Ambiguity

Ambiguity (arguably the more pressing challenge in NLP) is when the language used i can have *multiple different **valid** meanings*. There are many different types of ambigui

- Homophones: *blew* and *blue*
- Word senses: *bank* (finance or river?)
- Part of speech: *chair* (noun or verb?)
- Syntactic structure: *I saw a girl with a telescope*
- Quantifier scope: *Every child loves some movie*
- Multiple: *I saw her duck*
    - (word senses and syntactic)
- Reference: *John dropped the goblet onto the glass table and it broke.*
- Discourse: *The meeting is cancelled. Nicholas isn't coming to the office today*
- Syntactic Ambiguity: *Put the block in the box on the table in the kitchen*

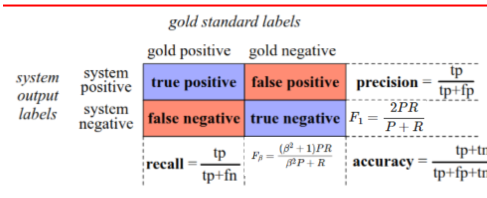
Zipf's Law, $f \times r :: k$

The **Kappa** coefficient measures agreement between two people making category judgements, correcting for expected chance agreement. For example, in a scenario where an item is annotated and 4 coding options are equally likely, then the two annotators will agree 25% of the time. Therefore an agreement of 25% will be assigned K=0 and will scale accordingly (e.g. 50% would be K=0.333 since 50 is a third of the way from 25 to 100).

K-Fold Cross-validation

We often split our dataset into test/train/dev pieces. We *only training the model on a **training** set*. We then can *test the model* on the **test set**. Devsets are used for evaluating different models, debugging, and optimising.

If our model is *too small* to reasonably create sufficiently sized sets, we can use *k-fold cross validation*. This process breaks the data into k pieces and treats one as a held-out set - the remaining are used to train a model. This held out set is used to test these different *folds*. We can then combine all learned information through the use of cross-validation.



MLE

We can estimate discrete probabilities with

$$P_{RF} = \frac{C(x)}{N}$$

where $C(x)$ is the *count* of x and N is the total number of items in the dataset.

3 equally likely events, and one more likely

$$P(a) = 0.7 \quad P(b) = 0.1 \quad P(c) = 0.1 \quad P(d) = 0.1$$

$$\begin{aligned} H(X) &= -0.7\log_2 0.7 - 0.1\log_2 0.1 - 0.1\log_2 0.1 - 0.1\log_2 0.1 \\ &= -0.7\log_2 0.7 - 0.3\log_2 0.1 \\ &= -(0.7)(-0.5146) - (0.3)(-3.3219) \\ &= 0.36020 + 0.99658 \\ &= 1.35678 \end{aligned}$$

N-gram LMs

We want to compute $P(w|h)$, the probability of a word w given a history h. For example, if our history is "*its water is so transparent that*" and we want to know th probability of our next word being *the*, we would be finding

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

A method of calculating this probability would be to use *relative frequency counts*

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

A **bigram** model ($n = 2$), for example, approximates the probability of a word by *only using* the conditional probability of the preceding word; we would approximate the above transparent water example with

$$P(\text{the}|\text{that})$$

This assumption that we can reasonably estimate the probability of a word based on only the prior is called a **Markov** assumption. N-grams make a Markov assumption that we only need to look $n-1$ words into the past. For n-gram size N,

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

However, given the *bigram assumption for the probability* of an individual word, w can compute the probability of a **complete word sequence** by substituting this in our original equation to get

Prod is multiply not add!!

$$P(w_{1:n}) \approx \prod_{k=1}^{n} P(w_k|w_{k-1})$$

How do we get the individual *bigram probabilities* $P(w_k|w_{k-1})$? We can estimate probabilities with MLE.

er a trigram independence assumption, all

$$P(\text{mast}|\text{I spent three years before the})$$
$$P(\text{mast}|\text{I went home before the})$$
$$P(\text{mast}|\text{I saw the sail before the})$$
$$P(\text{mast}|\text{I revised all week before the})$$

stimated as $P(\text{mast}|\text{before the})$. The general rule is

$$P_{MLE}(w_i|w_{i-2}, w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i)}{C(w_{i-2}, w_{i-1})}$$

amount of times the three words have appeared over the amount of times the ious two words appeared).

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

Entropy

Entropy is the *degree of uncertainty* a system has.

The definition of the **entropy** of a random variable X is

$$H(X) = \sum_x -P(x)\log_2 Px$$

It's basically giving the average log, weighted with the probability $P(X)$. Higher values of entropy are present in *more uncertain* systems.

Cross-entropy

For most corpora we cannot exactly calculate Entropy as we do not know the *exact* word sequence probability distribution. We can, however, measure how close *an estimated probability distribution* $\hat{P}$ is to the true distribution P via ***cross-entropy***:

$$H(P, \hat{P}) = \sum_x -P(x)\log_2 \hat{P}(x)$$

But the problem was that we didn't know $P(x)$!
We can approximate this by using *word sequences* $w_1, \ldots, w_n$ (with large n):

$$H_M(w_1, \ldots, w_n) = -\frac{1}{n}\log_2 P_M(w_1, \ldots, w_n)$$

Extrinsic

Extrinsic evaluations are "plugged into" a functional system, and then the system's performance is measured. This is most reliable, but an evaluation method for the system is also needed and the whole process is time consuming.

Intrinsic

Intrinsic evaluations are inherent to the current task, but choosing the correct measure is difficult; ideally, this correlates with extrinsic measures.

Perplexity

Perplexity is $2^{\text{cross-entropy}}$

Problem with using MLE for sentences

If we have a sentence $\vec{w}$ we could say

$$P_{MLE}(\vec{w}) = \frac{C(\vec{w})}{N}$$

The problem with this is that if we ever get a sentence that *has not appeared* in our corpus, $P = 0$; MLE thinks anything *that **hasn't** occurred will **never** occur.*

Laplace Smoothing

Add-one smoothing, sometimes Laplace smoothing, basically pretends every possible word was seen one more time than it actually was. We can't simply just add 1 though - the probabilities won't all sum up to 1, so we need to normalise.

We want:

$$\sum_{w_i \in V} \frac{C(w_{i-2}, w_{i-1}, w_i) + 1}{C(w_{i-2}, w_{i-1}) + x} = 1$$

Solve for x:

$$\begin{aligned} \sum_{w_i \in V}\left(\frac{C(w_{i-2}, w_{i-1}, w_i) + 1}{C(w_{i-2}, w_{i-1}) + x}\right) &= \frac{C(w_{i-2}, w_{i-1}) + x}{C(w_{i-2}, w_{i-1}) + x} \\ \sum_{w_i \in V} C(w_{i-2}, w_{i-1}, w_i) + \sum_{w_i \in V} 1 &= C(w_{i-2}, w_{i-1}) + x \\ C(w_{i-2}, w_{i-1}) + v &= C(w_{i-2}, w_{i-1}) + x \\ v &= x \end{aligned}$$

where v = vocabulary size.

Unfortunately, this normalisation flips our Zipfian curve and now the super low probabilities happen to the common words such as "*it*", "*the*", and "*and*".

