1 Self-Attention

Sequence to sequence models with attention are quite effective in transduction tasks, but their sequential nature limits parallelism. The **transformer** transduction model relies entirely on self-attention to compute representations of its input and output: it doesn't use sequence aligned RNNs nor convolutions.
As a result, training costs are reduced by 1-2 orders of magnitude.

So we want **parallelization** but RNNs are inherently sequential. They also generally need attention mechanism to deal with long range dependencies: path length between states grows with distance otherwise. We may be able to just use attention and skip the need for the RNN.

Attention Given a set of vector values and a vector query, attention is a technique to compute a weighted sum of the values dependent on the query.
The intuition is the following: the weighted sum is a selective summary of the information contained in the values, where the query determines which values to focus on. Attention is a way to obtain a fixed-size representation of an arbitrary set of representations (the values), dependent on some other representation (the query).

The generic attention, as we have seen before: we have some **values** $v_1, \dots, v_N \in R^{d_1}$, some **keys** $k_1, \dots, k_N \in R^{d_1}$ and a **query** $s \in R^{d_2}$

Compute the **attention scores** $e \in R^N$

Taking softmax to get the **attention distribution** $\alpha = \text{Softmax}(e) \in R^N$

Using the attention distribution to take the weighted sum of values

$$a = \sum_{i=1}^{N} \alpha_i v_i \in R^{d_1}$$

Thus obtaining the **attention output** or context vector a

There are variants, several ways to compute $e \in R^{d_1}$:

Basic dot-product attention $e_i = s^T k_i \in R$
This assumes $d_1 = d_2$, and is the version used in NMT

Multiplicative attention $e_i = s^T W k_i \in R$
Where $W \in R^{d_1 \times d_2}$ is a weight matrix

Additive attention $e_i = w^T \tanh(W_1 k_1 + W_2 s) \in R$
Where $W_1 \in R^{d_3 \times d_1}$, $W_2 \in R^{d_3 \times d_2}$ are weight matrices and $w \in R^{d_3}$ is a weight vector.
d_3 is the attention dimensionality, hyperparameter.

Issues with Recurrent Models

O(sequence length) steps for distant word pairs to interact. This means: hard to learn long-distance dependencies (because gradient problems!) and linear order of words "baked in" (and we already know that linear order isn't the right way to think about sentences

Non-parallelizable

What can we use instead of recurrence?

Word Windows Word window models aggregate local contexts (also known as 1D convolution). Stacking window layers allow interaction between farther words. The maximum interaction distance is $\frac{\text{sequence length}}{\text{window size}}$

Attention Attention treats each word's representation as a query to access and incorporates information from a set of values. We saw attention from decoder to encoder, let's see attention withing a single sentence. The maximum interaction distance becomes O(1) since all words interact at every layer!

Self-Attention We can think of attention as an approximated hashtable:

To look up a value we compare a query against keys in the table

In a hashtable: each query (hash) maps to exactly one key-value pair

In (self-)attention: each query matches each key to varying degrees. We return a sum of values weighted by the query-key match.

Given the **queries** $q_1, \ldots, q_T \in R^d$, the **keys** $k_1, \ldots, k_T \in R^d$ and the **values** $v_1, \ldots, v_T \in R^d$ (but in practice the number of queries can differ from the number of keys and values), in self-attention **the queries, keys and values are drawn from the same source**. E.g.: if the output of the previous layer is $x_1, \ldots, x_T$ (one vector per word), we could use $v_i = k_i = q_i = x_i$ (same vector for all of them).
The dot product self-attention operation is:

Compute key-query quantities

$$e_{ij} = q_i^T k_j k$$

Compute attention weights from affinities

$$\alpha_{ij} = \frac{e^{e_{ij}}}{\sum_k e^{e_{ik}}}$$

Compute outputs as weighted sum of values

$$\text{Output}_i = \sum_j \alpha_{ij} v_j$$

Vector Notation

With embeddings stacked in X, compute queries, keys and values: $Q = XW^Q, K = XW^K, V = XW^V$

