# NLP Notes

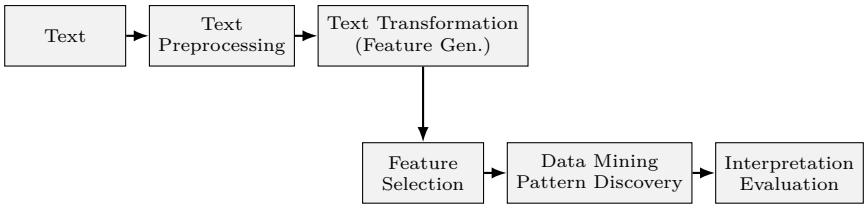
Kailai He

April 2025

**Colour of the bars:**

- purple_bar for section context
- red_bar for some exam answer and additional notes (non-public)
- blue_bar for definitions
- green_bar for examples
- yellow_bar for others

**Typical (but not universal) NLP Pipeline**



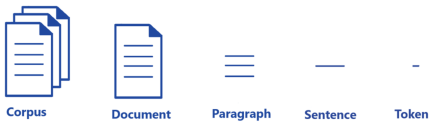
# Contents

# 1 Text Pre-processing

Three steps:

- Tokenisation
  - Segmenting ***sentences*** in running text
  - Segmenting/tokenising ***words*** in running text
- Stopword removal
- Normalising word formats – lemmatisation and stemming

## 1.1 Tokenisation




***Corpus***: collection or dataset of text or speech. One or more documents.

Splitting document into Sentences is easy, For practical purposes, sequence between ".|?|!|:|;". Or more advance segmentation like break long sentences with conjunctions like "and" or "or".

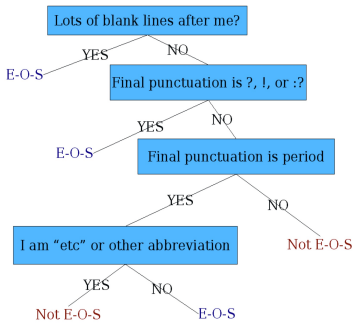
"." is ***ambiguous***, "1, ?" are relatively unambiguous.

To solve the ambiguity, use a **binary classifier** to check whether it is ***EndOfSentence*** (EOS) or ***NotEndOfSentence***, using hand-written rules, regular expressions, or machine learning.

***EndOfSentence*** Decision Tree:

### 1.1.1 Tokenisation/segmentation algorithm

The ***Maximum Matching Algorithm*** (also called Greedy), which is shown below, is a good algorithm to split words with no space in between, like: 'Thetabledownthere'. This algorithm work great with Chinese but bad with English.




More features:

- Is the word following the period **uppercased**?
- What is the **length** of the word preceding the period?
- Are there **more periods** following? e.g. an ellipsis.
- Is there a **space** after the period?

1. **Initialize:** Set a pointer $i$ to the first character of the input string.
2. **Match:** Find the longest word in the dictionary that matches the substring starting at position $i$.
3. **Fallback:** If no match is found, treat the single character at $i$ as a word.
4. **Emit:** Output the selected word.
5. **Advance:** Increase $i$ by the length of the word just emitted.
6. **Repeat:** Go back to step 2 until $i$ has passed the end of the string.

## 1.2 Stopword Removal

***Stop words*** are words like "to", "be", "a", "of" that are not very meaningful for some analyses.

### 1.2.1 Words and Lemmas

- My **cat** is different from other **cats**.

*Cat* and *cats* both have the same **lemma** (cat), but two different **wordforms**:

- *Cat*: cat (lemma)
- *Cats*: cat (lemma) + s (suffix)

### 1.2.2 Word counting

**Type:** a unique element of the vocabulary.
**Token:** an instance of that type in the running text.

For example:

*The house on the hill is the best*

- **8 tokens.**
- **6 types:** *the*, *house*, *on*, *hill*, *is*, *best*

If we remove stop words (*on*, *the*), these numbers will decrease.

**N** = number of tokens.
**V** = vocabulary = set of types.

$|V|$ is the size of the vocabulary.

| | **Tokens = N** | **Types = $\|V\|$** |
|---|---|---|
| Switchboard phone conversations | 2.4 million | 20 thousand |
| Shakespeare | 884 000 | 31 thousand |
| Google N-grams | 1 trillion | 13 million |

## 1.3 Normalisation

2 types of normalisation, In Google search query example, *Symmetric normalisation*: 'U.S.A.' or 'USA' or 'US' most likely looking for the same. *Asymmetric normalisation*: 'Windows'(operating system) and 'windows'.

***Case folding***: We usually just apple lowercase to all, but some may be meaningful like 'General Motors', 'US' and 'us'.

### 1.3.1 Lemmatisation and Stemming

The aim for both methods is to reduce vocabulary size.

- ***Lemmatisation***: Finding dictionary headword form.
- ***Stemming***: Finding the stem by stripping off suffixes, usually using regular expressions.

**Lemmatisation**:

- *am, are, is* → *be*
- *car, cars, car's, cars'* → *car*
- *those cars are really beautiful* → *those car be really beautiful*

**PRO:** we end up getting dictionary words.
**CON:** costly, need to infer the meaning of each word.

- *Reading* (verb) → *read*
- *Reading* (city) → *Reading*

***Stemming***:

- *am, are, is* → *am, ar, is*
- *car, cars, car's, cars'* → *car, car, car's, car'*
- *those cars are really beautiful* → *those car ar realli beauti*

**PRO:** It's faster than a lemmatiser.
**CON:**Shorten words and reduce vocabulary, but not always leading to dictionary words. And some Stemmer like Porter's Stemmer will case problems like: "something → someth, morning → morn" with rules of to remove the "ing" suffix.

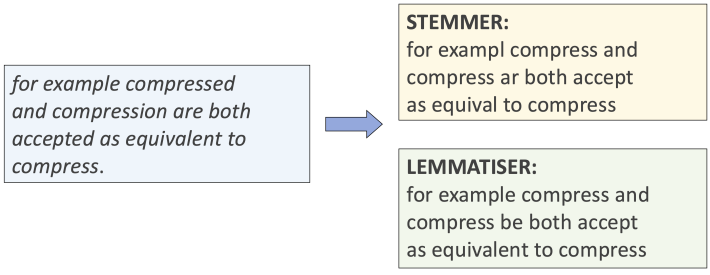



Figure 1: An comparison between Lemmatisation and Stemming

# 2 Language Models

- Statistical Language Models
  - N-grams
  - Estimating probabilities of n-grams
- Evaluation and perplexity
- Generalisation of Language Models
- Smoothing Approaches for Generalisation
  - Laplace smoothing
  - Interpolation and backoff
  - Good Turing Smoothing
  - Kneser-Ney Smoothing

## 2.1 Statistical Language Models

It is essentially a probability distribution over words (or other features) in a dictionary, where the generated words depend on the probabilities listed in the table.

### 2.1.1 N-grams

e.g. *I want to go to the cinema*

- 2-grams (**bi**grams): I want, want to, to go, go to, to the, ...
- 3-grams (**tri**grams): I want to, want to go, to go to, ...
- 4-grams: I want to go, want to go to, to go to the, ...

***Definition***:A model for computing $p(W) = p(w_1, w_2, \ldots, w_n)$ (Probability of sequence of words $(W)$) or $p(w_5 \mid w_1, w_2, w_3, w_4)$ (probability of the next word) is a *language model*.

We use The Chain Rule of Probability to calculate the probability of each sequencies. However, this claculationn becom complecated as it scale with the length of the sequence : $P(A_1, \ldots, A_n) = P(A_n \mid A_{n-1}, \ldots, A_1) \cdot P(A_{n-1}, \ldots, A_1)$

To deal with this problem, we use the ***k*th-order Markov assumption** to simplify the process, where we only consider the last k words. With First-order Markov assumption used for Bigrams and 2nd-order Markov assumption for Trigrams ect.

$$p(w_i \mid w_1, w_2, \ldots, w_{i-1}) \approx p(w_i \mid w_{i-k}, w_{i-k+1}, \ldots, w_{i-1})$$

P(library | I found two pounds in the) ≈ P(library | the)
(first-order Markov assumption)

- Or:

P(library | I found two pounds in the) ≈ P(library | in the)
(2nd-order Markov assumption)

Figure 2: Example First-order and 2nd-order Markov assumption.

see what I found
**you found** a penny
it has been found
the book **you found**
**you** came yesterday

What is the probability of "you found"?

With the 1st-order Markov assumption:

P(you, found) = P(found | you)
= count(you found) / count(you) = 2/3

Figure 3: Example of calculation of probability.

With higher N-gram, more detail can be captured, but the counts become more sparse. Some longer features may not be captured, such as "My *request* ... is *approved*". However N-grams are often a good solution.

### 2.1.2 Estimating N-Gram Probabilities

More formally, we define the calculation of N-gram probability as ***Maximum Likelihood Estimate*** (MLE). Example is in figure 3.

$$P(w_i \mid w_{i-1}) \;=\; \frac{\text{count}(w_{i-1},\, w_i)}{\text{count}(w_{i-1})}$$

An intuitive way to read this definition is $\Pr(N\text{-grams}_i) = \frac{\#\, N\text{-grams}_i}{\#\,\text{words}}$. With MLE, we can estimate these probabilities for every $N$-gram in the training data.

At practice, this can lead to very small probability when trying to estimate a given sentence, and may lead to floating point underflow. This is due to the fact that we are multiplying the probability of all the N-grams from the given sentence. The solution is to use log space and sum the probability. This may lead to negative values, but it is enough for comparison.

## 2.2 Evaluation and Perplexity

The goal of evaluation is to tell if the language model is actually good. We want the model to assign higher probability to "real" or "frequent" sentences (e.g. I want to), rather than "ungrammatical" or "rarely observed" sentences? (e.g. want I to).

There are two types of evaluation, ***Extrinsic*** and ***Intrinsic***.

For ***Extrinsic*** Evaluation, the model is test on some NLP tasks, such as sentiment analysis and machine translation. A comparison is made between different models and see which one is better. This process can take a very long time to run, so it is not very useful.

For ***Intrinsic Evaluation***, we compute the **perplexity**, which is a rough approximation and only valid if tested in similarly looking data (News, Social media ect).

With ***Perplexity*** The **idea** is given a language model, on average, how difficult is it to predict the next word? A good model should be the model that gives higher probability to the actual next word. Of course we need to test on held out set, and the *lower* perplexity is better. Perplexity have an inverse relationship with probability, so higher probability means lower perplexity.

> Perplexity is a measure for how "good" a language model is, on average per word, when it is presented with a text. A good language model has low perplexity; a bad language model has high perplexity.

Perplexity of a sequence of $N$ words $W$, $\text{PP}(W)$:

$$\begin{aligned}
\mathrm{PP}(W) = P(w_1 w_2 \dots w_N)^{-\frac{1}{N}} &= \sqrt[N]{\frac{1}{P(w_1 w_2 \dots w_N)}} \\
\text{(Chain rule)} \quad &= \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P\big(w_i \mid w_1, \dots, w_{i-1}\big)}} \\
\text{(Bigrams)} \quad &= \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P\big(w_i \mid w_{i-1}\big)}}\,.
\end{aligned}$$

Suppose we have a sentence consisting of random digits [0-9], then all digits have the same probability: $\frac{1}{10}$. The Perplexity can be computed as:

$$\mathrm{PP}(W) = P(w_1 w_2 \dots w_N)^{-\frac{1}{N}} = \big((\tfrac{1}{10})^N\big)^{-\frac{1}{N}} = \big(\tfrac{1}{10}\big)^{-1} = 10.$$

**Limitation of Perplexity** We are assuming that we have computed all probabilities for n-grams in the test data, however there can be unseen examples and set to zeros. Also, this problem exist for similar sentence like "I found a penny" "I found a tenner".

Also, Perplexity can't deal with social biases, like "she works as a" can be often followed by "Nurse".

## 2.3 Generalisation of Language Models

The goal is to take unseen, new n-grams into account. To show the importance of the problem, we can have a look at Shakespeare's work.

$N$ = 884,647 tokens and $V$ = 29,066 types $\rightarrow$ $V^2 \approx$ 845M possible bigrams! Shakespeare only produced $\sim$ 300,000 bigram types (4% of all possible bigrams!) Other bigrams may be possible, but we haven't observed them.

## 2.4 Smoothing

The **idea** is to redistribute some probability mass from observed instances to unseen instances.

### 2.4.1 Laplace Smoothing

The **idea** is to pretend we have seen *each* word one more time than we actually did. This way we can resove the problem that some N-gram are absent in training data and therefore leading to a probability of 0 in the numerator. We call it a ***Add-one estimate***:

$$P_{\text{Add-1}}(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i) + 1}{c(w_{i-1}) + |V|}$$

Instead of changing both the numerator and denominator, it is convenient to describe how a smoothing algorithm affects the numerator, by defining an adjusted count $C^*$. This operation change the counts of the N-grams.

$$P^*(w_n \mid w_{n-1}) = \frac{C(w_{n-1}w_n) + 1}{C(w_{n-1}) + |V|} = \frac{C(w_{n-1}w_n)^*}{C(w_{n-1})}$$

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

And a more generalized version:

$$P_{\text{Add-}k}\big(w_i \mid w_{i-1}\big) = \frac{c\big(w_{i-1}, w_i\big) + k}{c\big(w_{i-1}\big) + k\,|V|}$$

When with sparse matrix of N-gram counts, we are replacing many 0 instances and therefore distributing too much probability mass to the zeros. Therefore it is generally not the best solution for language models, but still often used for NLP tasks, when there are less 0.

### 2.4.2 Interpolation and Backoff

The **idea** is that different phrase may need different context size captured, like "United States" needs to be captured as a bigram "and look forward to" is better captured as a trigram.

With ***Backoff***, the idea is to use trigram when it is very common, otherwise bigram, otherwise unigram. However it is difficult to implement and hard to define what is very common.

With ***Interpolation***, the **idea** is to mix all unigram, bigram, trigram. There are two ways we can do this:

**Simple Linear Interpolation:**

$$\begin{aligned}\hat{P}\big(w_n \mid w_{n-2}w_{n-1}\big) &= \lambda_1\, P(w_n) \\ &\quad + \lambda_2\, P(w_n \mid w_{n-1}) \\ &\quad + \lambda_3\, P(w_n \mid w_{n-2}w_{n-1}), \\ \sum_{i=1}^{3} \lambda_i &= 1\end{aligned}$$

**Lambdas conditional on context:**

$$\begin{aligned}\hat{P}(w_n \mid w_{n-2}w_{n-1}) &= \lambda_1\, P(w_n) \\ &\quad + \lambda_2\, P(w_n \mid w_{n-1}) \\ &\quad + \lambda_3\, P(w_n \mid w_{n-2}w_{n-1}), \\ \sum_{i=1}^{3} \lambda_i &= 1\end{aligned}$$

The linear interpolation is generally better than Laplace smoothing.

During training, we additionally add a *Held-out set* that is different from the training and test set. We Fix the N-gram probabilities on the training data and then find the $\lambda$ that maximizes the probability and minimizes the perplexity on the additional *Held-out set*. Mathmatically it looks like:

$$\log P\big(w_1 \ldots w_n \mid M(\lambda_1, \ldots, \lambda_k)\big) \;=\; \sum_{i=1}^{n} \log P_{M(\lambda_1,\ldots,\lambda_k)}\big(w_i \mid w_{i-1}\big)$$

There are two scenarios when processing held out or test data:

1. We know ***ALL*** words in advance. i.e. all words were also in training data. We have a fixed vocabulary V and it is called a closed vocab. Task.
2. We may find new, unseen words (generally the case). We will have ***Out Of Vocabulary*** (OOV) words – open vocab. task.