We also want to weight these probabilities by a λ value such that the final interpolated probability is between 0 and 1. Generally, we can estimate the probability $\hat{P}$ with

Interpolation is a linear combo!!

$$\hat{P}(w_n|w_{n-2}w_{n-1}) = \lambda_1 P(w_n) + \lambda_2 P(w_n|w_{n-1}) + \lambda_3 P(w_n|w_{n-2}w_{n-1})$$

Interpolation

The values of λ are *hyperparameters*, and are acquired through training. Typically, we hold out a set corpus, and then choose the λ values that **maximise** that corpus, possibly through techniques such as EM.

Lidstone Smoothing

Add-α smoothing, sometimes Lidstone smoothing adds α instead, and this formula comes normalised

$$P_{+\alpha}(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i) + \alpha}{C(w_{i-1} + \alpha v)}$$

This also assumed we know the vocab size in advance, but if we don't we can add a single "unknown" item and use it for all unknown words found during testing. The question on how to choose α is still a bit uncertain; we can use splits and test multiple α values, choosing the one that minimises Cross-entropy on the devset.

Good Turing: 0-count

$$P(\text{unseen}) = \frac{N_1}{n}$$

This uses MLE, we divide this probability equally amongst all unseen events

$$P_{GT} = \frac{1}{N_0}\frac{N_1}{n} \rightarrow c^* = \frac{N_1}{N_0}$$

Good Turing: 1 count

$$P(\text{once before}) = \frac{2N_2}{n}$$

We estimate the probability that the next observation has count of 2 (add-one). We then divide that probability equally amongst all 1-count events

$$P_{GT} = \frac{1}{N_1}\frac{2N_2}{n} \rightarrow c^* = \frac{2N_2}{N_1}$$

Good Turing Smoothing

Our previous methods changed the denominator which had unintended effects even on the frequent events. Good Turing changes the numerator.

If MLE is $P_{MLE} = \frac{c}{n}$, Good Turing uses *adjusted counts* c^* instead: $P_{GT} = \frac{c^*}{n}$

Each probability gets pushed down to the count class.

c	N_c	P_c	$P_c(\text{total})$	c^*	P_c^*	$P_c^*(\text{total})$
0	N_0	0	0	0	$\frac{\frac{N_1}{N_0}}{N}$	$\frac{N_1}{N}$
1	N_1	$\frac{1}{N}$	$\frac{N_1}{N}$	$2\frac{N_2}{N}$	$\frac{2\frac{N_2}{N_1}}{N}$	$\frac{2N_2}{N}$
2	N_2	$\frac{2}{N}$	$\frac{2N_2}{N}$	$3\frac{2N_3}{N}$	$\frac{3\frac{N_3}{N_2}}{N}$	$\frac{3N_3}{N}$

- c: count
- N_c: number of different items with count c
- P_c: MLE estimate of prob. of that item
- $P_c(\text{total})$: MLE total probability mass for all items with that count.
- c^*: Good-Turing smoothed version of the count
- P_c^* and $P_c^*(\text{total})$: Good-Turing versions of P_c and $P_c(\text{total})$

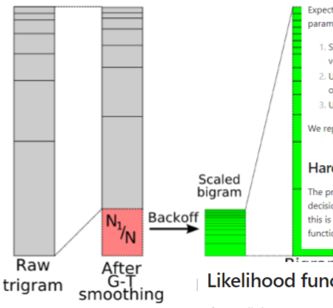
Katz back-off

Back-off is the process of "backing off" (reverting to, switching) to a $n-1$-gram if we have zero count for a given n-gram. In order fo a back-off model to give a correct probability distribution we neec to *discount* the higher order n-grams to save the "probability mass for lower order ones. We do this discounting after Good Turing Smoothing.

$$P(a|b) = \frac{P(b|a)P(a)}{P(b)}$$

$$\hat{w} = \arg\max_{w_\in V} \frac{P(x|w)P(w)}{P(x)}$$

$$\hat{w} = \arg\max_{w_\in V} P(x|w)P(w)$$



Formally, the probability for a back-off n-gram P_{BO} is

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

EM

Expectation Maximisation is the process when we estimate these parameters by making what we observe *maximally* likely.

1. Set all parameters (in our previous cases, matrix values) to arbitrary values (e.g. all costs = 1)
2. Using these parameters, compute optimal values for the variables (in our case, run Edit Distance)
3. Using these alignments, **recompute** the parameters

We repeat steps 2 and 3 until the parameters stop changing.

Hard EM vs True EM

The previous explanation of EM is *hard EM* - there are no "soft/fuzzy" decisions. In *True EM*, we compute the *expected* values of the variables; this is guaranteed to converge to a local optimum of the likelihood function.

Likelihood functions

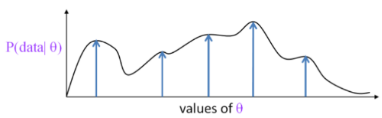
If we call the parameters of our model θ (in our case θ is the set of all character rewrite possibilities $P(x_i|y_i)$, we can compute $P(\text{data}|\theta)$.
If our data contains hand-annotated character alignments, then

$$P(\text{data}|\theta) = \prod_{i=1}^{n} P(x_i|y_i)$$

- If the alignments a are latent, we instead sum of possible alignments

$$P(\text{data}|\theta) = \sum_a \prod_{i=1}^{n} P(x_i|y_i, a)$$

'he likelihood $P(\text{data}|\theta)$ can have multiple local optima.



True EM is guaranteed to converge to one of these, but not guaranteed to find the **global** optimum.

Naive Bayes

If we have a document d and a set of categories C (e.g. spam/not spam, for an email app), we want to assign d to the *most probable* category $\hat{c}$:

$$\hat{c} = \text{argmax}_{c \in C} P(c|d)$$
$$= \text{argmax}_{c \in C} P(d|c)P(c)$$

So we need to define $P(d|c)$ and $P(c)$.

Naive Bayes Classifier

Combining everything, given a document d with features $f_1, f_2, \ldots, f_n$ and a set of categories C, choose the class $\hat{c}$ where

$$\hat{c} = \text{argmax}_{c \in C} P(c)\prod_{i=1}^{n} P(f_i|c)$$

where:

- $P(c)$ is the **prior probability** of class c before observing any data.
- $P(f_i|c)$ is the probability of **seeing** f_i in class c.

Naive Bayes Assumption and modelling $P(d|c)$

We can define a set of *features* (prescence of certain words/sequences, part of speech, etc) to help us classify the documents, and represent each document d as a set of features $f_1, f_2, \ldots, f_3$. We can then model $P(d|c)$ as

$$P(d|c) = P(f_1, f_2, \ldots, f_n|c)$$

Conditionally independent!!

Using a **naive Bayes assumption** (features are conditionally ir we can represent $P(d|c)$ further as

$$P(d|c) \approx P(f_1|c)P(f_2|c)\ldots P(f_n|c)$$

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

$$\hat{P}(f_i|c) = \frac{\text{count}(f_i, c) + \alpha}{\sum_{f \in F} \text{count}(f, c), + \alpha}$$

- $\text{count}(f_i, c)$ is the number of times f_i occurs in class c.
- F is the set of possible features.
- α is the smoothing parameter. This is included as due to our assumption, any "*zero-features*" will result in a flat zero probability for that class (due to multiplying them all together).
    - J&M claims this is often just add-one.