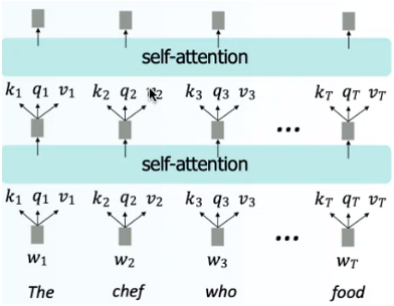
Compute attention scores between queries and keys $E = QK^T$

Take the softmax to normalize attention scores $A = \text{Softmax}(E)$

Take a weighted sum of values Output $= AV$

$$\text{Output} = \text{Softmax}(QK^T)V$$

Self-Attention as a NLP Building Block



This shows a stack of self-attention blocks like we might stack LSTM layers. Self-attention though cannot be a drop-in replacement for recurrence, for it has few issues.

No Notion of Order Self-attention is an operation on sets, meaning it has no inherent notion of order. Self-attention doesn't know the order of its inputs.
We need to encode the order of the sentence in our keys, queries and values. Consider representing each sequence index as a vector: $p_i \in R^d$, for $i \in \{1, \dots, T\}$ are position vectors. It's easy to incorporate this info into our self-attention block: just add the p_i into our inputs. With $\tilde{v}_i, \tilde{q}_i, \tilde{k}_i$ our old values, queries and keys

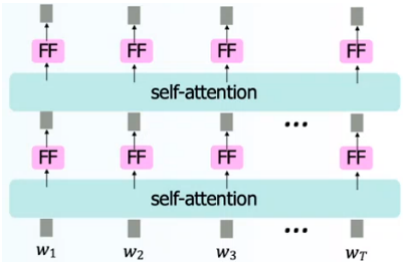
$$v_i = \tilde{v}_i + p_i$$

$$q_i = \tilde{q}_i + p_i$$

$$k_i = \tilde{k}_i + p_i$$

The p_i can be **sinusoidal position representations**: concatenate sinusoidal functions of varying periods. Periodicity indicates that maybe an "absolute position" isn't important, but it's not learnable. So the p_i can be all learnable: learn a matrix $p \in R^{d \times T}$ and the p_is are columns of that matrix. It's flexible, because each position gets to be learned to fit the data, but can't be used to extrapolate indexes outside $1, \dots, T$.

Adding Non-Linearities Therea re no elementwise nonlinearieis in self-attention, so stacking more self-attention layers just re-averages value records. A simple fix: we add feed-forward network to post process each output vector:



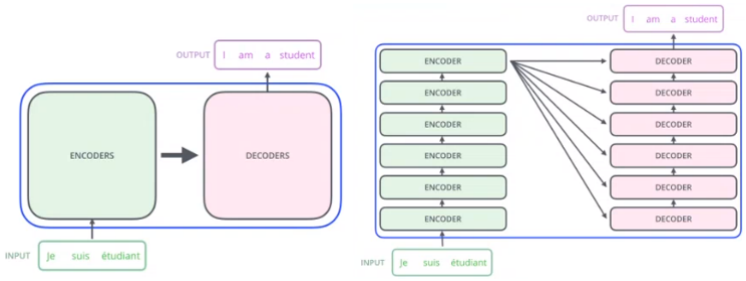
$$m_i = \text{MLP}(\text{Output}_i) + b_2 = W_2 \cdot \text{ReLU}(W_1 \cdot \text{Output}_i + b_1) + b_2$$