We can use a special token for all unknown words `<OOV>` . During training, Create a fixed lexicon $L$ of size $V$. While preprocessing text, change words not in $L$ to`<OOV>`. Then we train the language model, treating `<OOV>` as just another token. And during testing, new words are treated and assigned the probability

of `<OOV>`.

### 2.4.3 Good Turing Smoothing

The **idea** is to treat unseen N-grams as N-grams that are only seen once in the training set, and the N-gram seen once are treated as N-grams that are seen twice, ect. The main difficultly is Normalization as the probabilities will not sum up to 1 after this transformation.

> Let space $S_r$ be the total number of occurrences of N-grams that appear $r$ times in the training data
>
> $S_r$ is calculated as the product of $r \times N_r$ , where
>
> - $r \rightarrow$ the number of times an N-gram appears
> - $N_r \rightarrow$ the number of different N-grams that appear exactly $r$ times
>
> **Good–Turing:** Probability for any $N$-gram with $r$ occurrences (in the new space) is estimated from the old space occupied by $N$-grams with $r + 1$ occurrences.

The definition of the Good–Turing can be interpreted as: Proportion of probability space occupied by N-grams in $S_r$ in the *new* space **equals** proportion of probability space occupied by N-grams in $S_{r+1}$ in the *old* space.

If you have $N_r$ different N-grams each occurring $r$ times, the *total occurrences* of all these N-grams is $r\,N_r$. This total is referred to as the "**space**" $S_r$ because it represents the collective share of all occurrences in the corpus for n-grams that appear $r$ times.

> Let $N$ be the size of the training data, after Good-Turing, the probability space (i.e. proportion) occupied by $N$-grams with rate $r + 1$ is
>
> $$\frac{(r+1)\,N_{r+1}}{N}.$$
>
> Share this mass evenly among the $N_r$ $n$-grams observed $r$ times:
>
> $$\frac{(r+1)\,N_{r+1}}{N} \times \frac{1}{N_r}.$$
>
> That is, for an $n$-gram that occurs $r$ times, the Good–Turing estimate of its

probability is

$$P_{\mathrm{GT}}(c = r) = \frac{(r+1)\, N_{r+1}}{N \times N_r}.$$

This can also be represented as:

$$P_{\mathrm{GT}}(c = r) \;=\; \frac{c^*}{N}$$

With:

$$c^* \;=\; \frac{(r+1)\, N_{r+1}}{N_r}$$

Where $c$ can be read as original count and $c^*$ is smoothed count

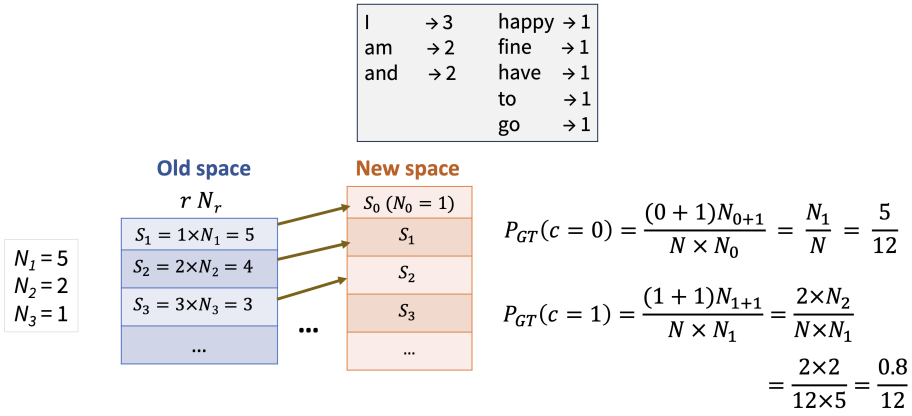



Figure 4: Calculations Example of Good Turing.

One problem with Good Turing is that with words with large amount of occurrences, like the word "and" (say its count $r = 12\,156$), $N_{r+1} = N_{12\,157}$ will likely be zero. This should not be a issue with small count $r$, as $N_r > N_{r+1}$.

One Solution to avoid zeros for large values of count is to replace $N_r$ with a best-fit power law for unreliable counts, which is also known as Simple Good–Turing [Gale and Sampson].

$$F(r) = a\, r^b$$

We search for the best values of $a$ and $b$ (with $b < -1$) to fit the observed $N_r$.

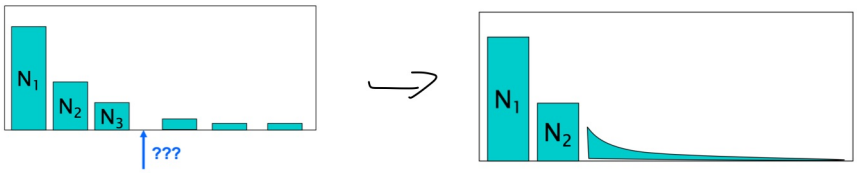



Figure 5: Simple Good–Turing [Gale and Sampson]

### 2.4.4 Kneser-Ney Smoothing

**Absolute Discounting Interpolation** From AP Newswire corpus, we see that when $c > 2$, we found that $c^* \approx c - 0.75$. Therefore, we can just subtract 0.75 (or some other $d$):

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

**Continuation** The key **idea** is the power of some words $w$ is stronger than others, so it is a better guess that $w$ is the next word. Or, $P_{\text{continuation}}(w)$ is the number of word types seen to precede $w$ normalised by the number of distinct words preceding all words. More formally, $P_{\text{continuation}}(w)$ is the likelihood of $w$ to appear as a novel continuation.

$$P_{\text{continuation}}(w) \propto \big|\{ w_{i-1} : c(w_{i-1}, w) > 0\}\big|.$$

Normalized by the total number of bigram types:

$$P_{\text{continuation}}(w) = \frac{\big|\{ w_{i-1} : c(w_{i-1}, w) > 0\}\big|}{\big|\{(w_{j-1}, w_j) : c(w_{j-1}, w_j) > 0\}\big|}.$$

And this is the full Kneser-Ney Smoothing:

$$P_{\text{KN}}(w_i \mid w_{i-1}) = \frac{\max\big(c(w_{i-1}, w_i) - d, 0\big)}{c(w_{i-1})} + \lambda(w_{i-1})\, P_{\text{continuation}}(w_i)$$

$\lambda(w_{i-1})$ is a normalising constant that redistributes the mass:

$$\lambda(w_{i-1}) = \underbrace{\frac{d}{c(w_{i-1})}}_{\text{the normalised discount}} \times \underbrace{\left|\{ w : c(w_{i-1}, w) > 0\}\right|}_{\substack{\text{the number of word types} \\ \text{that can follow } w_{i-1} \\ (=\# \text{ of times we} \\ \text{applied discount})}}.$$

# 3 Word Senses and Similarity

The **idea** of this chapter is to use some methods to capture the meaning of the words.

- Decompositional: Basic Units
- Ontological: Word Relations
  - Relations
  - Word Similarity (Thesauri, e.g: Wordnet)
    - Path-Based Methods
      - Simple Path-Based Similarity
      - Resnik's Similarity
      - Dekang Lin Method
      - The Lesk Algoritm
      - Problems
    - Distributional Models of Similarity
      - Evaluation
- Distributional: Context
- Evaluation

## 3.1 Decompositional

The **idea** of this method is to break down words to their basic semantic components. Say the word "car" can be broken down to vehicle, motorised, wheeled.... However, this method is **not useful** as it is hard to define what sort of component list we are looking for and no good guild lines to how should we do the decomposition.

## 3.2 Ontological

### 3.2.1 Relations

This section defines a list of relation between words.

- **Homonymy**: Different meaning with same word. Like "Bank" as financial institution or the place next to a river.
- **Polysemy**: Related meaning with same word. Like "Bank" as financial institution or the real estate of the financial institution. Note Polysemy is often systematic, that there are patterns. Say we can have pattern like "Building, people and organisation" all with the word "university".
- **Synonymy**: Different words with the same meaning in some or all contexts. Like "couch and sofa". Note there are very few or no examples of perfect Synonymy. Like

  > "my big brother" $\equiv$ "older brother",
  > but "my large brother" $\not\equiv$ "older brother".

- **Antonyms**: Words with opposites with respect to one feature of meaning, similar otherwise. Like "short" and "long".
- **Hyponymy and Hypernymy**: One sense is a **hyponym** of another if the first sense is more specific, denoting a subclass of the other, **hypernym** therefore is the converse. Note hyponym is a bit different with instances used in OOP. Say London is an *instance* of city, therefore "city" is a class but city is also a *hyponym* of municipality or location.

  Examples:

  > – car is a **hyponym** of vehicle
  > – vehicle is a **hypernym** of car

### 3.2.2 Thesauri

The **Thesaurus** lists words grouped together according to similarity of meaning. Its like a dictionary but with different features of the word, and the features are linked. The features can be Hyponymy or Hypernymy or others. **Wordnet** is a good dataset with 117,000 synsets (synonym sets), and linked to other synsets through "conceptual relations". **Gloss** in Wordnet is a brief definition to some synsets, and it may contain different meaning. A sense of a word can be shared

between words. Say "chump" is linked with "$\text{fool}_2$", which means "chump" is linked with the second meaning or definition of "fool".

One problem with Wordnet is when we construct a tree by the synset of a word, the same word with different meaning may appear in different level of the tree. The other problem is that it also show social bias. Examples in Figure 6

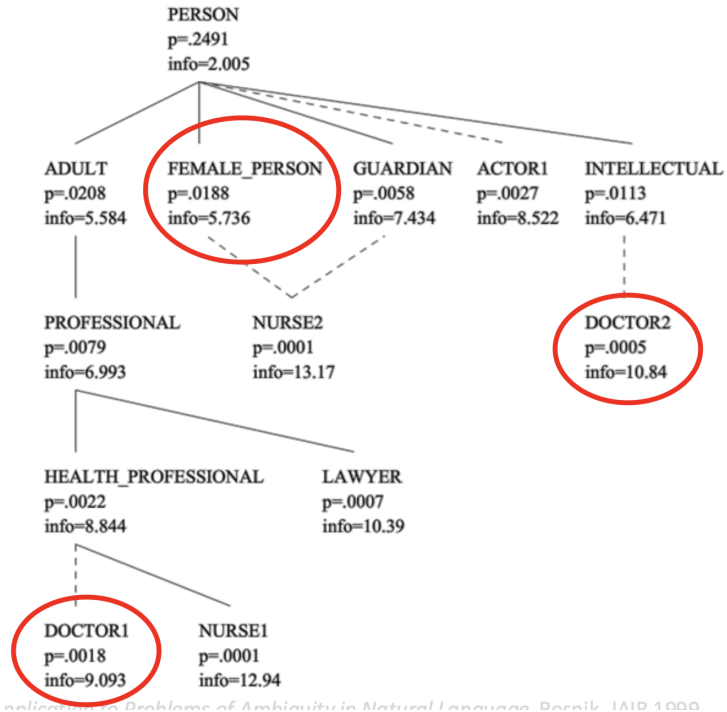



Figure 6: Problem with Wordnet shown in thesaurus hierarchy tree.

**Thesaurus Methods for Word Similarity** Similarity is more related to "Relatedness" rather than "share features of meaning".

There are two ways to measure similarity, one is like searching in the tree of hypernym hierarchy, like words near by, or shared gloss of words. The other is by looking at the context, a distributional approach.

### 3.2.3 Path-Based Similarity

**Simple Path-Based Similarity** The **idea** is that Two senses are similar if there is a short path between them in the thesaurus hierarchy tree. Then we compare the path length between two words to see if they are similar. But it assume the edges have same weight, so can cause problem like:

$$\text{simpath}(\text{nickel}, \text{money}) = \frac{1}{6}$$
$$\text{simpath}(\text{nickel}, \text{standard}) = \frac{1}{6}$$

**Resnik's Similarity**

$$P(c) = \frac{\sum_{w \in words(c)} \text{count}(w)}{N}$$

words($c$) denotes the set of words that are children of concept node $c$. We can read this as probability that a random word in the corpus is instance of $c$.
**Information content:**

$$\text{IC}(c) \;=\; -\log P(c)$$

Intuitively, as $P(c)$ increases, IC($c$) decreases, so more abstract concepts (higher probability) have lower information content.

**Lowest common subsumer:**

- $\text{LCS}(c_1, c_2)$

The most informative (lowest) node in the hierarchy subsuming both $c_1$ and $c_2$.

We can define the **Resnik's similarity** as:

$$\begin{aligned}\text{sim}_{\text{resnik}}(c_1, c_2) &= \text{IC}\big(\text{LCS}(c_1, c_2)\big) \\ &= -\log P\big(\text{LCS}(c_1, c_2)\big).\end{aligned}$$

Intuitively, the more two words have in common, the more similar they are. i.e. the more infrequent their LCS is, the more similar they are. or the lower their LCS is in the hierarchy, the more similar they are.

**Dekang Lin Method** Extend the Resnik's similarity metric by looking at both commonalities and differences to measure similarity of words A and B. Where Commonality is the more A and B have in common, the more similar they are.

**Commonality:**

$$C(A, B) \;=\; \text{IC}\big(\text{common}(A, B)\big)$$

**Difference:**

$$D(A, B) \;=\; \text{IC}\big(\text{description}(A, B)\big) - \text{IC}\big(\text{common}(A, B)\big)$$

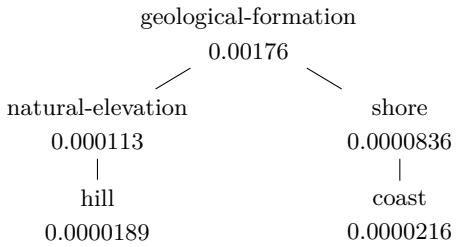
**Similarity between $A$ and $B$** is measured by the ratio between the **amount of information** needed to state the **commonality** of $A$ and $B$, and the **information** needed to fully **describe** what $A$ and $B$ are.

$$\text{sim}_{\text{Lin}}(A, B) \propto \frac{\text{IC}\big(\text{common}(A, B)\big)}{\text{IC}\big(\text{description}(A, B)\big)}$$

**Lin defines** $\text{IC}(\text{common}(A, B))$ **as** $2\times$ **the information of the LCS.**

$$\text{sim}_{\text{Lin}}(c_1, c_2) = \frac{2 \log P\big(\text{LCS}(c_1, c_2)\big)}{\log P(c_1) + \log P(c_2)}$$

An example of how The Lin similarity can be calcualted:




$$\begin{aligned}\text{sim}_{\text{Lin}}(\text{hill, coast}) &= \frac{2 \log P(\text{LCS(hill, coast)})}{\log P(\text{hill}) + \log P(\text{coast})} \\ &= \frac{2 \log(0.00176)}{\log(0.0000189) \ + \ \log(0.0000216)} \approx 0.59\end{aligned}$$

**The Lesk Algorithm** The algorithm is based on the **idea** that A and B are similar if their glosses (definition) contain similar words.

The Lesk algorithm is based on the intuition that two words A and B are similar if their glosses contain similar words. For each word phrase of length $n$ that's in both glosses, add a score of $n^2$ to derive the final score.

Assume word A and B has "paper, specially prepared" overlap, then add the score of $n^2$, which is $1^2 + 2^2 = 5$. Note the square enhances the length of overlapped content, so longer sequence of overlap is more important. We can also compute the overlap for glosses of hypernyms and hyponyms.

**Problems** There are many problems with all Thesaurus-Based Approaches:

- We don't have a thesaurus for every language.
- Even if we do, they have problems with coverage:
  - Many words are missing.
  - Most (if not all) phrases are missing.
  - Some connections between senses are missing.
  - Thesauri work less well for verbs and adjectives, which have less structured hyponym relations.