Costs

If we use Naive Bayes with small probabilities we run into *very* small eventual final probabilities. Many implementations solve this by using *costs* (negative log probabilities, summed, solved for the *lowest* cost overall). With this, Naive Bayes often looks like

$$\hat{c} = \text{argmin}_{c \in C} + ((-\log P(c) + \sum_{i=1}^{n} -\log P(f_i|c))$$

Naive Bayes is now a *linear classifier*, because it uses a linear function (in log space) over the input features.

Softmax

The classifier discussed above would only work with two classifiers, as we could set a *boundary* at 0.5. If we wanted more possible classes, we need a *generalisation* of the sigmoid; we want to compute $P(y_k = 1|x)$.

Softmax takes a vector $z = [z_1, z_2, \ldots, z_K]$ of K values and maps them to a probability function.

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{K}\exp(z_j)}, 1 \leq i \leq K$$

Softmax of z is then a vector itself

$$\text{softmax}(\mathbf{z}) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{K}\exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{K}\exp(z_i)}, \ldots, \frac{\exp(z_K)}{\sum_{i=1}^{K}\exp(z_i)}\right]$$

Semi-supervised learning (with EM)

Semi-supervised learning is a method of training the Naive Bayes model on *unlabelled data*, of which there is **much** more of.

We give a model a set of documents with features, classified to specific classes. Using EM, we then get our model to estimate a class and say *how confident* it is of that decision. This then affects *retraining* the model.

Logistic Regression Learning

We require two components to learn the parameters of the model (weights $\mathbf{w}$ and bias b)

- A cost function; a function that measures how close the system output and gold standard output are.
- An optimisation algorithm to iteratively update the weights

Gradient Descent

In general, gradient descent minimises the loss function. L is parameterised by weights ((w, b), but in this example we're gonna denote them with θ). We want to find a set of weights that minimises the loss function

$$\hat{\theta} = \arg\min_\theta \frac{1}{m}\sum_{i=1}^{m} L_{CE}(f(x^{(i)}; \theta), y^{(i)})$$

Gradient descent finds the *gradient of the loss function* at a current point and moves in the *opposite direction*. We can take the derivative to find the value of the slope $\frac{d}{dw}L(f(x; w), y)$.

$$\theta^{\text{new}} = \theta - \alpha\nabla J(\theta)$$

- θ^{new} is our new parameters
- θ is our old parameters
- α is some small weighting constant applied to the next bit
- $\nabla J(\theta)$ is the *vector derivative* of our cost function, when applied to the *old parameters* θ

Cross-entropy loss

We define a loss function L as

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

We first get a loss function that prefers the correct class labels to be *more likely*, called **conditional** maximum likelihood esimation, or CMLE. Given items $x^{(1)}, \ldots, x^{(N)}$ with labels $c^{(1)}, \ldots, c^{(N)}$, choose

$$\hat{w} = \text{argmax}_w \sum_j \log P(c^{(j)}|x^{(j)})$$

If we invert this to obtain the *lowest negative log likelihood loss*, this is called the **cross-entropy loss**. Say we want to maximise the probability of $p(y|x)$; there are two outcomes

$$p(y|x) = \hat{y}^y(1-\hat{y})^{1-\hat{y}}$$

If $y = 1$, this simplifies to $\hat{y}$, and if $y = 0$, to $1 - \hat{y}$.

If we take the log of both sides

$$\log p(y|x) = y\log\hat{y} + (1-y)\log(1-\hat{y})$$

To obtain cross-entropy loss L_{CE}, we just flip the sign:

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

and knowing $\hat{y} = \sigma(w \cdot x + b)$:

$$L_{CE} = -[y\log\sigma(w \cdot x + b) + (1-y)\log(1 - \sigma(w \cdot x + b))]$$

MaxEnt

Maximum Entropy models are best suited if we have a *lot* of feautres, and the features do not follow the assumption of conditional independence. It is similar to Naive Bayes in that

$$\hat{c} = \arg\max_{c \in C} P(c|x)$$

but we model $P(c|x)$ *directly*. This is done by, given observations $\vec{x}$ and a class c,

$$P(c|\vec{x}) = \frac{1}{Z}\exp(\sum_i w_i f_i(\vec{x}, c))$$

where the normalisation constant $Z = \sum_{c'}\exp(\sum_i w_i f_i(\vec{x}, c'))$

We train MaxEnt like any other form of Logistic Regression Learning.

Methods of combining stems and affixes

'here are four methods to combine **stems** and **affixes**:

- Inflection
    - stem + grammar affix (**no change** to grammatical category)
    - *walk -> walking*
- Derivation
    - stem + grammar affix (**change** to grammatical category)
    - *combine -> combination*
- Compounding
    - stems together
    - *doghouse*
- Cliticization
    - *I've, we've, he's*

Parsing and generation

When we discuss *parsing*, we're going from the surface to lexical form e.g. *foxes -> fox + N + PL*.

Generation goes the opposite way; *fox + N + PL-> foxes*.

We often want to produce an intermediate form corresponding to analysis in terms of *morphemes* (minimal meaningful units) before applying *orthological rules*. For our *foxes* example, we may produce "*fox ^ s #*", where morpheme boundary = ^ and # = word boundary.

Phrase Type	Abbreviation
Noun Phrase	NP
Verb Phrase	VP
Adjective Phrase	AP
Adverb Phrase	AdvP
Prepositional Phrase	PP

Finite State Transducers

Formally, a *finite state transducer* T with inputs from $\sum$ and outputs from $\prod$ consists of:

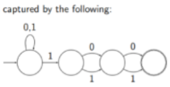
- states Q, S (for start), F (for finish)
- a transition relation $\Delta \subseteq Q \times (\sum \cup \{\epsilon\}) \times (\prod \cup \{\epsilon\}) \times Q$
    - This defines a **many-step transition relation** $\hat{\Delta} \subseteq Q \times \sum^* \times \prod^* \times Q$

FSAs

An *epsilon NFA* (ϵ-NFA) allows an input (in parsing) or output (in generation) defined by an arc to be the empty string.

A *nondeterministic finite state automaton (NFA)* is a finite state machine where a state can have more than one outgoing arc.

E.g., $(0|1)^*1(0|1)^2$ is captured by the following:



An *epsilon NFA* (ϵ-NFA) allows an input (in parsing) or output (in

HMMs

Formalising the tagging problem

To find the best tag sequence T for *untagged sentence* S

$$\underset{T}{\operatorname{argmax}}\, p(T|S)$$

We can use Bayes' rule to give us

$$P(T|S) = \frac{p(S|T)p(T)}{p(S)}$$

But if we only care about argmax_T we can drop $p(S)$:

$$\underset{T}{\operatorname{argmax}}\, p(T|S) = \underset{T}{\operatorname{argmax}}\, p(S|T)p(T)$$

$P(T)$ is the state transition sequence:

$$P(T) = \prod_i P(t_i|t_{i-1})$$

$P(S|T)$ are the emission probabilities:

$$P(S|T) = \prod_i P(w_i|t_i)$$

For any specific tag sequence T we can compute $P(S,T) = P(S|T)P(T)$

$$P(S|T)P(T) = \prod_i P(w_i|t_i)P(t_i|t_{i-1})$$

Markov Assumption

The **Markov Assumption** we're making here is that each tag is only dependent on the previous one (*bigram*) and that words are independent given tags.

Other previous models have had *markov assumptions* such as N-grams.