Future We need to ensure we don't "look at the future" when predicting a sequence, like in machine translation or language modeling. To mask the future, we could change the set of keys and queries to include only past words but it's inefficient. To enable parallelization, we mask out attention to future words by setting attention scores to $-\infty$

$$e_{ij} = \begin{cases} q^T k_j & j < i \\ -\infty & j \geq i \end{cases}$$

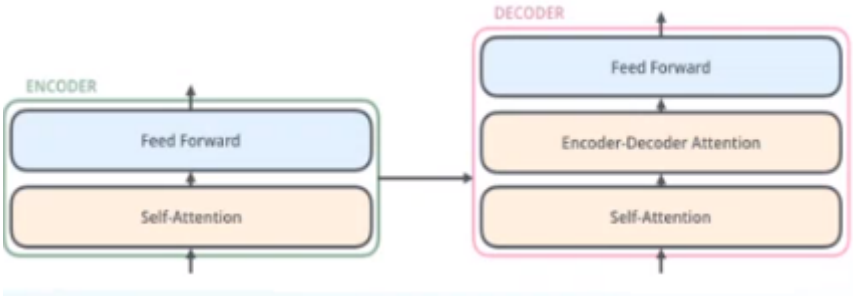
0.13.2 Transformers

The encoder maps an input sequence of symbol representations $(x_1, \dots, x_n)$ to a sequence of continuous representations $(z_1, \dots, z_n)$ Given z, the decoder then generates an output sequence $(y_1, \dots, y_m)$ of symbols one element at a time. At each step, the model is **auto-regressive**, consuming the previously generated symbols as additional input when generating the next.



The encoding component is a stack of encoders, and the decoding component is a stack of decoders of the same number.

Self-Attention The encoder's inputs first flow into a self-attention layer, that helps the encoder look at other words in the input sentence as it encodes a specific word. The outputs of the self-attention layer are fed to a feed-forward neural network: the exact same FFNN in independently applied to each position.
The decoder has both those layers, with an attention layer between them that helps the decoder focus on the relevant parts of the input sentence.



Multi-Headed Attention With a simple self-attention there's only a way for a word to interact with others. We can expand the model ability to focus on different positions: multiple sets of query/key/value weight matrices, apply attention and then concatenate outputs and pipe through linear layer.

1. From the input sentence
2. We embed each word (**word embeddings**)
   In all encoders other than the first we don't need the embedding, we directly use the output of the previous encoder.
3. We split into n heads and multiply with weight matrices
4. Calculate attention using the resulting $Q/K/V$ matrices
5. Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output layer.

Training Tricks

Residual Connections: deep networks are bad at learning the identity function. Therefore, directly passing "raw" embeddings to the next layer can be helpful

$$x_l = F(x_{l-1} + x_{l-1}$$

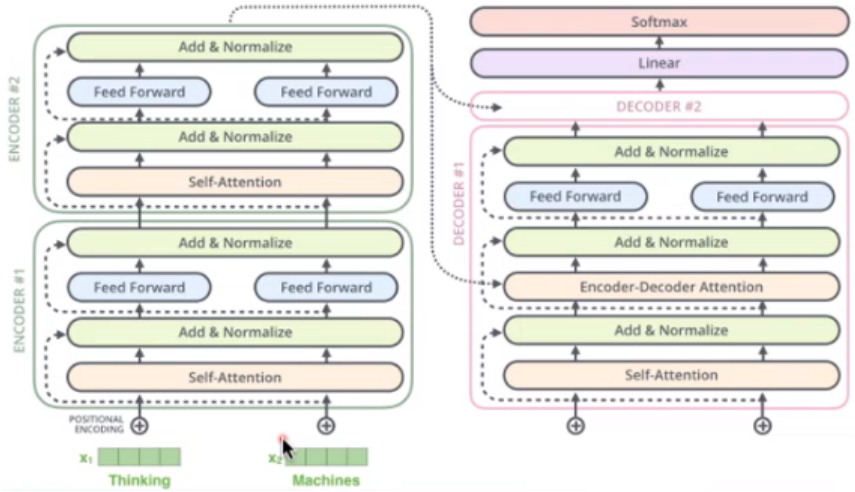
Layer Normalization: may be difficult to train the parameters of a given layer because its input from the layer beneath keeps shifting. Solution: reduce uninformative variation by normalizing to zero mean and std. dev. one within each layer

$$x'_l = \frac{x_l - \mu_l}{\sigma_l + \epsilon}$$

Helps models to train faster.