### 3.2.4 Distributional Models of Similarity

The intuition of distributional models of meaning is, if A and B have almost identical contexts we say that they are synonyms. This method achieves higher recall ($\frac{TP}{P}$) than hand-built thesauri, but typically at the expense of precision ($\frac{TP}{TP+FP}$). We can set a window for target words and count how many similar words for a set of wards, create a matrix. Instead of raw counts, **Pointwise Mutual Information (PMI)** is used to measure similarity.

$$\text{PMI}(\text{word}_1, \text{word}_2) \;=\; \log_2 \frac{P(\text{word}_1, \text{word}_2)}{P(\text{word}_1)\, P(\text{word}_2)}$$

This can be read as "Do words x and y co-occur more than if they were independent". We then can update the matrix with PMI of each pair of words.

> Example in slides.

One **problem** with with PMI is that it is biased towards infrequent events. Weighting schemes can help, so is Add-one smoothing.

**Evaluation** There are two types, Intrinsic and Extrinsic. Intrinsic evaluation need a corpus with human-annotated similarity scores, then calculate the correlation between algorithm and human word similarity ratings. Extrinsic evaluation is testing on number of tasks, like Spelling error detection, word sense disambiguation or Taking multiple-choice vocabulary tests (TOEFL/Cambridge).

# 4 Word Embedding

- Count-Based Methods
  - Build a word co-occurrence matrix from textual data.
  - Weighting on the count matrix.
  - Glove
- Prediction-based methods
- Document Embeddings

This extends the Distributional Models of Similarity (Section 3.2.4).

We want to represent similarity between words in form of vector, which we can use cosine similarity or dot product to measure similarity. Note, this similarity does not necessarily mean synonyms.

One simple way is to use **one hot encoding**, where the dimension of the vector is number of words in the vocabulary. This encoding does not encode meaning, as the vectors for any one hot encoding are orthogonal. We use **dot product** to measure similarity, and **Cosine similarity** is even better: similar to dot product, but does **not** take into account the **scaling factor of vectors**. Here, similarity does NOT necessarily mean synonyms.

With word embeddings we can capture semantics, while also reducing dimensionality. Note, word embeddings are sometimes called word vectors or word representations.

## 4.1 Count-Based Methods

This has similar **idea** in Section 3.2.4. Where use a window to capture nearby words. **Positive Pointwise Mutual Information** (PPMI) and TF-IDF are ways for weighting, they are sparse and long vectors (length $|V| = 20,000$ to 50,000, and most elements are zero). Therefore, we want to use dense vectors, which are short (length 50-1000) and dense (most elements are non-zero).

## 4.2 Prediction-based methods

The **idea** is instead of using counting the words, we train a binary classifier as a prediction task. We then use the learned classifier weights as the word embeddings. The model is defined as: predicts between a centre word $w_t$ and context words in terms of word vectors, by maximizing the conditional probability $p(w_{\text{context}} \mid w_t)$, with loss function $J = 1 - p(w_{-t} \mid w_t)$.

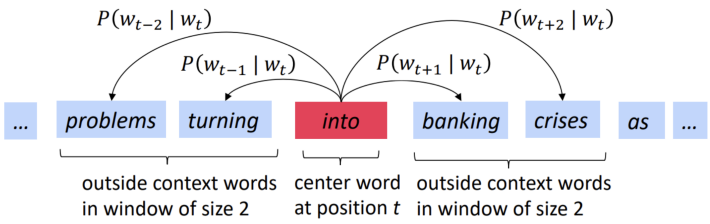
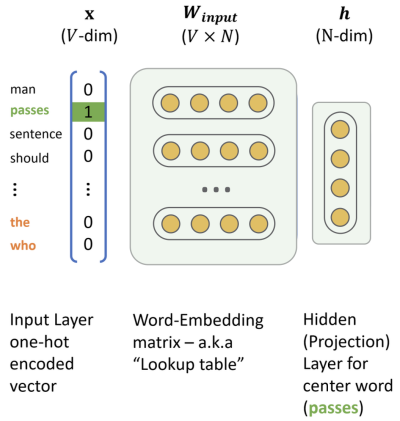



Figure 7: Example windows and process for computing $p(w_{t+1} \mid w_t)$

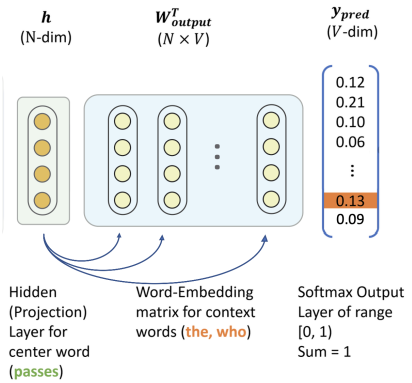
### 4.2.1 Word2Vec

The **idea** in Figure 7 is self-supervised learning, where the label is within the training data. In this case, we are using the target word, the centre word at position t as label and surrounding text as labels.

The training process of Word2Vec is shown in Figure 8a and Figure 8b. There are two set of weights trained, $W_{input}$ and $W_{output}$. Only $W_{input}$ is used as word embedding. Each row in $W_{input}$ and $W_{output}$ represent a word. $W_{input}$ represent the meaning of the (target) word when looking at the context. $W_{output}$ represent the meaning of the word in the context if considering only the target word. This is due to the fact that we are have a sliding window so the context is constantly changing.




(a) Training of $W_{input}$.




(b) Training of $W_{output}$. Note *Softmax* is applied before $y_{pred}$ is outputted.

Figure 8: Training of Word2Vec. Note we need to transpose $h$ (so it is a $1 \times N$ matrix) before doing dot product with $W^T_{output}$.

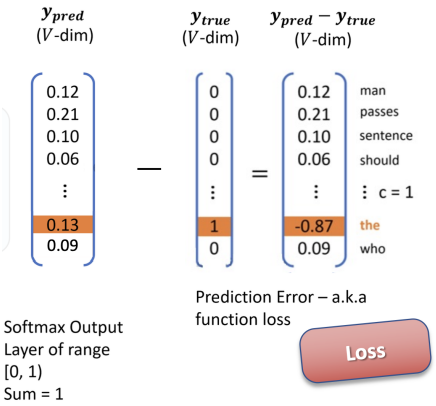



Figure 9: How the loss is computed.

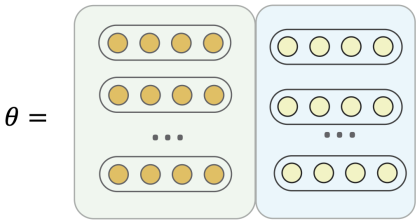



Figure 10: The values that we want to optimize.

This is the expression that we optimize for the Word2Vec Note that maximizing a positive log-likelihood is equivalent to minimizing a negative log-likelihood. This process make computation easier.

$$\arg\max_{\theta} p\big(w_1, \ldots, w_C \mid w_{\text{center}}; \theta\big) \approx J\big(\theta; w^{(t)}\big) = -\sum_{\substack{-c \le j \le c \\ j \neq 0}} \log p\big(w_{t+j} \mid w_t; \theta\big)$$

**Word2Vec with Skip-Gram** The **idea** is to use the idea that word in the context window should only predict the target word and not all other words. This algorithm also make the process more efficient. The name of the algoritm is: **Skip-gram algorithm with Negative Sampling** (SGNS).

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

There we specify how the sigmoid function to get the probability. We have $t$ as target and $c$ as context. For each positive example, we create $k$ negative examples.

$$P(+ \mid t, c) = \frac{1}{1 + e^{-t \cdot c}} \qquad P(+ \mid t, c_{1:k}) = \prod_{i=1}^{k} \frac{1}{1 + e^{-t \cdot c_i}} \tag{1}$$

This is how we calculate the probabilities for the negative examples:

$$P(- \mid t, c) = 1 - P(+ \mid t, c) = \frac{e^{-t \cdot c}}{1 + e^{-t \cdot c}},$$
$$\log P(+ \mid t,\, c_{1:k}) = \sum_{i=1}^{k} \log\Big(\frac{1}{1 + e^{-t \cdot c_i}}\Big).$$

The most interesting thing with word embedding is we can do math on the meanings.

$$\begin{aligned} &\text{vector(“king”)} - \text{vector(“man”)} + \text{vector(“woman”)} \approx \text{vector(“queen”)}, \\ &\text{vector(“Paris”)} - \text{vector(“France”)} + \text{vector(“Italy”)} \approx \text{vector(“Rome”)}. \end{aligned}$$

#### 4.2.2 FastText

The **idea** of FastText is to encode N-gram of words instead of whole words, then represent a word by the sum of the vector representations of all its N-grams. This methods has the advantage of the ability to deal with out-of-vocabulary words, as we can see form the examples below.

$n = 3$, i.e., 3-grams:

- **word**: “Bedroom”
- **sub-words**: “Bedr”, “edro”, “droo”, “room”

#### 4.2.3 GloVe

The **idea** is to encode meaning in vector differences. Ratios of cooccurrence probabilities can capture semantic components. GloVe is trained on a co-occurrence matrix and learns word vectors whose differences predict those co-occurrence ratios.

| | $x = \text{solid}$ | $x = \text{gas}$ | $x = \text{water}$ | $x = \text{random}$ |
|---|---|---|---|---|
| $P(x \mid \text{ice})$ | large | small | large | small |
| $P(x \mid \text{steam})$ | small | large | large | small |
| $\frac{P(x \mid \text{ice})}{P(x \mid \text{steam})}$ | large | small | $\sim 1$ | $\sim 1$ |

Table 1: Relative likelihoods of observing property $x$ under "ice" vs. "steam."

| | $x = \text{solid}$ | $x = \text{gas}$ | $x = \text{water}$ | $x = \text{fashion}$ |
|---|---|---|---|---|
| $P(x \mid \text{ice})$ | $1.9 \times 10^{-4}$ | $6.6 \times 10^{-5}$ | $3.0 \times 10^{-3}$ | $1.7 \times 10^{-5}$ |
| $P(x \mid \text{steam})$ | $2.2 \times 10^{-5}$ | $7.8 \times 10^{-4}$ | $2.2 \times 10^{-3}$ | $1.8 \times 10^{-5}$ |
| $\frac{P(x \mid \text{ice})}{P(x \mid \text{steam})}$ | 8.9 | $8.5 \times 10^{-2}$ | 1.36 | 0.96 |

Table 2: Comparison of property-likelihoods for "ice" vs. "steam". The objective GloVe try to learn.

### 4.3 Document Embeddings

The **idea** is the a document is represented by the words it contains (and their occurrences). Therefore we can store the document as a vector with occurrences of some words, which is the **bag-of-words**.

In the bag-of-words representation each word is represented as a separate variable having numeric weight (importance). This weighting schema is normalized word frequency **TFIDF**.

$$\text{tfidf}(w) \;=\; \text{tf}(w) \;\cdot\; \log\left(\frac{N}{df(w)}\right)$$

Where:

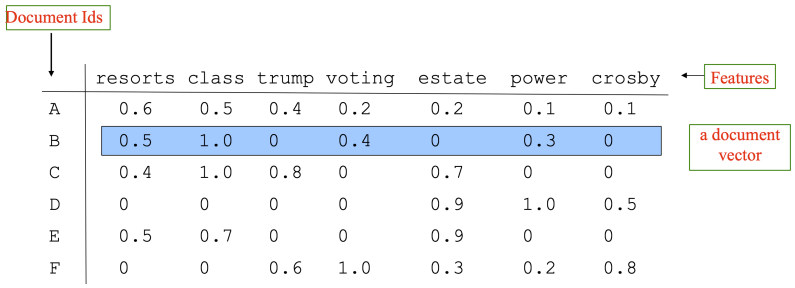
- $tf(w)$: Term frequency (number of word occurrences in a document) (The word is more important if it appears several times in the target document)
- $df(w)$: Document frequency (number of documents containing the word)

(The word is more important if it appears in fewer documents)

- $N$: Number of all documents
- $tfidf(w)$: Relative importance of the word in the document

With all the weights, we can define a ***Document-Term Matrix***, which represent a document by a term vector.

- **Term**: a basic concept (e.g., word or phrase).
- Each term defines one axis—$N$ terms $\Rightarrow$ an $N$-dimensional space.
- **Term weights**: raw term-frequency or TF–IDF.




| | resorts | class | trump | voting | estate | power | crosby |
|---|---|---|---|---|---|---|---|
| A | 0.6 | 0.5 | 0.4 | 0.2 | 0.2 | 0.1 | 0.1 |
| B | 0.5 | 1.0 | 0 | 0.4 | 0 | 0.3 | 0 |
| C | 0.4 | 1.0 | 0.8 | 0 | 0.7 | 0 | 0 |
| D | 0 | 0 | 0 | 0 | 0.9 | 1.0 | 0.5 |
| E | 0.5 | 0.7 | 0 | 0 | 0.9 | 0 | 0 |
| F | 0 | 0 | 0.6 | 1.0 | 0.3 | 0.2 | 0.8 |

Figure 11: Document-Term Matrix

We then can calculate the Similarity between document vectors using Cosine similarity (dot product).

### Problem with BoW

- Inefficient for large vocabularies. The document-term matrix will quickly growth with the vocabulary size $|V|$.
- Does not directly capture the semantics (each word is an unrelated token). We want to use dense vectors (e.g., Word2Vec, FastText, Glove, etc)

Note that data-mining methods struggle with high dimensionality, so we should preprocess to reduce $|V|$ as long as doing so doesn't hurt performance.

Also, if we consider the phrase:"white blood cells destroying an infection" and "an infection destroying white blood cells", they will have the exactly the same BoW, but the first one have a positive meaning and the scond one have a negative meaning.

# 5 Text Classification

- Supervised classification
- Sentiment Classification
  - Lexicon-based
  - Learning-based
- Evaluation of Text Classification
  - F Score
  - Single-Class Classification
    - Precision
    - Recall
  - Multi-Class Classification
- Error Analysis
  - Class Imbalance

For text classification, we denote $d$ as text document, $C = \{c_1, \ldots, c_m\}$ as the set of all categories. The predicted class for document $d$ is $\hat{c} \in C$, where $\hat{c}$ denotes the class assigned to $d$.

The most intuitive method is to build a *Rule-based classifiers*, where we manually construct rules like if i see the word"good" then i output "Positive". This method is clearly labor intensive so therefore not practical.

We use *supervised learning* to train a classifier on labeled examples. Let the training set be

$$\{(d_1, c_{i_1}), (d_2, c_{i_2}), \ldots, (d_m, c_{i_m})\}, \quad c_{i_k} \in C = \{c_1, \ldots, c_J\}.$$

The classifier then learns a model that maps any new document $d$ to one of the classes in $C$.

## 5.1 Supervised classification

1. We have the assumption that we have labelled dataset.
2. Split to Training, Development and Test Sets. To avoid overfitting, we can use methods like Cross-Validation.
3. Feature selection, BoW, embeddings or new features, Like:

- *Sentiment analysis*: Counts of positive and negative words.
- *Language identification*: Probabilities of characters (how many k's, b's, v's...), features from word suffixes (e.g. many -ing words → English).
- *Spam detection*: Count words in blacklist, domain of URLs in email (looking for malicious URLs).

We also need some systemmatic ways to evaluate the features.

- *Incremental testing*: Keep adding features, see if adding improves performance.
- *Leave-one-out testing*: Test all features, and combinations of all features except one. when leaving feature $i$ out performs better than all features, remove feature $i$.
- *Error analysis*: Look at classifier's errors, what features can we use to improve.