HMMs

For POS tagging, our *"generative"* **Hidden Markov Model** will do three things:

- **Model:** parameterised model of how both words and tags are generated $P(x, y|\theta)$ *(the transition and emission probabilities)*
- **Learning:** use a labelled training set to estimate most likely parameters of the model $\hat{\theta}$
- **Decoding:** $\hat{y} = \operatorname{argmax}_y P(x, y|\theta)$ (Viterbi)

Viterbi

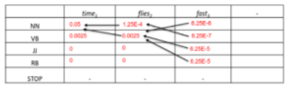
1. We initialise a new matrix, filling the first column as follows: $v_i^1 = a_{(\text{START},i)} b_{(i,x^1)}$ where a is the *transition probabilities* matrix and b is the *emission probabilities* matrix

a_{ij}	STOP	NN	VB	JJ	RB
START	0	0.5	0.25	0.25	0
NN	0.25	0.25	0.5	0	0
VB	0.25	0.25	0	0.25	0.25
JJ	0	0.75	0	0.25	0
RB	0.5	0.25	0	0.25	0

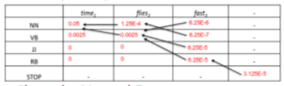
b_{ik}	time	flies	fast	...	...	...
NN	0.1	0.01	0.01	...	...	...
VB	0.01	0.1	0.01	...	...	...
JJ	0	0	0.1	...	...	...
RB	0	0	0.1	...	...	...

	$time_1$	$flies_2$	$fast_3$	-
NN	0.5x0.1=0.05			
VB	0.25x0.01=0.0025			
JJ	0			
RB	0			
STOP	-	-	-	

2. The next column can now be filled in, according to the following: $v_j^t = (\max_i (v_i^{t-1} a(i,j))) b(j, x_t)$, where $j \in [1, 2, \ldots, N]$ and $t \in [2, \ldots, |x|]$. Back-pointers should be tracked.
3. Repeat step 2 for each subsequent value of t, unless you've reached the end of the sentence ($t = |x|$) in which case proceed to step 4.



4. Fill the final cell according to the following: $v_{\text{STOP}}^{|x|+1} = \max_i \left(v_i^{|x|} a_{i,\text{STOP}}\right)$



Syntax and Parsing

Constituency tests

We need methods of testing whether a group of words form a *constituent*: that is to say, they should all be considered one object(/token?) in the scope of the sentence as a whole

1. The lecture that that the professor prepared was absolutely brilliant
   - Substitution
     - It was absolutely brilliant
   - (wh-)movement
     - What was absolutely brilliant? The lecture that that the professor prepared
   - Coordination
     - The lecture that the professor prepared **and the slideshow** was absolutely brilliant
   - Clefting
     - *It was* the lecture that the professor prepared *that was* absolutely brilliant

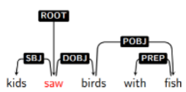
really youll only need:
nsubj - nominal/noun subject
dobj - direct object
amod - adjective modifier
advmod - adverbial modifier
det - determiner
conj - conjunction 14:39

this message is getting snipping tooled onto the cheatsheet 14:42

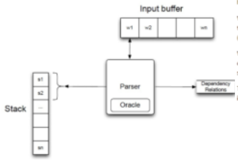
Dependency Parsing

Edge Labels

We can further annotate these **relations** (*edges*) with *labels*



Transition-based dependency parsing



There are three actions we can do with each word:

- **LeftArc:** Assign head-dependent relation between $s1$ and $s2$, pop $s2$
- **RightArc:** Assign head-dependent relation between $s2$ and $s1$, pop $s1$
- **Shift:** Put $w1$ at the top of the stack

This is sometimes called the *shift-reduce* algorithm; an alternative to CYK that doesn't even require a grammar.

Step	Stack	Word List	Action	Relations
0	[root]	[Kim,saw,Sandy]	Shift	
1	[root,Kim]	[saw,Sandy]	Shift	
2	[root,Kim,saw]	[Sandy]	LeftArc	nsubj(saw,Kim)
3	[root,saw]	[Sandy]	Shift	
4	[root,saw,Sandy]	[]	RightArc	dobj(saw,Sandy)
5	[root,saw]	[]	RightArc	root→saw
6	[root]	[]		

Compositional Semantics

A goal of NLP is to understand what people *mean* when they talk. But how do we know if we've succeeded?

We can use **semantics** to relate expressions to "the world", including both their *truth conditions and denotation* (literal meaning) and *connotation* (other associations).

We can exploit **compositionality** to augment a grammar with a *semantic component* which will deterministically derive the *logical* form of a sentence from its syntax tree.

This representation should be **unambiguous**, support **automated inference**, and be **verifiable, somehow.**

Context Free Grammars

CFGs, or **Context Free Grammars** are a way to represent grammar. Formally, a **CFG** is a tuple of 4 elements $G = (V, \sum, R, S)$:

- V - the set of *non-terminals.*
- $\sum$ - the set of *terminals*
- R - *the set of rules* in the form $X \to Y_1, Y_2, \ldots, Y_n$ where $n \geq 0$
  - $X \in Y, Y_i \in V \cup \sum$
- S is a dedicated start symbol

The term "context-free" is due to a subtree only being affected by what the parent's other child is, but **not the context**.

Chomsky Normal Form

A **CNF** grammar only contains rules in one of two forms:

$$C \to x$$
$$C \to C_1 C_2$$

That is to say, each rule expands to either a POS tag or *two inner rules.*

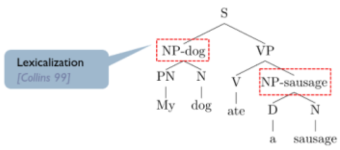
Converting to CNF

Any CFG can be converted to an equivalent CNF - the syntactic tree will look different, but the language remains the same. There are three steps to get a valid CNF grammar:

1. Remove any empty (*epsilon*) productions ($C \to \epsilon$).
2. Get rid of any *unary* rules ($C \to C_1$).
3. Split rules so we get binary rules ($C \to C_1\, C_2 \ldots C_n$ ($n > 2$)).

Structural Annotation

Regular PCFGs (*treebank*) do not produce the best parsers because *they do not encode anything more beyond single rules.* To extend this, we (ironically, for Context Free Grammars) need to incorporate some form of context, usually into the parents. This is known as *lexicalisation.*



Vertical & Horizontal Markovisation

A form of lexicalisation, in which each non-terminal in the tree is annotated with its lexical head. It also solves the problem of *close attachment*, which from what I can tell is when `PP` attaches to the closest preceding `NP` (it solves it by distinguishing `NP` s).

Vertical Markovisation increases context, whereas Horizontal Markovisation (a form of *binarisation*) tries to reduce context. We can combine different orders of **both** vertical and horizontal markovisation to best optimise the model's performance

CYK for parsing

We can use CYK to parse a sentence.

Given a:

- grammar $G = (V, \sum, R, S)$
- sequence S of words $(w) = (w_1, w_2, \ldots, w_n)$
  produce a *parse tree* for w.

If we imagine "fence posts" between each word and one on each end, labelled starting from 0, we can use `span(i, k)` to refer to words between fence posts i and j.