Scaled Dot Product Attention: after layer normalization, the mean and variance of vector elements is 0 and 1, respectively. However the dot product still tends to take on extreme values, as its variance scales with dimensionality d_k. The updated self-attention equation is

$$\text{Output} = \text{Softmax}\left(\frac{QK^t}{\sqrt{d_k}}\right)V$$



Transformers Library

Hugging Face Transformers Installed with `pip install transformers`.
Typical pattern is to load a model, then initialize two objects:

Tokenizer
Can use specific tokenizer for specific model

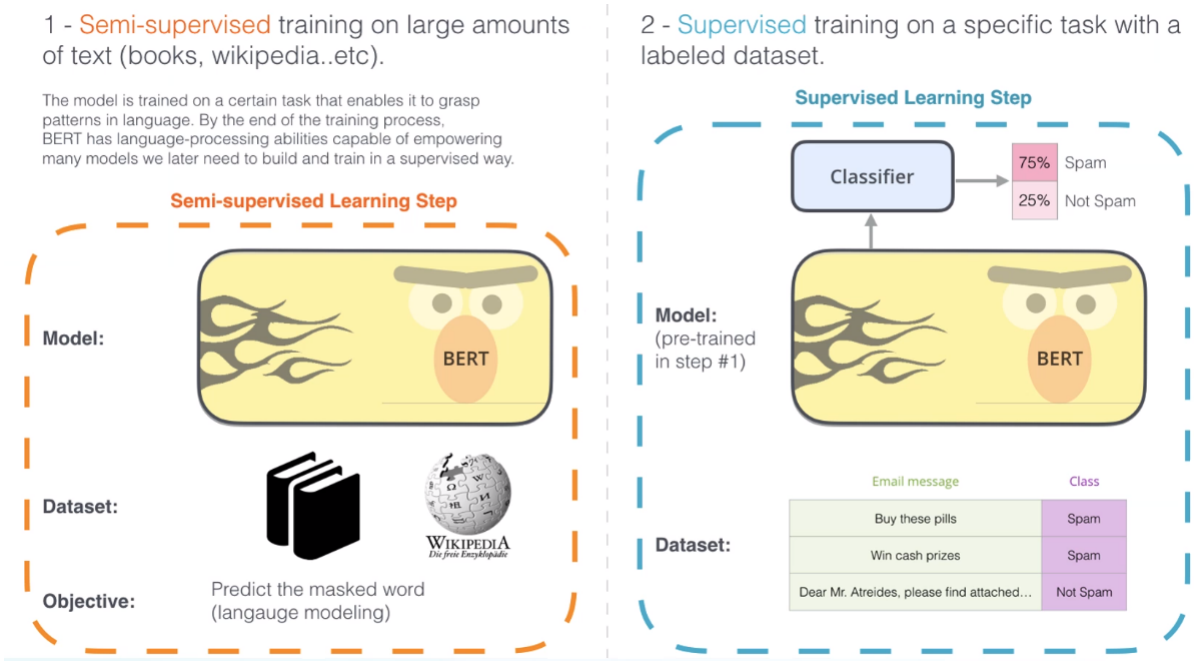
Model, three types: encoders (e.g. BERT), decoders (e.g. GPT2), encoder-decoders

Transformers Architectures

From pretrained word embeddings: start with pretrained word embeddings with no context, and learn to incorporate context in an LSTM or transformer while training on the task.
Issues: the training data we have for out downstream task (like question answering) must be sufficient to teach all contextual aspects of language. Also most of the parameters in our network are randomly initialized.

In modern NLP we pretrain the whole model: all (or almost all) NLP networks are initialized via pretraining. Pretraining methods hide parts of the inputs from the model, and then train the model to reconstruct those parts.
This has been effective at building strong representations of language, parameter initialization for strong NLP models and probability distributions over language that we can sample from.