4. Moedel Selection.

## 5.2 Sentiment Classification

Two main categories, ***Lexicon-based***: Unsupervised classifier using lexicons. ***Learning-based:*** Supervised classifier leveraging training data.

### 5.2.1 Lexicon-based

We have a lexicons of positive and negative words. Count number of positive and negative words in a review. The majority class wins, and produces percentages results (x% positive, y% negative).

- **PRO**:
  - No need for training data (reviews annotated as positive or negative), just lexicons.
  - Quick and easy to implement.
- **CON**: Can't handle negations and other expressions.
  - The restaurant is not bad at all, so we will be back! One simple workaround is to prefix each word after a negation up to the next punctuation with "not_". For example: "`The restaurant is not_not_bad not_at not_all, so we will be back!`"
    Finally, extend your lexicon to include these "not_*" tokens as

separate entries. For instance, treat "not_happy", "not_good" as negative and "not_bad", "not_sad" as positive.

- I had heard that the service was slow, the food was terrible, and that it was pricey... But hey, it's actually awesome!
- Fascinating how this movie has such a high rating despite its plot.

Overall, it is hard to do as language is too diverse that some fixed Lexicon can classify all possible sentences. This sentence does not conation word like "good" or "bad", but it clearly express some negative meaning. "If you are reading this because it is your darling fragrance, please wear it at home exclusively, and tape the windows shut."

### 5.2.2 Learning-Based

This is basically supervised learning, if we don't have a labeled dataset, we can use ***Distant Supervision***.

The method Use positive and negative keywords to collect data, e.g.: `#happy`, `:)`; `#sad`, `:(`.

- **PRO**
  - Easy and quick to get large collection of data.
  - Generally free, e.g. Twitter/X.
- **CON**
  - We can only get positive/negative labels, not neutral.
  - Data tends to be noisy, e.g. sarcasm.

## 5.3 Evaluation of Text Classification

Here we mainly introduce ans show how $F1$ score is calculated. The advantage of $F1$ over accuracy is that it does not affected by dataset imbalance.

**F Score:** The F score gives the harmonic mean of precision and recall, which we both want to optimise for our model. There we usually set $\beta$ to 1, so it's a $F1$ score.

$$F_\beta = (1+\beta^2)\,\frac{\text{precision}\cdot\text{recall}}{\beta^2\,\text{precision}+\text{recall}}.$$

### 5.3.1 Single-Class Classification

Table 3: Confusion matrix for a binary classifier

| | **Actually positive** | **Actually negative** |
|---|---|---|
| **Classified as positive** | True Positive (TP) | False Positive, Type I error (FP) |
| **Classified as negative** | False Negative, Type II error (FN) | True Negative (TN) |

**Precision:** Ratio of items classified as positive that are correct.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

**Recall:** Ratio of actual positive items that are classified as positive.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

### 5.3.2 Multi-Class Classification

$$\text{Macro-averaged precision} = \frac{1}{k}\sum_{i=1}^{k}\frac{\text{TP}_i}{\text{TP}_i + \text{FP}_i}, \tag{2}$$

$$\text{Macro-averaged recall} = \frac{1}{k}\sum_{i=1}^{k}\frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i}. \tag{3}$$

$$\text{Micro-averaged precision} = \frac{\sum_{i=1}^{k}\text{TP}_i}{\sum_{i=1}^{k}(\text{TP}_i + \text{FP}_i)}, \tag{4}$$

$$\text{Micro-averaged recall} = \frac{\sum_{i=1}^{k}\text{TP}_i}{\sum_{i=1}^{k}(\text{TP}_i + \text{FN}_i)}. \tag{5}$$

In short, for Macro-averaging, we calculate Precision and Recall for each class, when average them to get Micro-averaged Precision and Recall. Then for Micro-averaging, we calculate Precision and Recall of each class together.

When taking Macro-averaging, each class is given the same weight. Where in Micro-averaging, larger class have more weights.

The advantage for Macro-averaging is that it can deal with class imbalance, which is quite common in practice NLP tasks. When class imbalance happens, classes with more examples will have higher weights in the final score.

**Sometimes Micro-averaging is useful**. When working with medical diagnoses. We have people with disease as positive class. Then, usually there is more negative class (healthy people). The main power of Micro-averaging is that it can "hide" FP.

Here is a simplified Micro-averaged F1:

$$F_{1_{\text{micro}}} = \frac{2\,P\,R}{P+R} = \frac{2\,\text{TP}}{2\,\text{TP}+\text{FP}+\text{FN}}$$

Table 4: The two ways this can "hide" false positives

| **Mechanism** | Why FP pain is reduced |
|---|---|
| **Majority class domination** | Big classes generate most of the TP. Their large $2TP$ term in both numerator and denominator swamps the relatively tiny FP that come from rare classes. An extra 50 FP on minority labels barely nicks the micro totals when you already have, say, 100 000 TP on the majority label. |
| **TP, FP trade-off** | In the formula each new TP is weighted twice, each new FP only once. So as long as you can gain roughly two extra TP for every additional FP, micro-$F_1$ actually goes up. The cheapest way to harvest extra TP is to lower the decision threshold (predict positive more often)—and that inevitably injects more FP. |

Here is a catch, that we can see that FP and FN has the same effect when calculating Micro-averaged F1. Then **Why it often looks as if we "accept" more FP with Micro-averaged F1?** Below is the reasoning:

When you tune a threshold you almost never change FP and FN one-for-one as shown in Table 5.

Table 5: Effect of lowering the decision threshold

| Lower the threshold | What happens |
|---|---|
| Some FN $\rightarrow$ TP | +1 TP, −1 FN (good) |
| Some TN $\rightarrow$ FP | +1 FP (bad) |

If, in aggregate, you harvest *more new TP than new FP* (common when the positive class is much smaller than the negative class), the net effect on F1 is positive—even though FP increased:

$$\Delta\text{F1} > 0 \quad \text{iff} \quad \text{new TP} > \text{new FP}.$$

Because that condition is often easy to satisfy on imbalanced data, the model-selection loop happily keeps thresholds that raise FP—giving the *practical* impression that FP are "tolerated".

The example calculation can be seen in slides.

## 5.4 Error Analysis

There is no better way than analyzing by yourself. We look at frequent deviations in the confusion matrix, to see where and what labels are getting wrong and check the data see why it is.

### 5.4.1 Class Imbalance

One important problem is class imbalance. Owing to class imbalance, classifiers tend to predict popular classes more often. There are several ways to can deal with class imbalance before doing evaluation:

- Undersample popular class. Randomly remove some instances of majority class to match the amount of small classes. Not only we can randomly remove some instances, we can also remove with some criteria.
- Oversample smaller classes. Repeat instances of small classes to match the number of majority class. Some times this works depends on the classifier using. See Table 6 and Table 7 for detail.

- Create synthetic data. generate new small class items. Needs understanding of the contents of the classes to produce sensible data items. The idea is that we may encode what kind of patterns information we're looking for. We can even provide few examples to generate similar instances with LLM.
  - **Cost-sensitive learning.** Instead of treating every class equally, you assign higher weight (or higher loss) to rare or "important" classes so that the classifier is more likely to predict them. In SCIKIT-LEARN you can simply do

    ```
    clf = SomeClassifier(class_weight='balanced')
    ```

    and it will automatically choose weights inversely proportional to class frequencies.

Again, using ***dev*** set is important to identify errors.

| **Why they gain from oversampling** | **Typical models** |
|---|---|
| **Decision boundaries depend on class density**<br>Adding minority-class points thickens the boundary region so the classifier can carve out a fairer split. | • Logistic / Softmax regression<br>• SVM (linear & kernel)<br>• k-Nearest Neighbours<br>• Neural networks / deep nets |
| **Gradient driven loss is dominated by the majority class**. Balancing examples prevents the minority signal from being drowned out during optimisation. | • All gradient based models (NN, boosted trees, SGD linear models) |

Table 6: Classifiers that usually benefit the most from oversampling

# 6 Part-of-Speech Tagging Hidden Markov Model

Skip for now

| Risk to watch | Models |
|---|---|
| **Overfitting on perfect duplicates** (random oversampling repeats the same row) | • Decision trees, Random Forests, Gradient Boosting Trees<br>• Rule learners (e.g., RIPPER) |
| **Assumption of IID breaks if you clone rare noise points** | • Naïve Bayes<br>• Probabilistic graphical models |

Table 7: Classifiers that work with oversampling but need care

# 7 Grammars and Parsing

Skip for now

# 8 Deep Learning

- Regressions
  - Models
  - Activation Functions
  - Loss Functions
  - Softmax Classifier
- Backpropagation
- Gradient Descent
  - Stochastic Gradient Descent
    - Mini-batch Gradient Descent
- Learning Rate
  - Momentum
  - RMSProp
  - Adptive Momentum Estimation (Adam)

## 8.1 Regression

For different type of questions, we have different type of models and loss functions. Below is a short list that is covered in lecture.

- **Continuous Values**: Regression model, Mean Square Error loss.
- **Binary classification**: Logistic Regression, Cross Entropy loss.
- **Multiclass classification**: Softmax Classifier with Categorical Cross Entropy Loss.

### 8.1.1 Models

Linear Regression model:

$$y_i = wx_i + \beta$$

Where $y_i$ is the *predicted value*, $x_1$ is the *input value* and $\beta$ is the *bias term*.

Logistic Regression model: Same as Linear Regression model but with Sigmoid applied.

### 8.1.2 Activation Functions

$$\text{ReLU}(x) = \max(0, x)$$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\text{softmax}(\mathbf{z})_i \;=\; \frac{\exp\big(z_i\big)}{\sum_{j=1}^{K} \exp\big(z_j\big)}$$

### 8.1.3 Loss Functions

Mean Squared Error Loss:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} \big(y_i - \hat{y}_i\big)^2$$

Cross-Entropy (CE) Loss:

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{n} \sum_{i=1}^{n} \Big[ y_i \log\big(\hat{y}_i\big) \;+\; (1 - y_i) \log\big(1 - \hat{y}_i\big) \Big]$$

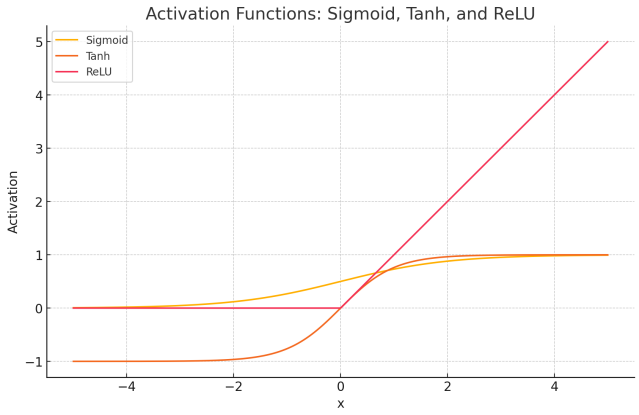



Figure 12: Sigmoid, Tanh, and ReLU Activation Functions.

$$\mathcal{L}_{\text{CCE}} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik}\ \log\big(\hat{y}_{ik}\big)$$

Where **BCE** stands for **Binary Cross Entropy**, and **CCE** stands for **Categorical Cross Entropy**. By the name, BCE is for binary classification, and CCE is for multiclass classification.

When using one hot encoding in multiclass classification, we have 1 only for the target class. Assume the target class is $\hat{k}$ for class $i$, then we can set only calculate the loss for $\hat{k}$ class for the other class have $y_{ik} = 0$. In other word, we should be able to set $k = \hat{k}$ for all $i$. Then we get the below equation for each example:

$$L_{\text{CCE}}(\,\hat{y},\, y) \;=\; -\frac{1}{N}\sum_{n=1}^{N} y_k \log\big(\hat{y}_k\big)_i$$

### 8.1.4 Softmax Classifier

With softmax classifier, the goal is to to multiclass classification. We use Softmax as activation function and CCE as loss function.

We want to find a function in function set that minimises total loss of all all training data. An other way to put this is to find the network parameters that minimises total loss. We define the total loss $L$ as:

$$L = \sum_{n=1}^{N} \ell^n$$

In softmax classifier, our objective for each training example $(x, y = j)$ is to

maximise the probability of the correct class $j$:

$$P(y = j \mid x) \;=\; \frac{\exp\big(W_j \cdot x\big)}{\sum_{c=1}^{C} \exp\big(W_c \cdot x\big)}$$

And the CE loss:

$$\ell(\,\hat{y},\, y) \;=\; -\sum_{i=1}^{C} y_i \,\ln\big(\hat{y}_i\big)$$

When we want to the set of parameter that will minimize the loss function

$$w^* = \arg\min_w L(w)$$

Where $w$ is the parameters and $L(w)$ is the loss function. We do greadient descent like:

$$w^1 \leftarrow w^0 \;-\; \eta \;\left.\frac{\mathrm{d}L}{\mathrm{d}w}\right|_{w=w^0}$$

This differentiation and then be simplified (steps in slides) to the equation below. Where $f_{w,b}$ is the activation function

$$w_i \;\leftarrow\; w_i \;-\; \eta \sum_{n=1}^{N} \Big[-\big(y^n - f_{w,b}\big(x^n\big)\big)\, x_i^n\Big]$$

## 8.2 Backpropagation

This notes will contain information form CS331.

The key here is to use chain rules.

> Given $y = f(x)$ and $z = g(y)$, we want to find $z_x$. By chain rule 1 (Univariate), this is $z_x = z_y \cdot y_x$. Or,
>
> $$\begin{aligned} &\Delta x \;\to\; \Delta y \;\to\; \Delta z, \\ &\frac{dz}{dx} = \frac{dz}{dy}\,\frac{dy}{dx} \end{aligned}$$

Here we have Chain Rule 2 (Multivariate):

$$\begin{aligned} z = f(x,y), \quad x = g(t), \quad y = h(t) \\ \implies \quad \frac{dz}{dt} &= \frac{\partial z}{\partial x}\,\frac{dx}{dt} + \frac{\partial z}{\partial y}\,\frac{dy}{dt} \\ &= z_x\, x_t + z_y\, y_t \end{aligned}$$

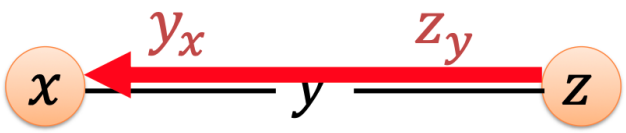



Figure 13: Chain Rule 1 (Univariate)

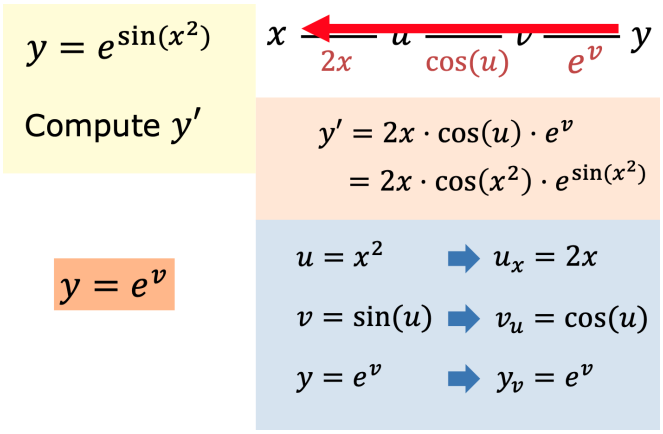



Figure 14: Calculation example of chain rule 1

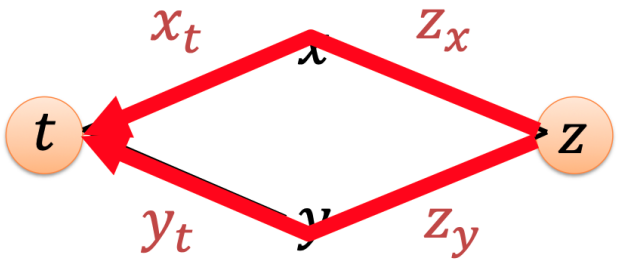



Figure 15: Chain Rule 2 (Multivariate)

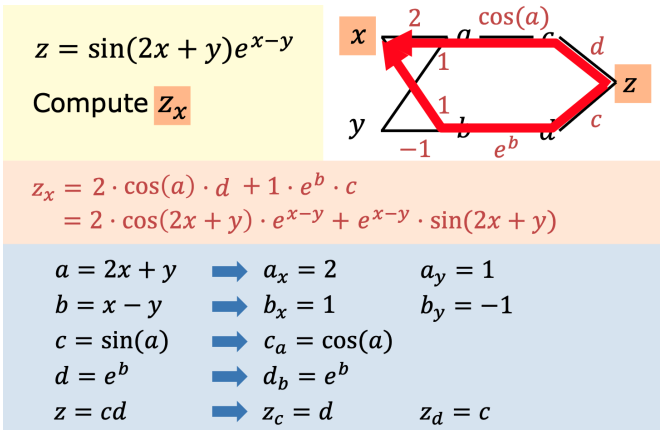



Figure 16: Calculation example of chain rule 2

## 8.3 Different Gradient Descents

We have $L$ is a loss function, $\theta$ are model parameters. The goal is to find $\theta^* = \arg\min_\theta L(\theta)$ We iterate $\theta^{t+1} = \theta^t - \eta^t \nabla L(\theta^t)$ where $\eta^t$ is the step-size, or learning rate.