The general idea is:

1. We compute for every `span` a set of admissible labels
   - This starts from small trees (i.e single words) and proceeds to larger ones
2. When we're done, we check if S is among the admissible labels for the whole sentence
   - That is to say, if a tree with the signature $[0, n, S]$ exists
3. If yes, *the sentence belongs to the language.*

This does not always give us direct clear meaning. If we don't have absolute values (one possible label for each word), then we have an ambiguous sentence

Lambda-calculus and Beta reduction

Using lambda calculus allows us to work with "partially constructed formulae". If φ is a *well-formed* FoL expression and x is a variable, then $\lambda x \varphi$ is a well formed FoL expression (a function known as the λ-*term*).

$$\lambda x \varphi(a) = \varphi[x/a]$$

When we apply this function this is known as ***Beta*** **(β) *reduction***. For example:

$$\lambda y \lambda x(\exists e(\text{eat}(e, x, y) \wedge e \prec n))(rice) \text{ becomes}$$

$$\lambda y \lambda x(\exists e(\text{eat}(e, x, rice) \wedge e \prec n))$$

In short, if we have $\lambda x.\,(x(a\,b\,x))\,n$, beta reducing this replaces all instances of x in $x(a\,b\,x)$ with n.

S → NP VP *VP.Sem(NP.Sem)* **(Sentences)**
NP → MassN *MassN.Sem* | PropN *PropN.Sem* **(Noun phrases)**
VP → Vi *Vi.Sem* | Vt NP *Vt.Sem(NP.Sem)* **(Verb phrases)**
PropN → Fred *fred* | Jo *jo*. . . **(Proper nouns)**
MassN → rice *rice* | wood *wood* . . . **(Mass nouns)**
Vi → talked $\lambda x \exists e(talk(e, x) \wedge e \prec n)$ | . . . **(Intransitive verbs)**
Vt → ate $\lambda y \lambda x.\exists e(eat(e, x, y) \wedge e \prec n)$ | . . . **(Transitive verbs)**

SDRT and Logical Form

Segmented Discourse Representation theory is the **logical form** of monologue. The logical form consists of:

- Set A of labels $\pi_1, \pi_2, \ldots$
  - each label stands for a discourse segment
- A mapping $\mathcal{F}$ from each label to a formula representing its content
- Vocabulary contains coherence relations e.g. *Elaboration*(π_1, π_2)

Example, using John's safe

π_1: John can open Bill's safe
π_2: He knows the combination

π_0: *Explanation*(π_1, π_2)
π_1: ιx(*safe*(x) & *possess*(x, *bill*) & *can*(*open*(e_1, *john*, x)))
π_2: ιy(*combination*(y) & **of**(y, **x**) & *knows*(**john**, y))

Lexical Semantics

Hyponyms and Hypernyms

Words can be *hyponyms* or hypernyms

- A **hyponym** is a subset of another word
- A **hypernym** is a *superset* of another word
  An **A-B** relationship of these words is called an *ontology.*

Regular Polysemy

Polysemy is when two separate word senses are in some way related to each other, we can capture patterns. **Regular** polysemy is when two words exhibit polysemy for the same reason (e.g. *Cherry* and *Orange* both meaning a **colour** and **fruit**).

Discourse Coherence

Coherence

Adding context to events can change our minds on how we interpret then, even if the original context is still present.

Coherence is important (to associating otherwise unrelated sentences together), but it can be challenging to represent it computationally.

The following examples showcase some challenging problems a computer may be asked to solve

Time

John hit Max on the back of his neck.
Max fell. John pushed him.
Max rolled over the edge of the cliff.

Word meanings

A: Did you buy the apartment?
B: Yes, but we rented it./ No, but we rented it.

Bridging

John took an engine from Avon to Dansville.
He picked up a boxcar./He also took a boxcar.

Dishonesty

a. M (to K and S): Karen 'n' I're having a fight,
b. M (to K and S): after she went out with Keith and not me.
c. K (to M and S): Wul Mark, you never asked me out.

a. P: Do you have any bank accounts in Swiss banks, Mr. Bronston?
b. B: No, sir.
c. P: Have you ever?
d. B: The company had an account there for about six months, in Zurich.

Distributional Semantics

Distributional hypothesis

The idea that words in *similar contexts* imply *similar meaning*

First and second order co-occurence

There are two types of *co-occurence* between words:

First-order co-occurence

- *Syntagmatic association*
- Typically near each other; *wrote* is first-order associate of *book*

Second-order co-occurence

- *Paradigmatic association*
- Words that have **similar neighbours**; *wrote* is a second-order associate of *said* and *remarked*

PMI

Pointwise mutual information tells us how likely the co-occurence is than if the words were independent.

$$\text{PMI}(x, y) = \log_2 \frac{P(x, y)}{P(x)P(y)}$$

$P(x, y)$ is the probability of seeing the words x and y in the same context.
$P(x)P(y)$ is the *predicted* probability of the same **if** x **and** y **are independent.**

Vector space representation of context vectors

We could also take the *dot product* of two vectors $\vec{v}$ and $\vec{y}$.

use *Euclidean Distance*

$$\sqrt{(\sum (v_i - w_i)^2)}$$

$$\vec{v} \cdot \vec{w} = \sum_i v_i w_i$$

This also falls prey to the frequency issue mentioned above, but if we *normalise the dot product* we solve this. Normalising the dot product like this is the same as finding the *cosine* of the angle between the vectors

Latent Semantic Analysis

LSA is a technique which analyses a document to infer meaning *from contexts*. We apply a *sliding context window* to the document. For sake of example, let's say we use a window of $L = 2$ on the following sentence:

> I saw a cute grey cat playing in the garden

If our **central** word was "cat", our **context** words would be "cute", "grey", "playing", and "in"

> I saw a *cute grey* ***cat*** *playing in* the garden

LSA then forms a $N(w, c)$ matrix, where each cell tells us the number of times *word* w appeared in *context* c. We calculate this value by with two sets of vectors: *word (central) vectors* and *context vectors*; when a word is the central one, we use the value from that, and when it is a context word, we use that. Formally, the value of each element is

$$\text{tf}(w, d) \cdot \text{idf}(w, D)$$

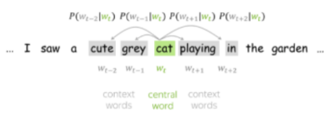
where

- we are focusing on word w
- we are focusing on document d, from a collection D
- tf is *term frequency* ($N(w, d)$); the amount of times the word w appears in the document d
- idf is the *inter-document frequency*

$$\log \frac{|D|}{|d \in D : w \in d|}$$

This isn't very good unless we use PMI to calculate our co-occurences.

Skip-gram model



Negative-Log-Likelihood Cost Function

For each position $t = 1, \ldots, T$ in a corpus, *Word2Vec* predicts the likelihood (the context words) within an m sized window given a centra word w_t:

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

θ represents all variables to be optimised. The **actual cost function** (objective function) $J(\theta)$ is the *average* of this negative-log-likelihood

$$\textbf{Loss} = J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \leq j \leq m, j \neq 0} \log P(w_{t+j}|w_t, \theta)$$

We can bring back our old friend Softmax to compute the probability $P(o|c)$ for a *central* word c and a *context* (outside) word o:

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

The u^T here just means the transpose of the vector, turning it into a row vector so it can be *dot producted* with v_c

Optimisation with Gradient Descent

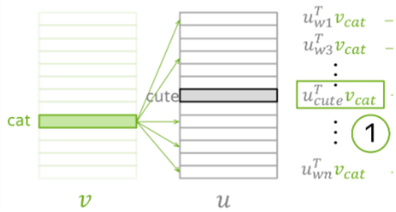
We want to train the parameters (θ) v_w and u_w for all w words in the vocabulary.