Pretraining Transformers Two-step development



Pretraining through language modeling Recall the language modelling task: model $P_\theta(w_t \mid w_{1:t-1})$, the probability distributions over words given their past contexts (lots of data for this).
Pretraining through language modeling is training a neural network to perform language modeling on a large amount of texts and **saving the parameters for later**.

Pretraining-Finetuning Paradigm

1. Pretrain on language modeling: learn general things
2. Finetune on your task: adapt to the specific task

SGD

Pretraining a language model provides base parameters $\hat{\theta}$

Finetune a model on a task initializing parameters to $\hat{\theta}$

The training may help because the SGD sticks (relatively) close to $\hat{\theta}$ during finetuning. So, maybe the finetuning local minima near $\hat{\theta}$ tend to generalize well, and/or the gradients of finetuning loss near $\hat{\theta}$ propagate nicely.

Pretraining for Three Types of Architectures

Decoders: language models, nice to generate from, can't condition on future words.

Encoders: bidirectional context so can condition on future (how to pretrain?)

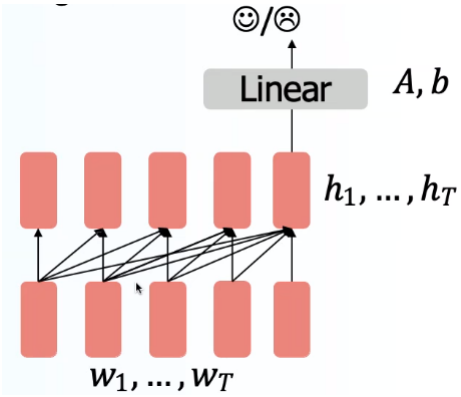
Encoder-Decoders: good parts of both, what's the best way to pretrain them?

Pretraining Decoders When using language model pretrained decoders, we can ignored that they were trained to model $P_\theta(w_t \mid w_{1:t-1})$. We can finetune them by adding a classifier on the last word hidden state

$$h_1, \ldots, h_T = \text{Decoder}(w_1, \ldots, w_T)$$

$$y \simeq Aw_T + b$$

Where A and b are randomly initialized and specified by the downstream task. Gradients backpropagate through the whole network.



Red means pretrained, so note how the linear layer hasn't been pretrained and must be learned from scratch.
It's natural to pretrain decoders as languege models and then using them as generators, finetuning their $P_\theta(w_t|w_{1:t-1})$. This is helpful in tasks where the output is a sequence, with a vocabulaty like that at pretraining time.

$$h_1, \ldots, h_T = \text{Decoder}(w_1, \ldots, w_T)$$

$$w_t \simeq Aw_{t-1} + b$$

With A, b pretrained in the language model.

Generative Pretrained Transformer (GPT): how to format inputs to our decoder for finetuning tasks. E.g. in natural language inference we can label pairs of sentences as entailing/contradictory/neutral.

Pretraining Encoders Encoders get bidirectional context, can't use language modeling. Idea: replace some fraction of words in the input with a mark "[MASK]" and predict these words