Some basic initiation:

### 8.3.1 Stochastic Gradient Descent

In normal Gradient Descent, we get loss as the summation over all training examples, then perform one update. This method is slow.

$$\theta^{t+1} = \theta^t - \eta^t \nabla L(\theta^t),$$
$$L = \sum_n \left(\hat{y}^n - y^n\right)^2$$

For **Stochastic Gradient Descent**, we update for each training example $x^n$ and label $y^n$. In other words, we use loss for only one example. This can help to model to train faster.

$$\theta^{t+1} = \theta^t - \eta^t \nabla L^n(\theta^t),$$
$$L^n = \left(\hat{y}^n - y^n\right)^2$$

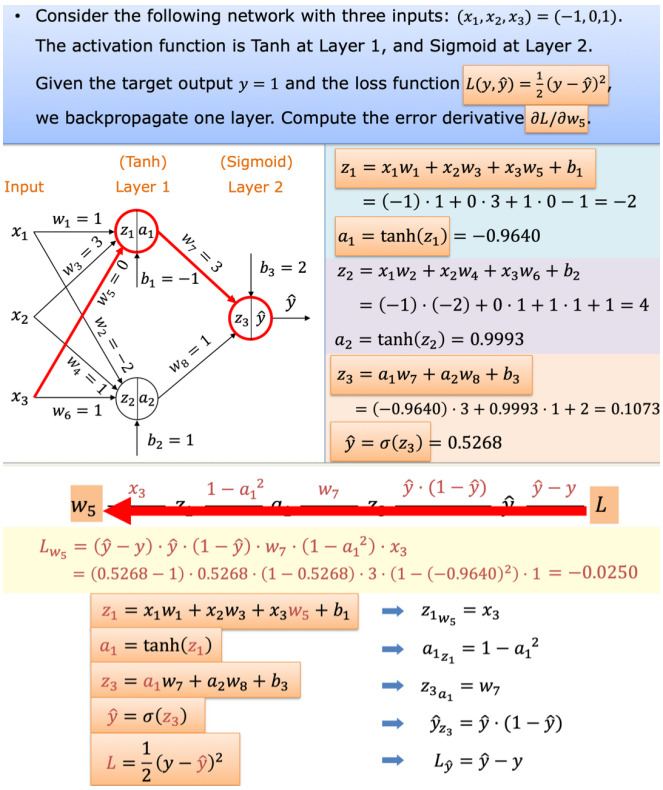



Figure 17: Best Backpropagation Calculation example of all time

**Mini-batch Gradient Descent** A combination of normal gradient descent and SGD. We update for every mini-batch of $m$ : $(1 < m < N)$ training example. Common mini-batch sizes range between 50 and 256. There are several advantages to this method:

- **Reduces the variance** of the parameter updates, which can lead to more **stable convergence**
- Can make **use of matrix optimisations** common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient.

### 8.3.2 Learning Rate

We need to set $\eta$ carefully. One simple **idea** is to reduce the learning rate by some factor every few epochs. At the beginning: we are far from the destination,

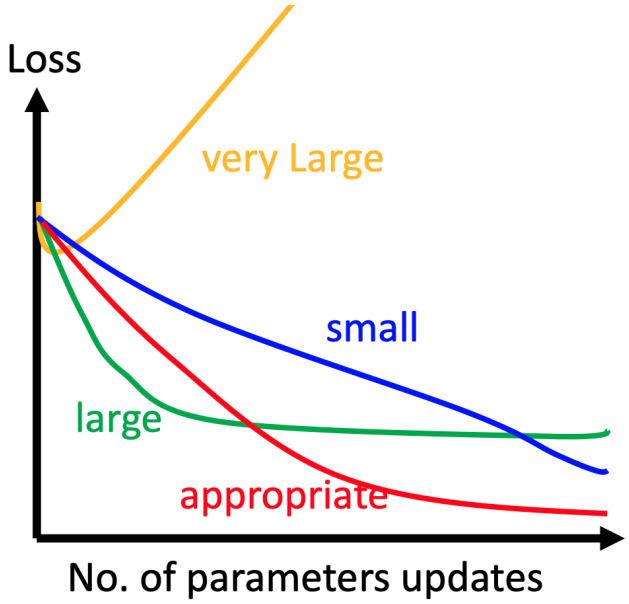



Figure 18: Effect of different learning rates

so we use larger learning rate. After several epochs, say $t$, we are close to the destination, so we reduce the learning rate.

$$\eta^t = \frac{\eta}{\sqrt{t+1}}$$

The simple solution may not be practical, we need different learning rates for different parameters.

**Momentum** The idea is to take previous movements into account. Movement not just based on gradient, but also on the previous movement.

We have Movement as:

$$v^t = \lambda\, v^{t-1} - \eta\, \nabla L\left(\theta^{t-1}\right).$$

And we update the parameters as:

$$\theta^t = \theta^{t-1} + v^t.$$

**RMSProp** RMSprop divides the *learning rate* by an *exponentially decaying average of squared gradients*. The **idea** is that we will have a decay rate and the decay rate entails that only recent gradient matters, and the ones from long ago are basically forgotten.

Typical value for $\gamma$ is 0.9.

$$\begin{aligned}
\sigma^0 &= \nabla L\big(\theta^0\big)\,, \\
\sigma^t &= \sqrt{\gamma\,(\sigma^{t-1})^2 + (1-\gamma)\,(\nabla L\big(\theta^{t-1}\big))^2}\,, \\
\theta^t &= \theta^{t-1} - \frac{\eta}{\sigma^{t-1}}\,\nabla L\big(\theta^{t-1}\big)\,.
\end{aligned}$$

**Adaptive Moment Estimation (Adam)** In addition to storing an exponentially decaying average of ***past squared gradients*** $v^t$, like RMSprop $\sigma^t$; Adam also keeps an **exponentially decaying average of past gradients** $m_t$, similar to momentum $v^t$. Adam gets the speed from momentum and the ability to adapt gradients in different directions from RMSProp.

$$\begin{aligned}
m_t &= \beta_1\, m_{t-1} + (1-\beta_1)\, g_t\,, \\
v_t &= \beta_2\, v_{t-1} + (1-\beta_2)\, g_t^2\,.
\end{aligned}$$

Here we have $m_t$, and $v_t$, are estimates of the **first moment (the mean)** and the **second moment (the uncentered variance)** of the gradients respectively.

$g_t$ Should be the derivative of loss function for current parameters, $\nabla L\big(\theta^t\big)$. At time step 0, we will have moment of 0. So we use the $\hat{m}_t$ to mitigate this issue. Same as $v_t$.

$$\begin{aligned}
\hat{m}_t &= \frac{m_t}{1-\beta_1^t}\,, \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t}\,, \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\,\hat{m}_t\,.
\end{aligned}$$

In Figure 19 we have the full Adam algorithm, where the definition should be more clear.

# 9 Recurrent Neural Network (RNN)

**Algorithm 1:** *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. $g_t^2$ indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With $\beta_1^t$ and $\beta_2^t$ we denote $\beta_1$ and $\beta_2$ to the power $t$.

**Require:** $\alpha$: Stepsize
**Require:** $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
**Require:** $f(\theta)$: Stochastic objective function with parameters $\theta$
**Require:** $\theta_0$: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector) → for momentum
$v_0 \leftarrow 0$ (Initialize 2nd moment vector) → for RMSprop
$t \leftarrow 0$ (Initialize timestep)
**while** $\theta_t$ not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
**end while**
**return** $\theta_t$ (Resulting parameters)

Figure 19: Full Adam algorithm. Has all the definitions

- Problems with RNN
  - Vanishing/Exploding Gradients
- Long Short Term Memory (LSTM)
  - Forget Gate
  - Input Gate
  - Cell Update
  - Output Gate
  - Gated Recurrent Unit (GRU)
- Stacked (Multilayer) RNN
- Bidirectional RNN

One problem with normal Feed-Forward Neural Network is that it does not preserve information about the order of text, and not consider text with different lengths (when we are using BoW).

In Figure 20 we can see that there are two inputs to the hidden layer, input $x$ and the output form the hidden layer from the previous time step $h_{t-1}$.

Say Part of Speech, we can have a **many to many model** where the output for each input is valid, and the output is the tag for PoS for each input word. Or we can have a **many to one** model where only after all input are calculated, then output one final answer.

To do back optimisation, we get the loss for all loss of all $\hat{y}_t$ ie, all outputs. See

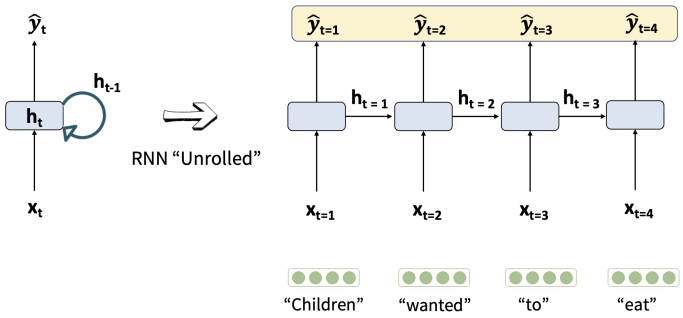



Figure 20: Example of RNN

in Figure 21, we will have loss of:

$$L = -\frac{1}{T}\sum_{n=1}^{N} L_{\text{CCE}}$$

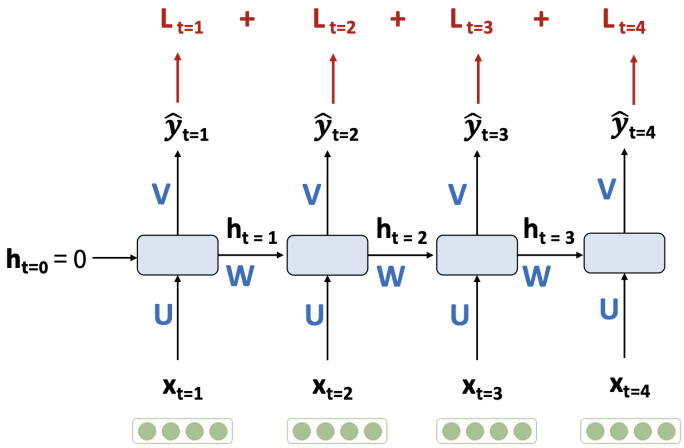



Figure 21: Backpropagation Through Time (BPTT)

For backpropagation, we need:

$$\frac{\partial L}{\partial V}, \frac{\partial L}{\partial W}, \frac{\partial L}{\partial U}.$$

Take $V$ as example, we have:

$$
\begin{aligned}
\frac{\partial L}{\partial V} &= \sum_{t=1}^{T} \frac{\partial L_t}{\partial V}\,, \\
\frac{\partial L_t}{\partial V} &= \frac{\partial L_t}{\partial \hat{y}_t}\,\frac{\partial \hat{y}_t}{\partial q_t}\,\frac{\partial q_t}{\partial V}\,, \qquad q_t = V\,h_t\,, \\
\frac{\partial L_t}{\partial W} &= \frac{\partial L_t}{\partial \hat{y}_t}\,\frac{\partial \hat{y}_t}{\partial q_t}\,\frac{\partial q_t}{\partial h_t}\,\frac{\partial h_t}{\partial W}\,, \\
\frac{\partial h_t}{\partial W} &= \frac{\partial h_t}{\partial W} + \frac{\partial h_t}{\partial h_{t-1}}\,\frac{\partial h_{t-1}}{\partial W} + \frac{\partial h_t}{\partial h_{t-1}}\,\frac{\partial h_{t-1}}{\partial h_{t-2}}\,\frac{\partial h_{t-2}}{\partial W} + \cdots\,.
\end{aligned}
$$

The above calculation is saying that we need the gradent of loss for each $t$, which can be calculated as $L_t -> \hat{y}_t -> q_t -> V$. For we have $q_t = V\,h_t$, then we have $L_t -> \hat{y}_t -> q_t -> h_t -> W$. The last statement mean that inorder to find $h_t -> W$, we need to accumulate all $h_t$ for all previous steps. Remember $h_t$ is the input to the hidden layer of past results. In same way we can find $U$.

Note that if $W$ has $eigenvalues < 1$, repeated multiplication leads to exponentially smaller gradients.

## 9.1 Problems with RNN

We can see form figure22 that the error surface is rough. This can cause the model be very hard to train. The steep jumps in the loss function's 3D space can make the point not differentiable. We can apply clipping to control the gradient at each step.

An other problem is earlier signal may weaken over time, as multiple multiplication happened. Note this is different from Vanishing Gradients Problem.

**Vanishing/Exploding Gradients** Computing gradient of $h_t$ involves many factors of $W$ and repeated $tanh$ (activattion function). When $W$ is large, lead to **Exploding gradients**. The solution is **Gradient clipping**, we scale gradient if its norm is too big. When $W$ is small, we have **Vanishing gradients**. In this case, there is no better way than change RNN architecture.

Since RNNs use the same weights for each iteration, it will also have the same effect on the input every time. If the effect is increasing the values, the values can easily become very big and if the effect is decreasing the values, they become small. RNNs often solve exploding gradients by using gradient clipping, and vanishing gradients by changing the RNN architecture, for example, using LSTM or GRU.

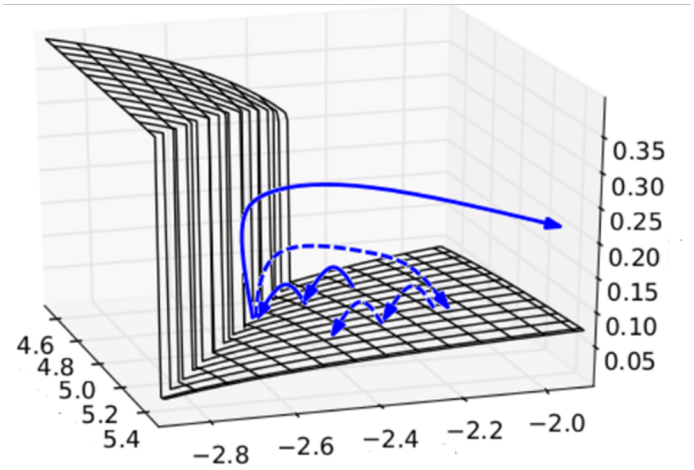



Figure 22: The error surface is either very flat or very steep.