Each update is for a single pair of words - one *center word* and one of its *context words*. Say with our previous example, our *central word* is "cat", and our *context word* is "cute", our loss term becomes

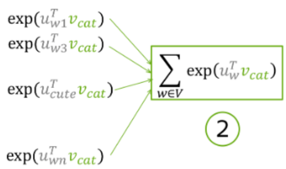
$$J_{t,j}(\theta) = -\log P(\text{cute}|\text{cat})$$
$$= -\log \frac{\exp(u_{\text{cute}}^T v_{\text{cat}})}{\sum_{w \in Voc} \exp u_w^T v_{\text{cat}}}$$
$$= -u_{\text{cute}}^T + \log \sum_{w \in Voc} \exp u_w^T v_{\text{cat}}$$

- We only use
  - from vectors for **central words**
  - v_{cat}
  - from vectors for **context words**
  - all u_w (for all words in $Vocab$)

1. Take the *dot product* of v_{cat} with all u.



2. exp and $\sum$ (sum) all.



3. Get the loss for this step.

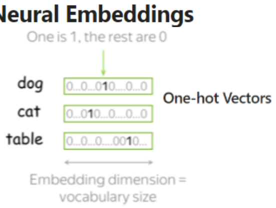
$$J_{t,j}(\theta) = -u_{\text{cute}}^T v_{\text{cat}} + \log \sum_{w \in V} \exp(u_W^T v_{\text{cat}})$$

4. Evaluate the gradient and make an update

$$v_{\text{cat}} := v_{\text{cat}} - \alpha \frac{\partial J_{t,j}(\theta)}{\partial v_{\text{cat}}}$$
$$u_w := u_w - \alpha \frac{\partial J_{t,j}(\theta)}{\partial v_{u_w}} \forall w \in V$$

(∂ is talking about *partial derivatives* here. It's so we know "which way to go" to minimise the cost of J)

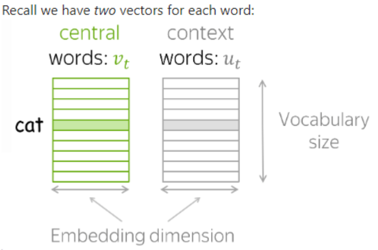
Neural Embeddings

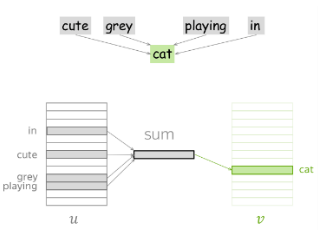


This grows with the size of the vocabulary, which isn't very ideal, but also this method doesn't actually capture any semantic meaning! We need better measures, as our model will be useless at predicting words if it doesn't know what they mean.

Computing $P(w_{t+j}|w_t, \theta)$

Recall we have *two* vectors for each word:





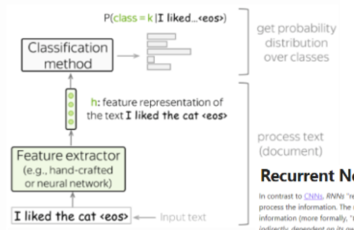
CBOW: from sum of context predict central

Continuous Bag of Words model

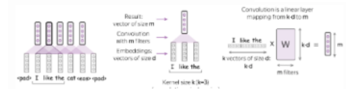
The previously mentioned Skip-gram model has been the basis for Word2Vec; it predicts *context words* given a *central word*. The *Continuous Bag of Words* (**CBOW**) model predicts *a central word* given *context words*, by summing the context vectors.

Neural Classifiers

We can use our networks to *classify* a document of text. We first use a feature extractor (which can be hand crafted, or also learned with a *Neural Network*) to obtain a *feature representation* of the text, and then can use a classification method to get a probability distribution over possible classes $P(\text{class} = k|\text{input text})$.



Convolution is a Linear Operation Applied to Each Window



Formally,

- $(x_1, \dots, x_n)$ are representations of the input words $x_i \in \mathbb{R}^d$
- d is the size of an input embedding
- k is the *kernel size* - the length of a convolution window
- m is the output channels, the number of convolution filters (produced by the CNN).

  A convolution is a linear layer $W \in \mathbb{R}^{(k \cdot d) \times m}$. For a k sized window, it takes the concatenation of the vectors

$$u_i = [x_i, \dots, x_{i+k-1}] \in \mathbb{R}^{k \cdot d}$$

and multiplies by the convolution matrix

$$F_i = u_i \times W$$

Classification techniques

A typical classification technique for this involves Logistic Regression. Remember we are interested in finding out $P(y = k|x)$. The pipeline here goes as follows:

1. Get the feature representation $h = (f_1, f_2, \dots, f_n)$
2. Take $w^i = w_1^i, \dots, w_n^i$ - vectors with feature weights for each of the classes
3. For each class, take the dot product of the feature representation h with feature weights $w^{(k)}$:

$$w^{(k)}h = w_1^{(k)} \cdot f_1 + \dots + w_n^{(k)} \cdot f_n$$

4. Use Softmax to get class probabilities

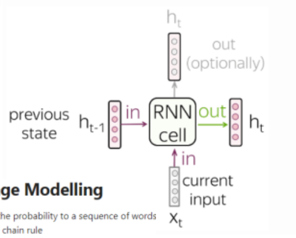
$$P(\text{class} = k|h) = \frac{\exp(w^{(k)}h)}{\sum_{i=1}^{K} \exp(w^{(i)}h)}$$

Recurrent Neural Networks

In contrast to CNNs, *RNNs* "read" a sequence of tokens one by one and process the information. The main feature of RNNs is that it **remembers** information (more formally, "*the value of some unit is directly, or indirectly, dependent on its own earlier outputs as an input*"). This one-by-one nature makes these models particularly easy to adapt for generation.

Anatomy of an RNN cell

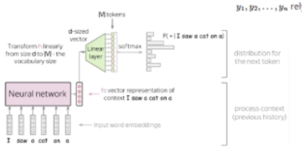
Each individual cell of an RNN receives a new input vector (such as a token embedding) and the *previous network state* (which hopefully encodes all the previous information). It then produces an output from this:



Neural Language Modelling

A *language model* assigns the probability to a sequence of words $y_1, y_2, \dots, y_n$ relying on the chain rule

$$P(y_1, y_2, \dots, y_n) = \prod_{t=1}^{n} P(y_t|y_{<t})$$



Ngram language models

- relies on a short prefix, to get a distribution over next tokens
- explicit independence assumption (can't use context outside of the ngram window)
- smoothing is necessary

RNN language models

- 'compresses' the past into a state, used to compute the distribution over next tokens
- no independence assumptions; the gradient descent learns to compress the past
- all the information is carried through hidden states (hard to carry it across long distances)

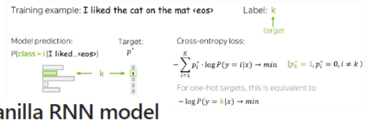
Representing documents as vectors

We can continously train a Neural Network to *create **it's own** classes*, and obtain it's *own* feature representation. Ideally, we'd want a NN to be able to show if two documents are in the same class, due to their document vectors being close to each other.

What NN learns (hopefully):

Text vectors point in the direction of the corresponding class vectors

The standard loss function for training such a model is Cross-entropy loss.