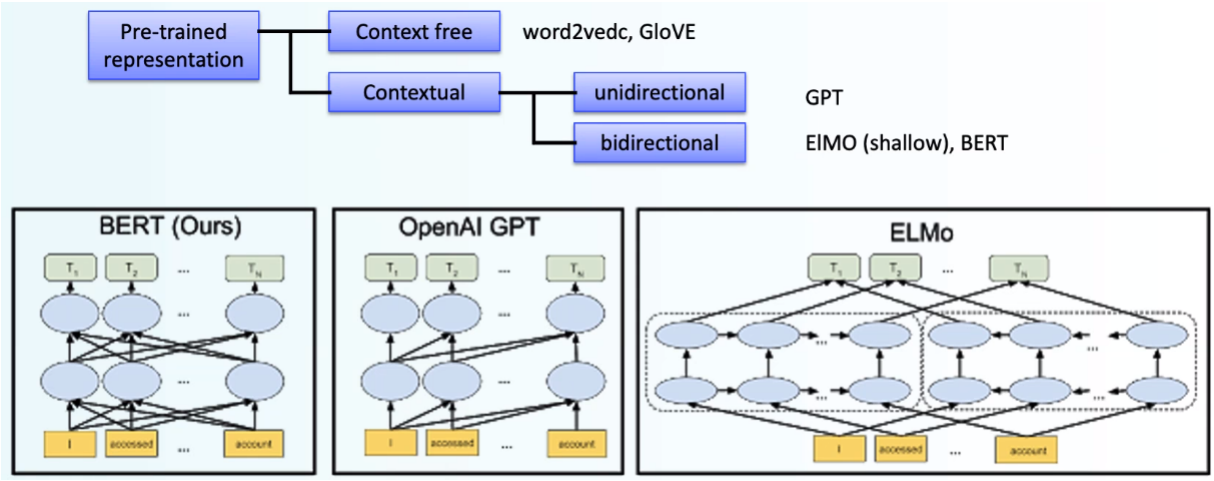
$$h_1, \ldots, h_T = \text{Encoder}(w_1, \ldots, w_T)$$

$$y_i \simeq Aw_i + b$$

Only add loss terms from words that are "masked out". If $\tilde{x}$

Problems with previous methods Language models only use left context **or** right context, but language understanding is bidirectional. Why are language models unidirectional? Directionality is needed to generate a well-formed probability distribution (we don't care about this) and words can "see themselves" in a bidirectional encoder.

BERT This is the first, deeply bidirectional, unsupervised language representation, pre-trained using only a plain-text corpus.
BERT-base is 12 layers, BERT-large is 24 layers, of transformer encoders.



Masked LM Mask out $k\%$ of the words (typically $k = 15$) and predict the masked words. Too little masking: too expensive to train.
Problem: mask token never seen at fine-tuning. Solution: 15% of the words to predict don't replace with [MASK] 100% of the time. Instead: 80% of the time replace with [MASK], 10% replace with random word, and 10% keep the same.
To learn relationships between sentences (**next sentence prediction**), predict whether Sentence B is actual sentence that follows Sentence A, or a random sentence.

Wordpiece models give a good balance between the flexibility of single characters and the efficiency of full words for decoding. Also sidesteps the need for special treatment of unknown words: common words are in the vocabulary, whule other words are built from pieces (e.g. hypathia =

Pretraining Encoder-Decoders For them we could do something like language modeling, but where a prefix of every input ois provided to the encoder and is not predicted

$$h_1, \ldots, h_T = \text{Encoder}(w_1, \ldots, w_T)$$

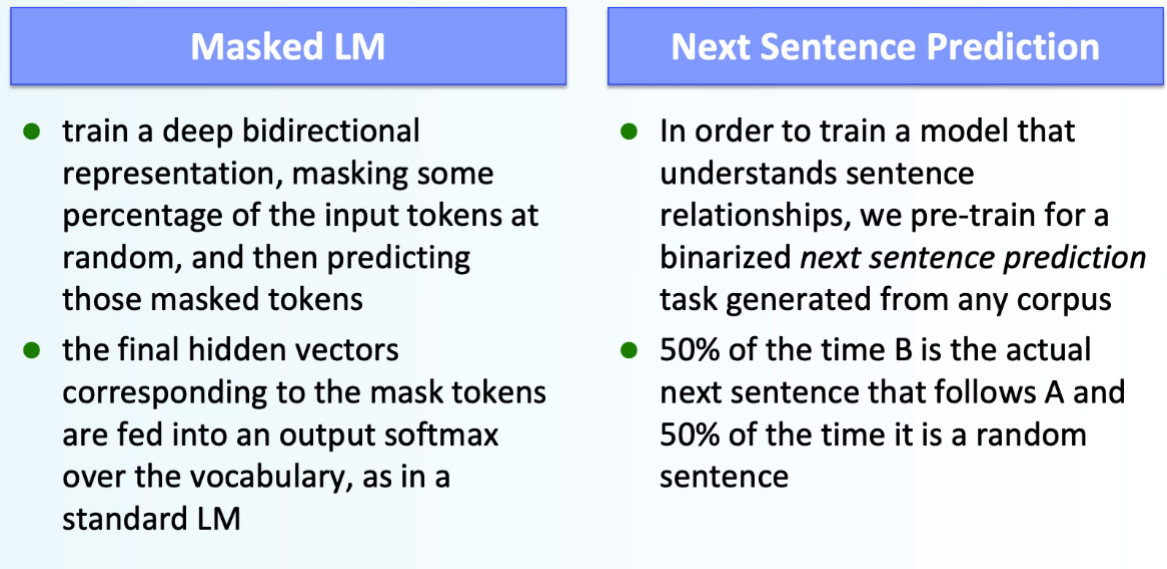
$$h_{T+1}, \ldots, h_2 = \text{Decoder}(w_1, \ldots, w_T, h_1, \ldots, h_T)$$