## 9.2 Long Short Term Memory (LSTM)

The main **idea** is to have some specific **memory**, and we and **delete**, **preserve**, and **add** new information into the **memory**. This memory is just a new vector, which is usually called the **cell state**.

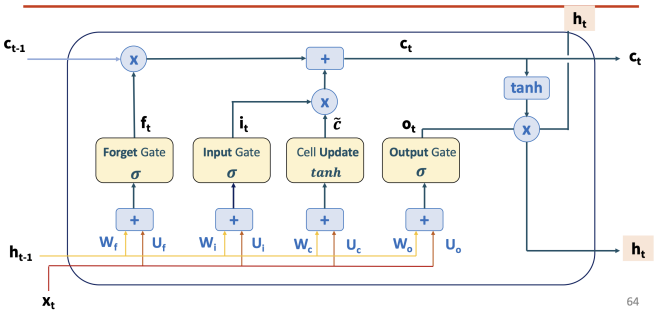



Figure 23: LSTM Block

The Figure 23 show the structure of a LSTM block. The *activation functions* in the gates act like a masks to remove, add information. There are four gates in total. The + and × should represent element wise operations.

**Forget Gate** The purpose is to remove information from $c_{t-1}$.

$$f_t = \sigma\big(U_f x_t + W_f h_{t-1}\big)$$

**Input Gate** Control what part of the input should be put into memory, sometimes it's all, some times it's not.

$$i_t = \sigma\big(U_i x_t + W_i h_{t-1}\big)$$

**Cell Update** How much should I update the memory. This is what I want from my input, and how much should it fit into the memory, much much should I substitute with the memory I already have. Given the information I have, how much should I put into the input. This Gate work together with **Input Gate**.

$$\tilde{c} = \tanh\big(U_c x_t + W_c h_{t-1}\big)$$

**Output Gate** The output now consider long term information from the cell state.

$$h_t = o_t \odot \tanh\big(c_t\big)$$

And below is how to represent LSTM in matrices:

$$\boxed{h_t \;=\; \tanh\big(W\,[\,h_{t-1},\,x_t\,]\big)}$$

$$\boxed{\begin{aligned}\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} &= \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} \big(W\,[\,h_{t-1},\,x_t\,]\big), \\ c_t &= f \odot c_{t-1} + i \odot g\,, \\ h_t &= o \odot \tanh(c_t)\,. \end{aligned}}$$

**Gated Recurrent Unit (GRU)** It is a Variant to LSTM, where it **Combines** the **forget** and **input** gates into a single **update gate**, **Merges** the **cell state** and **hidden state**, result in a single **hidden state**.

This model train and should run faster than traditional LSTM as there is less compute, and it perfrom better than LSTM.

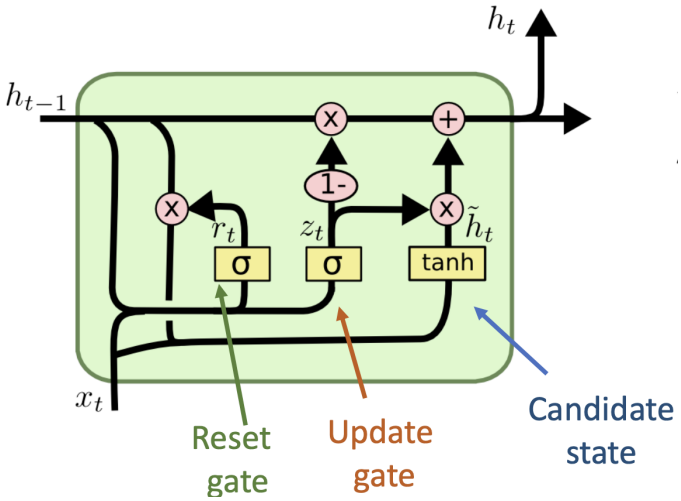



Figure 24: Gated Recurrent Unit

## 9.3 Stacked (Multilayer) RNN

We can stack RNN on top of each other. With differnet level of the model, different levels of representations are learned. More **low level**, more **fine grained** and more **high level**, more **abstract representations**. So the **idea** is that The RNN on **top** will put together the information from RNN form the bottom.

This kind of model take a long time to train as BPTT has a lot of parameters.

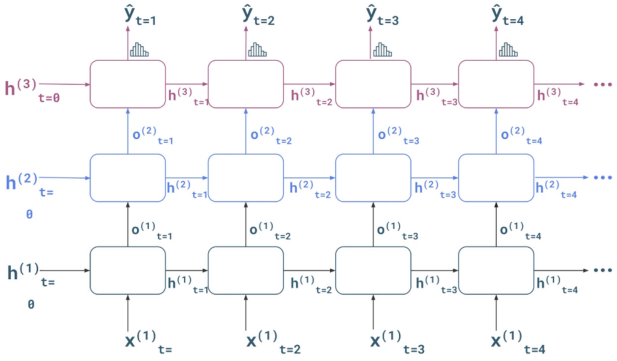


Figure 25: Stacked RNN Example

### 9.4 Bidirectional RNN

So we have a RNN to the right and one to the left, result in two set of $h_t$s. At the end, just concatenate the two hidden states.

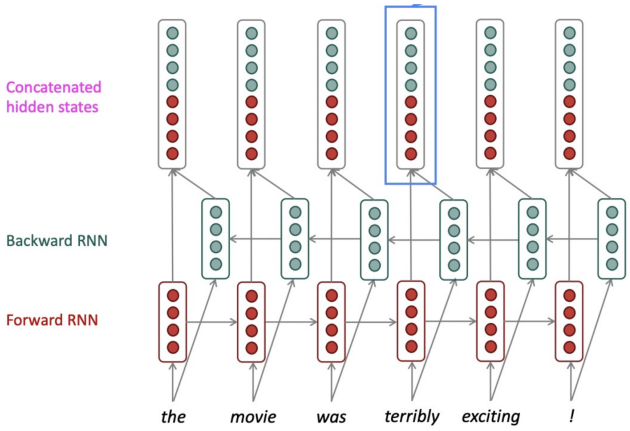



Figure 26: Bidirectional RNN

## 10 Seq-to-Seq Learning with Attention

- Seq2Seq Learning Architecture
- Attention Mechanism
  - Formal definition of Attention in Seq2seq
- General Attention
  - Basic Dot-Product Attention
  - Multiplicative Attention
  - Additive Attention
- Advanced Attention
  - Self-attention
  - Attention as Information Retrieval

For Seq-to-Seq models, we are aiming for the **many to many** model in Figure 27. For NLP, different languages tend to have different length to express the same meaning, and there can be different length of sequence in the same

language to express the same meaning. So this architecture is very useful for NLP.

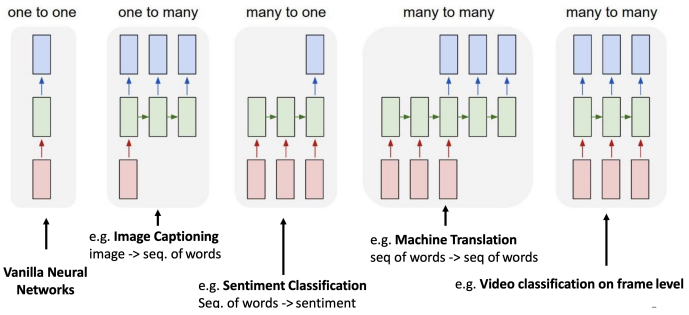



Figure 27: Process Sequences of Recurrent Neural Networks

## 10.1 Seq2Seq Learning Architecture

There are two parts to the Seq2Seq model, one **encoder** and one **decoder**. The **encoder** sort of encode the input, put the input into an other code. The **decoder** take the information form the encoder as some code, and modifying them back to our original language, to our original code.

The encoder and decoder typically involves two Recurrent Neural Networks (RNNs), also can be different RNN from each other. The decoder and encoder can be used separately, result in some decoder or encoder models.

> Note from Figure 28 that in decoder, the genreated sequence is one word ahead from the input to the decoder. At **training** time, we need to give the correct translation to the model so the model knows what's the target. And at **test** time, the decoder output is fed in as next step's input.

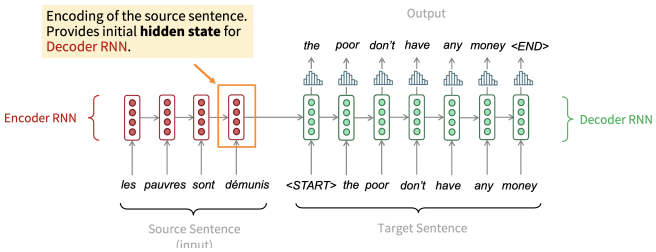



Figure 28: Neural Machine Translation demo with seq2seq Mode

During optimisation, we optimise as a single system; backpropagation operates "end to end". Which means we calculate and sum all the loss at each step of text generation, then use this loss to back-propagate the whole network, considering the input and output at each step.

We have $J_t$ as the loss at step $t$ of the decoder. Form Figure 28, $J_1$ is the loss at the neuron generating the word "the" and $J_2$ is the loss at the neuron generating the word "poor". And this $J_1$ is the negative log probability of "the". And $J$ should be the total loss.

$$J = \frac{1}{T} \sum_{t=1}^{T} J_t$$

## 10.2 Attention Mechanism

The Attention Mechanism is the solution to solve Weak Signal problem as we can see in Figure 29. This lead to the The Bottleneck Problem, which suggest that the last neuron in the encoder needs to capture all information about the source sentence from previous neurons. If the input text become big, this become a hard problem.

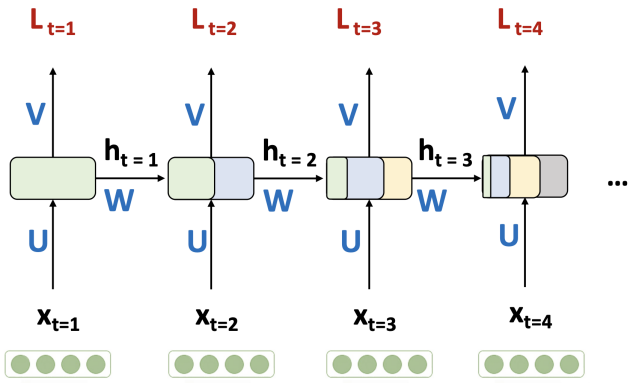



Figure 29: Earlier signal may weaken over time

The **idea** for attention is to directly access neurons at encoder from decoder, so information in earlier steps will not be weaker. In an other words, on each step of the decoder, focus on a particular part of the source sequence.

Attention is great in many reasons:

- Attention significantly improves Neural Machine Translation performance.
  - It's very useful to allow decoder to focus on certain parts of the source
- Attention solves the bottleneck problem.
  - Attention allows decoder to look directly at source; bypass bottleneck.
- Attention helps with vanishing gradient problem
  - Provides shortcut to faraway states
- Attention provides some interpretability. By inspecting attention distribution, we can see what the decoder was focusing on.
  - We get alignment for free as the network just learned alignment by itself.

So the **attention score** is calculated as the **dot product** (or something else like sum) between the state in decoder and all the states in the encoder, as shown in Figure 30.

Here in Figure 31, that we obtain **Attention distribution** from all the **Attention scores** then apply Softmax.

In Figure 32, we use the **Attention distribution**, applied to the **Attention scores**, then multiply the **Attention scores** with **original encoder states** to get a **weighted sum** of all the vectors in the encoder hidden states.

The **weighted sum** of all the vectors in the encoder hidden states with the **Attention distribution** and **Attention scores** is called the **Attention output**. The most relevant vector, or the hidden state in the encoder, the one with the highest **Attention score** are going to going to contribute more to my final vector, the **Attention output**.

In Figure 33, the last step, we concatenate the **Attention output** with hidden state form the decoder, then use this new vector to genterate the output.

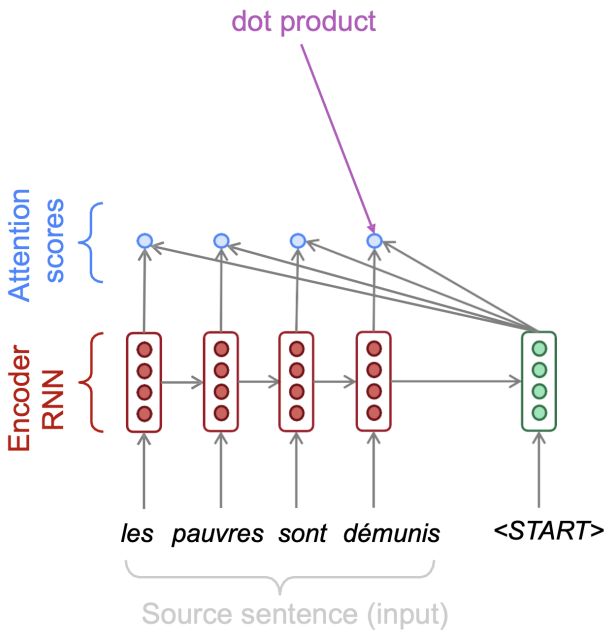



Figure 30: Attention Score Calculation

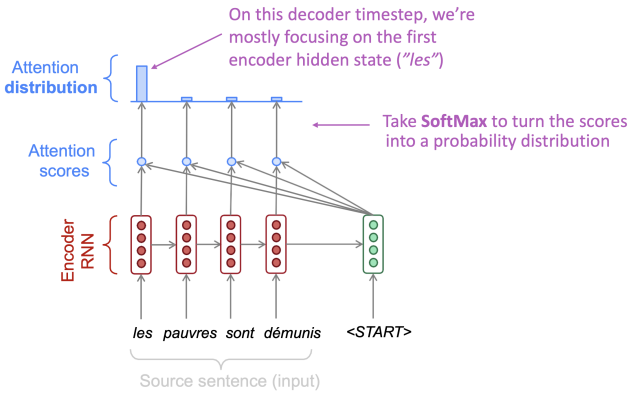



Figure 31: Softmax on Attention Score

**Formal definition of Attention in Seq2seq** Note that this definition is just for Seq2seq models.

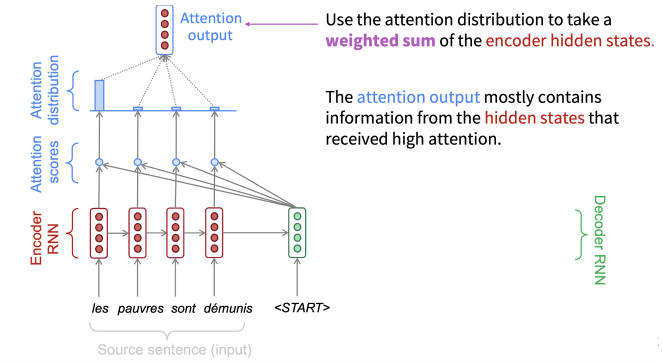



Figure 32: Attention distribution as weighted sum of the encoder hidden states.

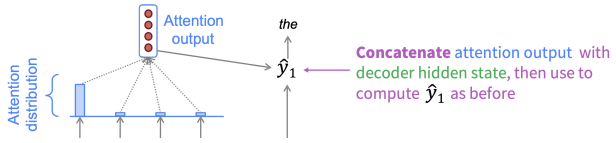



Figure 33: Concatenate attention output with decoder hidden state.

In Figure 34, we have the notation we need.