Vanilla RNN model

By combining these cells alongside token embeddings, we can iterate through all tokens and produce a final result. To get from previous state h_{t-1} to h_t with input x_t, we perform

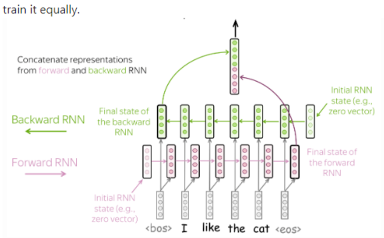
$$h_t = \tanh(h_{t-1}W_h + x_tW_x)$$

Here, $\tanh()$ is our *activation function* ($\mathbf{ReLU}$ is often used instead too), and the W represent *hyperparameters* (can be optimised through training).

However, if we just "read the last state", it might not be very good for classifications that require a bit more thinking

Bidirectional RNN

If we run two RNNs going in opposite directions as one layer, we can train it equally.



This can make it forget the *structure* of the document, and it can be hard to parse in retrospect.

CNNs

Convolutional Neural Networks have **translational invariance**; if we wanted to find **if an image contains a cat**, and didn't care **where the cat was**, we could use a CNN and process all images **containing cats** **somewhere equally**. We can use a CNN for text in certain contexts. For example, if a feature is *very informative*, sometimes we don't care **where** in a text it appears - just as long as it **does appear**.

<pad> ***I like the*** cat on a mat <eos> <pad>

<pad> I ***like the cat*** on a mat <eos> <pad>

<pad> I like ***the cat on*** a mat <eos> <pad>

<pad> I like the ***cat on a mat*** <eos> <pad>

Pooling

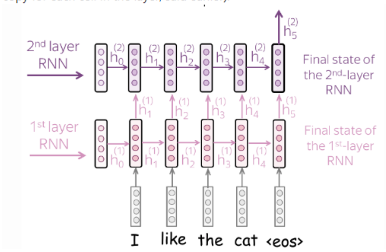
There are two types of pooling used:

- **Max** pooling takes the maximum over each dimension, so we get the maximum values for each feature from all the filters
- **Mean** pooling works exactly the same but computes the mean of each feature from all the filters.

Pooling goes in *strides*, so we have multiple pools for different windows of text. We apply a *convolution* to these individual features and then can use *global pooling* (max) to average a final vector of feature strengths for the whole network.

Multi-layer RNN

If we stack more layers, piping the final result of layer $n-1$ as the *input* (x_i) to the final cell in layer n, then inputs for the higher RNN are representations coming from previous layers (we also do this using the copy for each cell in the layer, said earlier).



Residual connections

When stacking multiple layers, the gradients don't propagate as well. If we add a block's input to it's output, this solves the problem. This is known as a *residual connection*. If we apply a gated sum (the gate $g = \sigma(Wx + b)$) to the input x and output h, and then combine, this is known as a *highway connection*.

Text Generation and Encoder-Decoder Models

While the immediate thought might be straight to *ChatGPT* (and we will get onto Transformers), most topics in text-generation cover the field of *machine translation*, and this is where most research and write-up has been.

Greedy Decoding

We could theoretically just sample the most probable word generated by our Neural Language Model (called *greedy decoding*). This can result in *boring, common* words **always** getting chosen.

We can slightly control this with *temperature*.

$$\frac{\exp(h^Tw)}{\sum_{w_i \in V} \exp(h^Tw)} \qquad \frac{\exp(\frac{h^Tw}{\tau})}{\sum_{w_i \in V} \exp(\frac{h^Tw}{\tau})}$$

The higher the temperature, the *flatter* the probability distribution. When sampling with Softmax, this results in more *diverse* samples

Encoder-decoder framework

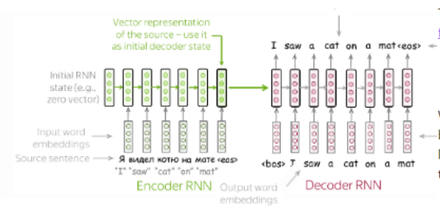
A form of *sequence-to-sequence modelling*, encoder-decoders consists of two (obvious) parts. The **encoder** takes in a *source sentence* and converts it to a *representation of that sentence* to be *fed into the **decoder** part*. The **decoder** then takes this *representation* and produces a *target sentence*.

Conditional Language Modelling

Seq2Seq tasks can be modelled as *Conditional Language Models*; the difference from a regular *language model* now being a *condition on source x*.

$$P(y_1, y_2, \dots, y_n|x) = \prod_{t=1}^{n} p(y_t|y_{<t}, x)$$

RNNs for encoder-decoder

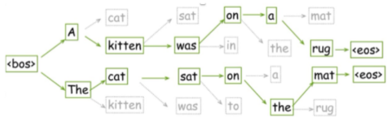


Beam Search

The probability of a sentence using the Recurrent Neural Networks RNNs for encoder-decoder framework is now

$$y' = \arg\max_y p(y|x) = \arg\max_y \prod_{t=1}^{n} p(y_t|y_{<t}, x)$$

We now need to find $\arg\max$. We can use Greedy Decoding from earlier, but another idea is to keep track of *possible decodings*, and then pick the likely ones as we iterate through the tree of possibilities generated from the decoder.

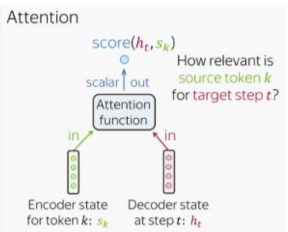


Usually, the beam size is 4-10. Increasing beam size is kinda against the point - it'll end up close to greedy decoding while being slower.

Attention

If we are forced to encode rich information into a limited size vector, we will inevitably miss something - and it could be important!

Neural Attention Model



At each decoder step, attention

- receives attention input
  - a decoder state h_t and *all encoder states* $s_1, s_2, \dots, s_m$
- Computes the attention scores
  - the "relevance" of encoder state s_k for decoder state h_t
- Computes the attention weights
  - this is a probability distribution of Softmax applied to the attention scores
- Computes the attention output
  - The weighted sum of encoder states (per state???????!?!?!?!?!?!?!?!?!?) with attention weights

- **Attention input**
  - "all encoder states, one decoder state"
    - $s_1, s_2, \dots, s_m \quad h_t$
- **Attention scores**
  - "How relevant is *source token k* for *target step t*?"
    - $\text{score}(h_t, s_k), k = 1..m$

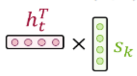
We go through each token in the *source sentence* and then also *the representation*.

- **Attention weights**
  - "attention weight for *source token k* at *decoder step t*"
    - $$a_k^{(t)} = \frac{\exp(\text{score}(h_t, s_k))}{\sum_{i=1}^{m} \exp(\text{score}(h_t, s_i))}, k = 1..m$$
- **Attention output**
  - "source context for *decoder step t*"

$$c^{(t)} = a_1^{(t)}s_1 + a_2^{(t)}s_2 + \dots + a_m^{(t)}s_m = \sum_{k=1}^{m} a_k^{(t)}s_k$$

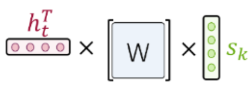
Dot product

$$\text{score}(h_t, s_k) = h_t^T s_k$$



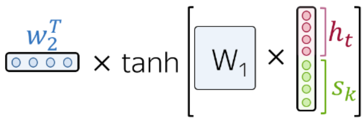
Bilinear

$$\text{score}(h_t, s_k) = h_t^T [W] s_k$$

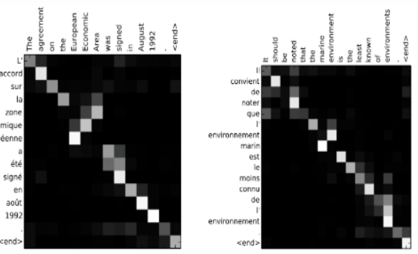


Multilayer perceptron

$$\text{score}(h_t, s_k) = w_2^T \cdot \tanh([W_1] \times [h_t, s_k])$$



Going back to our machine translation assignment, we can see the *alignment* of all source tokens, and see which ones are actually counting.