$$y_i \simeq Aw_i + b \quad i > T$$

The encoder portion benefits from bidirectional context, the decoder portion

What pretraining objective to use? Span corruption.
Replace different-length spans from the input with unique placeholders, then decode out the spans that where removed. This is implemented in text preprocessing. This model, T5, can be finetuned to answer a wide range of questions, retrieving knowledge from its parameters.

Pre-training Tasks



0.14 Analysis of Language Models

Aka BERTology

Questions About Language Models

What can be learned via language model pretraining?

What **can't** be learned via language pretraining?

What will replace the Transformer?

What does deep learning try to do?

What do neural models tell us about language?

How these models affect people and transfer power?

What Linguistic Knowledge is Present in LM? POS tagging through word embedding clusters. NER via Masked Language Model.

Unsupervised NER Given a Masked Language Model, submit a masked sentence, look at possible outputs

LM Effectivness LMs exhibit surprising abilities in several language tasks. But do they really understand language? Consider the Natural Language Inference (NLI) task. What if the model is using simple heuristics to get good accuracy? A diagnostic test set is carefully constructed to test for a specific skill or capacity of your neural model. E.g. HANS (Heuristic Analysis for NLI Systems) tests syntactic heuristics in NLI.

Heuristic	Definition	Example
Lexical overlap	Assume that a premise entails all hypotheses constructed from words in the premise	**The doctor** was **paid** by **the actor**. $\xrightarrow[\text{WRONG}]{}$ The doctor paid the actor.
Subsequence	Assume that a premise entails all of its contiguous subsequences.	The doctor near **the actor danced**. $\xrightarrow[\text{WRONG}]{}$ The actor danced.
Constituent	Assume that a premise entails all complete subtrees in its parse tree.	If **the artist slept**, the actor ran. $\xrightarrow[\text{WRONG}]{}$ The artist slept.

LM as Linguistic Test Subjects How do we understand language behavior in humans? One method: **minimal pairs**, what sounds "okay" to a speaker but doesn't with a small change?
E.g. "*the chef who made the pizzas is here*" vs "*the chef who made the pizzas **are** here*". Idea: verbs agree in number with their subjects, subject-verb relationship.
Assign higher probability to the acceptable sentence in the minimal pair. Just like HANS, we can develop a test set with carefully chosen properties: specifically, can language models handle "**attractors**" in subject-verb agreement?

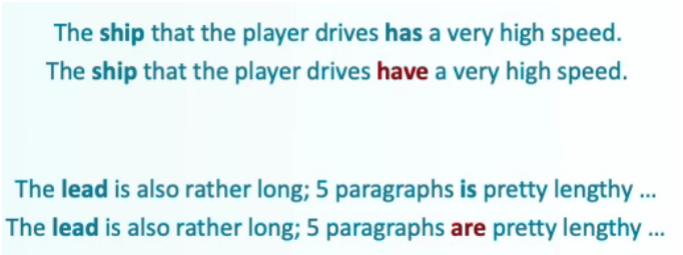
0 attractors: *the chef is here*

1 attractor: *the chef who made the **pizzas** is here*

2 attractors: *the chef who made the **pizzas** and prepped the **ingredients** is here*

...