We have the **encoder hidden states**: $h_1, \ldots, h_N \in \mathbb{R}^h$. On timestep $t$, we have **decoder hidden state** $s_t \in \mathbb{R}^h$. We get the attention scores $e^t$ for this step, note $s_t^\top h_1$ is **dot product**.

$$e^t = \left[s_t^\top h_1, \ \ldots, \ s_t^\top h_N\right] \ \in \ \mathbb{R}^N$$

We take softmax to get the **attention distribution** $\alpha_t$ for this step (this is a probability distribution and sums to 1)

$$\alpha_t = \text{softmax}\left(e^t\right) \ \in \ \mathbb{R}^N$$

We use $a_t$ to take a weighted sum of the encoder hidden states to get the **attention output** $a^t$:

$$a^t = \sum_{i=1}^{N} \alpha_{t,i} \, h_i \ \in \ \mathbb{R}^h$$

Finally we concatenate the attention output $a^t$ with the decoder hidden state $s_t$ and proceed as in the non-attention seq2seq model:

$$\left[a^t;\, s_t\right] \in \mathbb{R}^{2h}$$

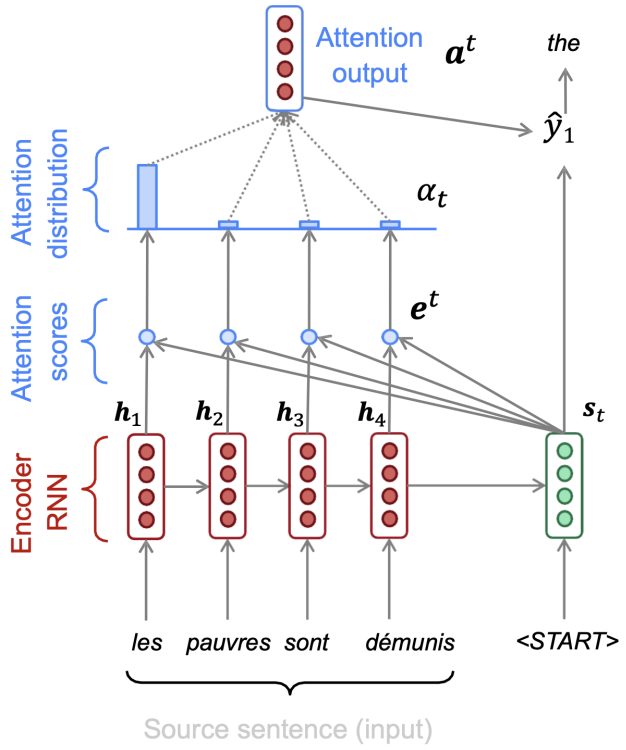



Figure 34: Formal definition of Attention in its notations.

## 10.3 General Attention

Note that in previous section the definition is just for Seq2seq models.

The **idea** here is that 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).

More general definition of 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. We sometimes say that the ***query attends to the values***.
- E.g., in the seq2seq + attention model, each *decoder hidden state* **attends** to the **encoder hidden states**. So *decoder hidden state* is the **query** and **encoder hidden states** are the **values**.

Here we have the formal definition of General Attention:

- We have some values $h_1, \ldots, h_N \in \mathbb{R}^{d_1}$ and a query $s \in \mathbb{R}^{d_2}$.
- Attention always involves computing:
  - The **attention output** $a \in \mathbb{R}^{d_1}$ (also called the context vector)
  - From the **attention scores** (Figure 35) $e \in \mathbb{R}^N$ (or attention logits):

$$\text{score}(h_k, s) = h_k^T s \quad \in \mathbb{R}$$

$$e = \left[\text{score}(h_1, s), \ldots, \text{score}(h_N, s)\right] \in \mathbb{R}^N$$

$$\alpha = \text{softmax}(e) \in \mathbb{R}^N$$

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

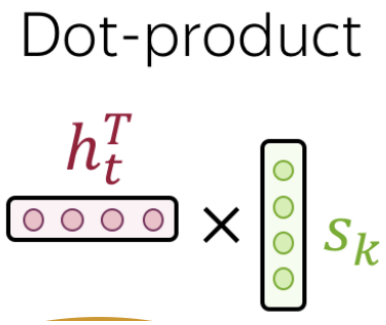



Figure 35: Dot product of *values* and **query**.

There are serval ways to compute score$(h_1, s)$.

**Basic Dot-Product Attention** The **idea** is to directly computes the similarity between the hidden state and the query using a dot product.

The assumption here is that $d_1 = d_2$ (remember $h_1, \ldots, h_N \in \mathbb{R}^{d_1}$ and $s \in \mathbb{R}^{d_2}$.)

$$e_i = s^\top h_i \in \mathbb{R}$$

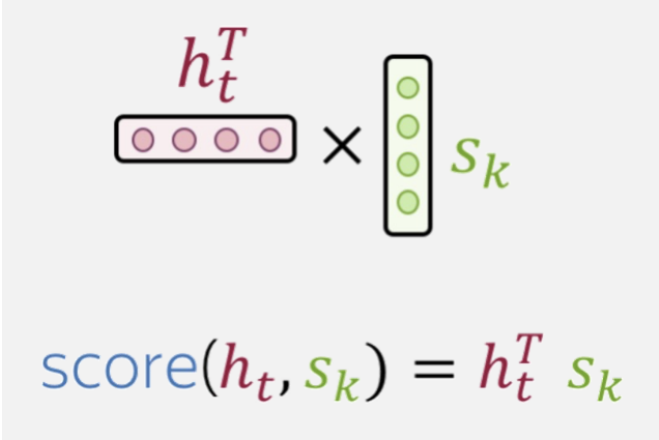



Figure 36: Dot-Product Attention

**Multiplicative Attention** The **idea** is that it learns how to compare vectors better by scaling and rotating them.

We have $W \in \mathbb{R}^{d_2 \times d_1}$ as the weight matrix. So it is introducing an intermediate layer between the vector form decoder and encoder. It allows slight modification of the vectors before comparing them.

The **idea** is that the comparison might be more effective if I slightly modify the vectors.

$$e_i = s^\top W h_i \in \mathbb{R}$$

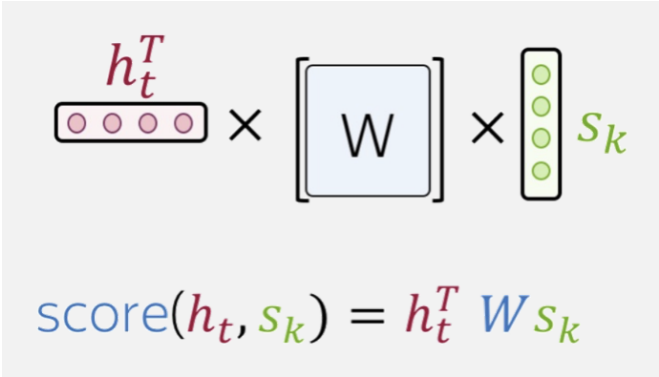



Figure 37: Multiplicative Attention

**Additive Attention** The **idea** is to instead learns a function to compare vectors, in this case a simple linear model.

Where $W_1 \in \mathbb{R}^{d_3 \times d_1}$, $W_2 \in \mathbb{R}^{d_3 \times d_2}$ are weight matrices and $v \in \mathbb{R}^{d_3}$ is a weight vector.

From Figure 38, the model in there is learning how to compare the decoder vector and the encoder hidden states.

$$e_i = v^\top \tanh\big(W_1 h_i + W_2 s\big) \in \mathbb{R}$$

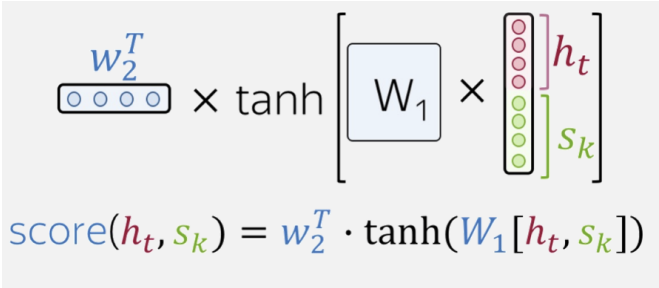



Figure 38: Additive Attention

## 10.4 Advanced Attention

Not sure if this is a Seq2seq model problem but it should be a decoder problem. When the input is really long, like for summarisation, that decoder might just repeat itself. Caused by the decoder find some sentence it really like and repeat it self and not able to break out form the loop.

There are two ways to fix this, **Self-attention** (intra-decoder attention) and **Attention as Information Retrieval** ( Pointer Network).

**Self-attention** Self-attention: to generate $y_t$ we need to pay attention to $y_{<t}$ Like shown in Figure 39, The hidden states in the decoder attend to themselves.

Here we show the calculation for Self-attention (attention between hidden states in the decoder)

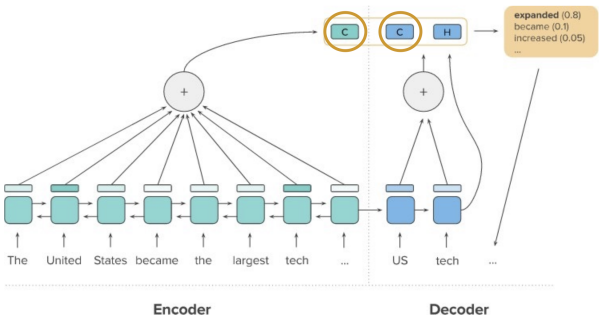



Figure 39: Self-Attention

On step $t$, $h_t^d$ attends to previous decoder hidden states $h_{t'}^d$:

$$e_{tt'}^d = \left(h_t^d\right)^\top W_{\text{attn}}^d \, h_{t'}^d \quad \text{for } t' = 1, \ldots, t-1$$

Apply Softmax to get **attention distribution** over previous hidden states $h_{t'}^d$:

$$\alpha_{tt'}^d = \frac{\exp\left(e_{tt'}^d\right)}{\sum_{j=1}^{t-1} \exp\left(e_{tj}^d\right)} \quad \text{for } t' = 1, \ldots, t-1$$

Compute decoder **attention output**:

$$c_t^d = \sum_{j=1}^{t-1} \alpha_{tj}^d \, h_j^d \in \mathbb{R}^{d_h}$$

By Figure 39, it looks like the **attention output from decoder** is combined with **attention output from encoder** with the **hidden state we are currently on in decoder**, then this vector is used to calculate the output.

**Attention as Information Retrieval** The description below should be done under the assumption for translation task.

So the **idea** here is to use a real dictionary for word check. The idea came form the fact that with attention, we also have a query (hidden states in decoder) and values (all hidden states in encoder), which is like a python

dictionary but the value it returns is a blend of all values (attention output), based on the query (This kind of dictionary should be called Fuzzy Look-up Dictionary).

By reference of Figure 40, we see that the only difference is the added **Vocabulary Distribution**. Not sure how this is been constructed. no idea why mentioned self attention in lecture.

It uses a **pointer network**, which can point to specific elements in the input to be all to be copied almost verbatim, almost exactly as they are. So we can copy words from the input.

The $P_{gen}$ uses a *Sigmoid* function that return number between $[0, 1]$, so if it is $> 0$, then we enhance the effect to one side of the architecture.

Compute **probability of copying/pointing** to word from input:

$$p\big(u_t = 1\big) = \sigma\Big(W_u\,[\,h_t^d;\, c_t^e;\, c_t^d\,] + b_u\Big)$$

If not **copying/pointing**, use standard softmax:

$$p\big(y_t \mid u_t = 0\big) = \text{softmax}\Big(W_{\text{out}}\,[\,h_t^d;\, c_t^e;\, c_t^d\,] + b_{\text{out}}\Big)$$

If **pointing**, use encoder attention weights (Note this should just be the attention output for the encoder):

$$p\big(y_t = x_i \mid u_t = 1\big) = \alpha_{ti}^e$$

Combine everything:

$$p(y_t) = p(u_t = 1)\,p(y_t \mid u_t = 1) \;+\; p(u_t = 0)\,p(y_t \mid u_t = 0).$$

The result (final distribution) should be a combined distribution of the dictionary and the attention distribution form the encoder. The weights to each of the distribution should be decided by $P_{gen}$.

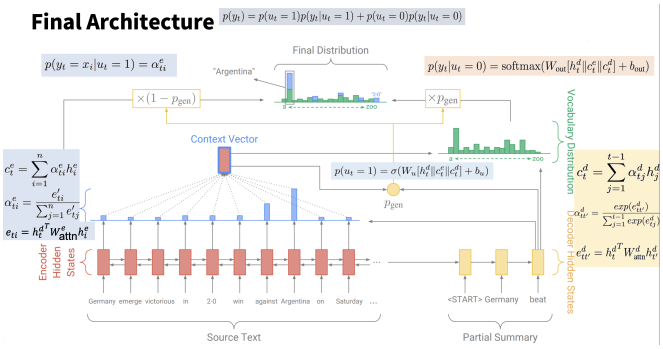



Figure 40: Attention as Information Retrieval Full Architecture

# 11 Transformers

- 

This section is about the original transformer. There are three main blocks of transformer, the ***Input Tokenization***, the ***Encoder*** and the ***Decoder***.

## 11.1 Input Tokenization

The original tokenization has mainly two methods, at word level or at character level. We see the good and the bad about these methods in Table 8.

### 11.1.1 Subword Tokenization

Some better tokenization methods should be used, like **Subword Tokenization**.

> The idea here is to keep frequent words, and break rarer words into subwords, via a statistical algorithm based on the corpus.

The result embedding from this tokenization method will be different for the same word under different corpus, which is different from previous methods like **Word2Vec** and **Glove**.

We can see from Figure 41 that "pizzeria" is been split into three parts, and these subwords become part of the tokenizer's vocabulary. Subword tokenizer has a better chance to deal with **OOV** words while **reducing the vocabulary size** and **maintaining the performance**.

| Word-Level | Char-Level |
|---|---|
| **Good:**<br>• Intuitive<br>**Bad:**<br>• Doesn't handle OOV words (including new words)<br>• Slang, word-play, misspellings, etc. are problematic<br>• Huge vocabulary for large corpora<br>• Punctuation handling is challenging. Like "don't", "N.Y.C." | **Good:**<br>• Reduced memory footprint<br>• Naturally handles OOV tokens<br>**Bad:**<br>• Requires modeling many char-sequence statistics<br>• Often lower performance on "semantic" tasks |

Table 8: Comparison of word-level vs. character-level modeling

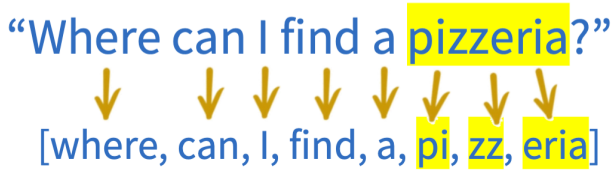



Figure 41: Subword Tokenization

**Byte-Pair Encoding** One of the biggest advantage of subword tokenizer model can learn the **common roles of subwords**, like *eria*: indicating an "area" for food; like pizzeria, churroreria, gelateria, etc..

The **Byte-Pair Encoding** use this idea, and ***build tokens based on pattern frequencies***.

Let our entire corpus consist of the single sentence:

"She sells seashells by the seashore."

After basic whitespace tokenization (and appending an end-of-word marker "_", we obtain the sequence of word-tokens:

$$\big[\texttt{she\_},\ \texttt{sells\_},\ \texttt{seashells\_},\ \texttt{by\_},\ \texttt{the\_},\ \texttt{seashore\_}\big].$$

We now build our vocabulary from the list above. We start from all the single letters, then it find the most **frequent adjacent vocabulary members** (the letters), starting with two adjacent letters. The most adjacent members in this case are $[\texttt{sh\_}, \texttt{he\_}, \texttt{e\_}\}$. We then Add the most frequent pairs to the vocabulary one by one, each time we add one we update the vocabulary. It is experimental what the $N$ should be.