Bahdanau attention

A Bidirectional RNN encoder.
Uses a Multilayer perceptron attention score.
The attention is applied between decoder steps: state h_{t-1} is used to compute attention $c^{(t)}$, and both are passed to the decoder at step t.

Luong Attention

A unidirectional RNN encoder.
Uses a Bilinear function.
The attention is applied after state h_t has finished computing. The output is not immediately sent as final: an updated representation ($\tilde{h}_t$) using $c^{(t)}$ and h_t is sent to make the actual prediction.

$$\tilde{h}_t = \tanh(W_c[h_t, c^{(t)}])$$

Transformers

Transformers ask the question "why can't we do *everything* with Attention?"

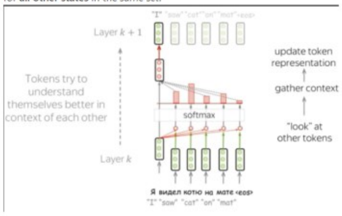
Any by everything, they mean *processing* in **both** *encoder and decoder*, and also *encoder/decoder interaction*.

Using this, transformers can floor Recurrent Neural Networks because each token literally knows the whole sentence; all of a transformer's encoder token can interact with each other.

Transformers come in "blocks", each a multi-layer network that maps sequences of input vectors $(x_1, \dots, x_n)$ to sequences of output vector $(z_1, \dots, z_2)$

Self-attention

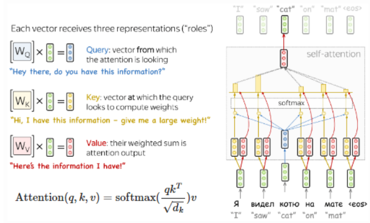
For each state in a set of states (say, in the *encoder*), we will use attention for **all other states** in the *same set*.



This computation happens in parallel for each state

Query-key-value attention

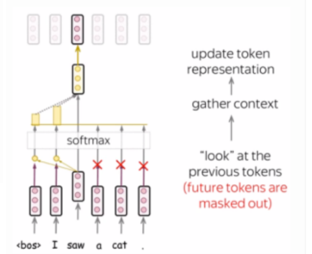
To keep this more structured, each input token receives three representations depending on what purpose it's serving.



$$\text{Attention}(q, k, v) = \text{softmax}(\frac{qk^T}{\sqrt{d_k}})v$$

Masked self-attention

We will run into problems when using the *decoder* for generating text - can't look into the future! We perform attention with a mask to hide the future tokens.



Even during training, we can't get the decoder to "see future". This masking also makes training transformers $O(1)$.

Feed-forward blocks

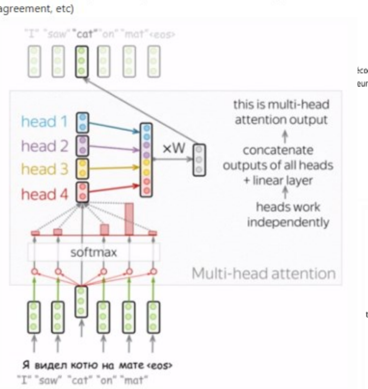
Between layers, we include a *feed-forward* block - two linear layers with $\mathbf{ReLU}$ between them

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

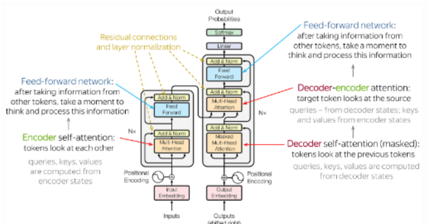
This is used to process the new information gained from attention.

Multi-head attention

Each word can be part of many relations (verb inflection, gender agreement, etc)



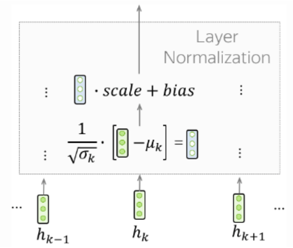
This is the motivation behind *multi-head* attention - we capture *different ideas* (recall multi-layer only *refines* these ideas).



Layer normalisation

We normalise the vectors in each layer to make sure they don't get too **crazy!!!!**

LayerNorm has trainable parameters **scale** and **bias** (trainable for each layer).



Transfer Learning

The general idea of *transfer learning* is to "transfer" knowledge from one model/task to another.

- **From scratch**
  - might not even *learn relationships*
- **Pre-trained**
  - Using pre-trained embeddings from stuff such as Word2Vec will lead to embeddings that **know relationships** between words but are *not specific*
- **pre-training, and *then* fine-tuning**
  - *now specific to our task*

Positional encoding

Transformers don't inherently know the order of an input. These can be learned, but fixing *positional encodings* doesn't hurt quality. Transformers use

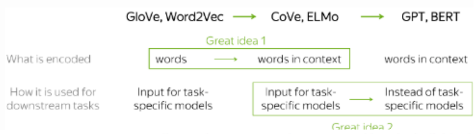
$$PE_{pos,2i} = \sin(pos/10000^{2i/d_{model}})$$
$$PE_{pos,2i+1} = \cos(pos/10000^{2i/d_{model}})$$

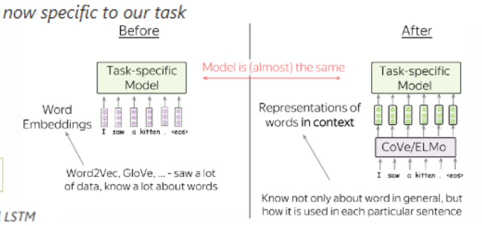
and each input is a **sum of the two embeddings** *token* and *position*.

Limitations of transfer with word embeddings

- They encode word meanings without *considering context* (merging all senses for a word)
- They also lack *larger linguistic unit* representation (phrases, sentences, paragraphs).

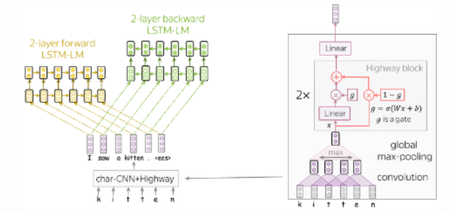
Word-in-context embeddings



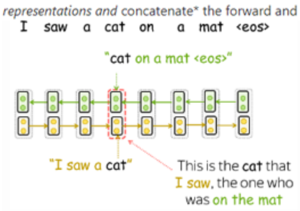


ELMo

Embeddings from **L**anguage **Mo**dels *(ELMo) is a simple model, consisting of two-layer* LSTM *(not covered in course just trust the plan) models, forward and backward. Each character of a word is pooled by a [CNN](CNN).*



To actually get the representations, we combine the individual LSTM representations and concatenate the forward and backward vectors.



Masked Language Modelling

BERT also uses *masked language modelling.*

At each training step, BERT:

- randomly picks 15% of the tokens
- replaces each of these chosen tokens with something
- and then predicts the original chosen tokens