Some examples for subject-verb agreement with attractors that a model got wrong:



Prediction Explanations What in the input led to this output? For a single example, what parts of the input led to the observed prediction? **Saliency maps**: a score for each input word indicating its importance to the model's prediction.
To make a saliency map, there are many ways to encode the intuition of "importance".

Simple gradient method: for words $x_1, \ldots, x_n$ and the model's score for a given class (output label) $s_c(x_1, \ldots, x_n)$ take the norm of the gradient of the score with respect to each word

$$\text{Salience}(x_i) = \|\nabla_{x_i} s_c(x_1, \ldots, x_n)\|$$

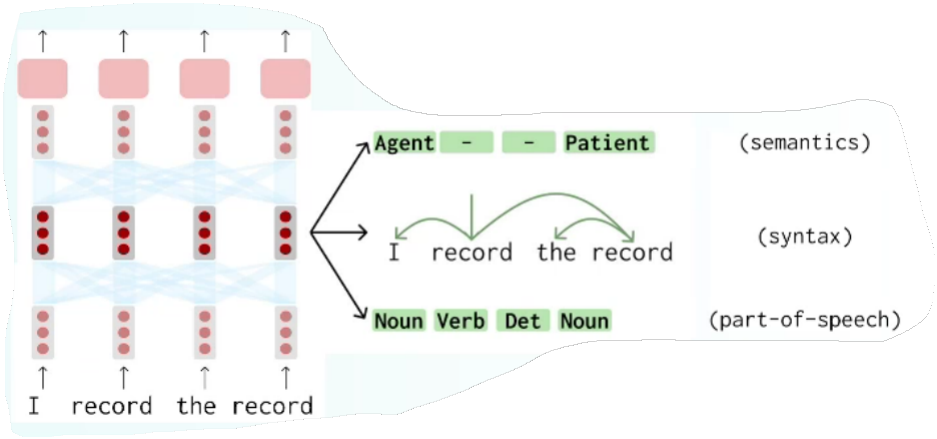
High gradient norm means changing the word locally would affect the score a lot, so it's very important. Not perfect: linear approximation may not hold well. There are many more methods proposed.

Another way is explanation by input reduction, making changes to the document and judge how much it would affect the result. What is the smallest part of the input I could keep and still get the same answer? Idea: run an input saliency method, iteratively removing the less important words.
Another way is analyzing models by breaking them: can we break models by making seemingly innocuous changes to the input?

0.14.1 Probes

Probing Supervised analysis of neural networks.

Premise: pretrained transformers provide surprisingly good general-purpose language representations?
Question: what do pretrained representations encode about lingustic properties which we have annotated data?



We have some property y (like POS). We have the model's word representations at fixed layer $h_1, \ldots, h_T$ where $h_i \in R^d$ where the words are at indices $1, \ldots, T$. We also have a function family F like the set of linear models, or 1-layer FFNN with fixed hyperparameters. We **freeze** the parameters of the model, so its not finetuned, then we train our probe: a function

$$\hat{y} \simeq f(h_i)$$

with $f \in F$.
The extent to which we can predict y from h_i is a measure of the accessibility of that feature in the representation. This helps in gaining a rough understanding into how the model processes its inputs. Also may help in the search for causal mechanisms.

Contextual Representation of Language

Finding a Parse Tree-Encoding Distance Metric Our potentially tree-encodicding distances are parametrixerd by the lkinear transformations $B \in R^{k \times n}$

$$\|h_i - h_j\|^2$$

Finding B B chosen to minimize the difference between true parse tree distances from a hguman-parsed corpis and the predicteds distances from teh fixed word representations transformed by B

$$\min_B \sum_l \frac{1}{|s_l|^2} \sum \left(d(w_i, w_j) - \|B(h_i, h_j)\|^2 \right)$$