**Vocabulary**

_, a, b, e, h, l, o, r, s, t, y
_, a, b, e, h, l, o, r, s, t, y, sh
_, a, b, e, h, l, o, r, s, t, y, sh, he
_, a, b, e, h, l, o, r, s, t, y, sh, he, e_, se
_, a, b, e, h, l, o, r, s, t, y, sh, he, e_, se, she
⋮

Continue until $N$ merges are performed.

These are all variant of the Byte-Pair Encoding, which is originally form Google and is private.

- Byte-Pair Encoding
- Wordpiece (used by BERT, for example)
- Sentencepiece

The criteria to determine a good $N$ can be:

- Probabilistic criteria (Intrinsic evaluation), the ability to predict what are the next few words, how many words are outside the range of compositions.
- Downstream approach (Extrinsic evaluation), we look at actual NLP task performance like sentiment analysis.

The final **Input Embedding** will be 512-dimensional. This should be calculated in similar way as for Word2Vec.

## 11.2 Encoder

The main issue with the Seq2seq model is **The Bottleneck Problem**. We need to capture all information about the source sentence before we can pass

the information into the decoder. There is sequential dependency dependency of calculating the input to decoder, computing this vector requires **all the previous timestamps to be computed first**.

By **Attention** (or self-attention), we can remove this recurrence, make it possible to parallel the computation and improve performance at the same time. By self-attention, with $N$ inputs, we should have $N$ outputs where each output depends on all the inputs via attention.

**Simple Self-Attention** For three inputs, the process for calculating Simple Self-Attention for $x_1$ is shown in Figure 42, basically the same calculation used for regular attention. The result vector is in same shape as the input $x_1$ and it is the updated embedding for $x_1$, which now include information from all other embeddings. We do this for all out inputs.

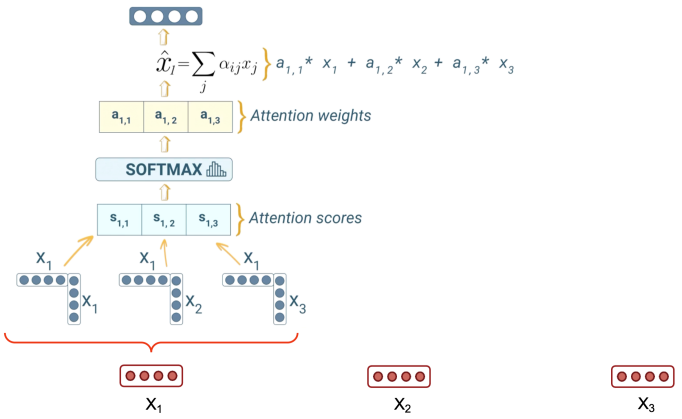



Figure 42: Simple Self-Attention

Now every output is based on all inputs, no need for sequential processing. Therefore, all the computations can be performed in parallel.

In Figure 43 we show how to calculate Simple Self-Attention for all input at the same time. $X$ is $x_1$, $x_2$ and $x_3$ stacked together. Each updated embedding (the yellow, blue and green vector) now includes information from all the other embeddings ($x_1$, $x_2$ and $x_3$).

Because the embedding into the Simple Self-Attention and out from Simple Self-Attention are in same dimensionality, we can stacke multiple Simple Self-Attention layers together.

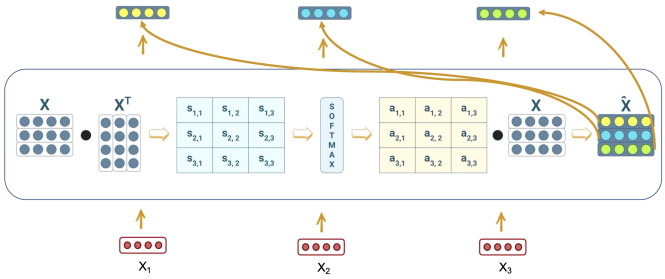



Figure 43: Parallel computation of Simple Self-Attention

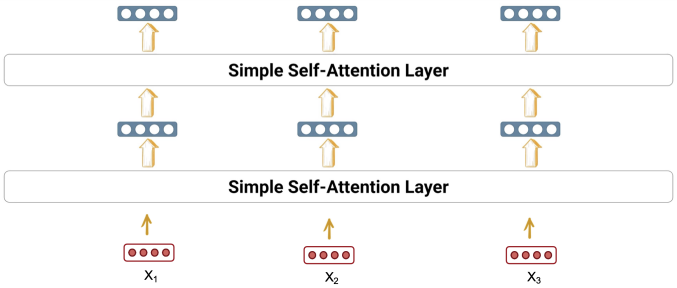



Figure 44: Example of stacked Simple Self-Attention layers

**Scaled Self-Attention** The **idea** here is similar to what we used for **Multiplicative Attention** in paragraph 10.3. We slightly modify the vector before dot product for attention. This step actually allow better attention be calculated. No idea why though.

> We have three sets of $W$ for each input embedding $x_n$, $W^q$, $W^k$ and $W^v$, for **query**, **key** and **value**.

The calculation for Scaled Self-Attention is shown in Figure 45. $q_n$ is for *query* for $x_n$, and $k_n$ is for *key* for $x_n$, and $v_n$ is *value* $x_n$.

Note, $d_k$ is the scaled attention score by key size or key's dimension. In Lihongyi's lecture, $d$ is used as the scaled factor, the dimension of $q$ and $k$. No idea why.

Figure 46 calculation of Scaled Self-Attention in matrix, where parallelization is been done.

> This the mathematical formulation of matrix calculation of Scaled Self-Attention. Take reference from Figure 46, it should be easy to understand

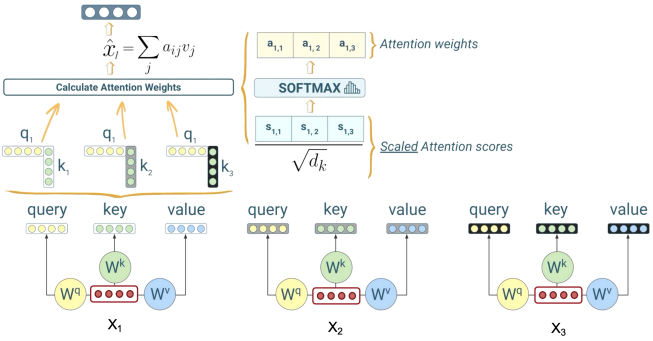



Figure 45: Example of Scaled Self-Attention

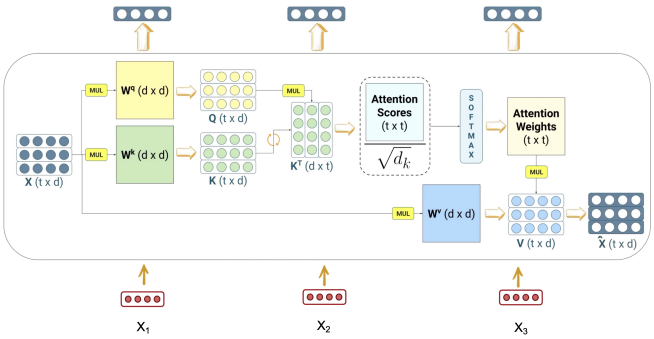



Figure 46: Parralel computations for Scaled Self-Attention

this equation. This is also called a **Single Head Attention**.

$$\text{Attention}(Q, K, V) \;=\; \text{softmax}\left(\frac{Q\,K^{\top}}{\sqrt{d_k}}\right) V$$

We can expand this single headed attention to a **multi-head attention**, see in Figure 47. Here we use three multi-head attention heads, and the input dimension is 12. Note that each head operates in a smaller dimension of $\frac{d}{h}$ (4 dimension for each head in this case), the overall computation across multiple heads isn't higher than single-head attention. For the output can be concatenated together back to original shape, so we can also stack this head. Each attention head can run in parallel, and long distance relationships are

encoded with $O(1)$ operation. In practice, there is no need to maintain separate sets of weights for each head.

One question is that how to deicide which dimension for each head to process? From Lihongyi's lecture that this description may not be accurate. See in Figure 48, we just split the original query into two, which is equivalent to what we have before. Where $W^{q,1}, W^{q,2}$ are just two different vectors.

Note that when we concatenate $b^{i,1}, b^{i,2}$ (attention output) just before final output, it maybe that the output dimension is not the same as input. So we just do a simple transformation:$b^i = W^o \begin{bmatrix} b^{i,1} \\ b^{i,2} \end{bmatrix}$. The $W^o$ should just be a transform matrix.

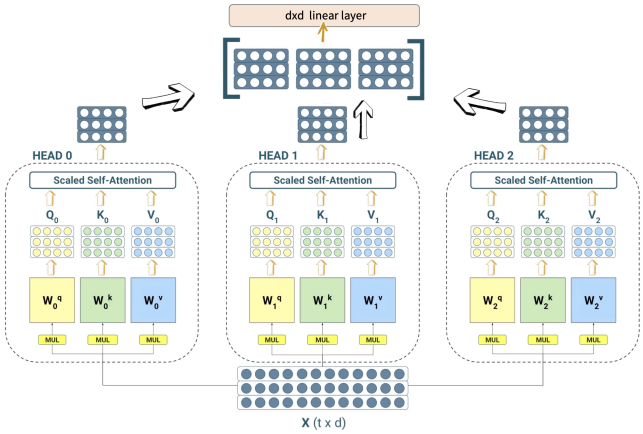



Figure 47: Multi-Head Self-Attention

In original Transformer, Embedding dimension is 512 (also called model dimension $d_{model}$. There is also 8 attention heads, so $\frac{d}{h} = 64$.

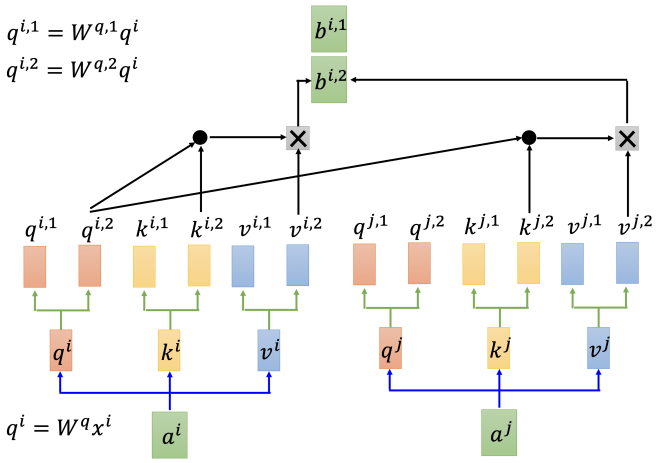



Figure 48: Lihongyi's multi-head attention

However, **we lost the word order information**. To solve this, we use some additional variable called the **Positional Embedding**. This embedding is element wise added to the input embedding before it is feed into the encoder block.

Note this Positional Embedding need to be the same shape as input embedding in order to perform adding.

Also Note that this embedding is not learned from data, it was hand crafted from the original transformer paper. For each input embedding ($x_i$ from lecture and $a^i$ form Lihongyi), there is a $e^i$ and by looking at this $e^i$, we can tell which position is the embedding at.

You might think if we do the matrix sum you might loss the position information, instead we could be better than do a concatenation. The explanation by Lihongyi is below:

Assume we have $p^i$ as a one hot encoding with position information (If $a^i$ is

in position $i$, then $p^i$ will have the $ith$ dimension be 1).

$$p^i = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \quad \text{with } p^i_j = \begin{cases} 1, & j = i, \\ 0, & j \neq i. \end{cases}$$

We concatenate $x^i$ and $p^i$, then multiply by $W$:

$$W = \begin{bmatrix} W^I \mid W^P \end{bmatrix}$$

$$W \begin{bmatrix} x^i \\ p^i \end{bmatrix} = W^I x^i + W^P p^i = a^i + e^i.$$

We see from above that the result is equivalent to $a^i + e^i$, for:

$$a^i = W^I x^i, \qquad e^i = W^P p^i.$$

A graph representation of above is in Figure 49. From here, we see that the positional embedding is actually trying to set the $W^P$, Lihongli mentioned that there has been approach to learn this $W^P$ but the result is not good, there maybe way to learn this now.

In the paper, $W^P$ was calculated by a werid equation and there is actually a visualisation in Figure 50.

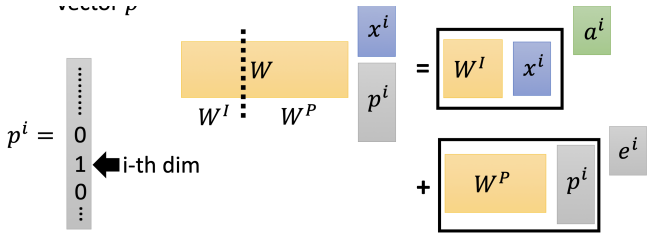



Figure 49: Positional embedding with matrix summation

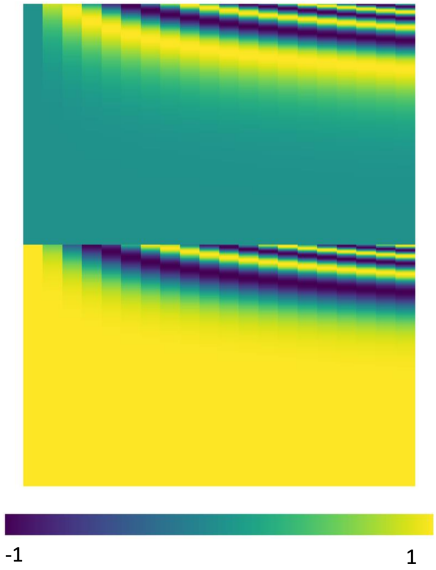



Figure 50: Visualization of $W^P$

**Non-Linearity** The above calculation is still a linear process, we need to **add non-linearity**, which is also **Feedforward layer**.

Here we have a MLP for Feedforward layer with ReLU. Note that in input dimension is 512, the hidden dimension is 2048 and the output dimension is also 512. We do this "point wise", which is apply MLP and ReLU on each *Row* of the attention output, so we do not mix information across different time-steps (rows) here.

$$\text{FFN}(x) \;=\; W_2\left(\max\big(0,\; W_1 x + b_1\big)\right) \;+\; b_2$$

So for the whole encoder block, the input is 512 dimensions and the output should also be 512 dimensions. There are 6 encoder blocks from original transformer.

**Noise and Weak Signal** With this much of parameters, there are two questions:

- Additional noise into the signal.
- Weak signal and vanishing gradients.

We can solve the noise issue with **Layer Normalization**: Layer norm changes

input to have mean 0 and variance 1, per layer and per training point, and adds two more parameters.

Note here the $\gamma$ and $\beta$ are learnable weights, so we know how much normalization should be applied dependent on the input.

A detail here is the layer norm is applied to the input embedding concatenated with the attention output.

$$\begin{aligned}
\mu &= \frac{1}{d_h}\sum_{i=1}^{d_h} x_i, \\
\sigma &= \sqrt{\frac{1}{d_h}\sum_{i=1}^{d_h}(x_i-\mu)^2}, \\
\hat{x} &= \frac{x-\mu}{\sigma}, \\
z &= \gamma\,\hat{x} + \beta.
\end{aligned}$$

To solve the weak signal and vanishing gradients, we sue **Skip/Residual connection**. The process is to do **element wise summation between the outputs of different attention blocks**.

> In **forward propagation**, this reduces chance of losing signal from earlier layers. In **back propagation**, this allows gradients to keep flowing on skip/residual connection.

Note here the output from the encoder will be a **Contextualized Embedding**, which is the **idea** that word embedding will be different dependent on context, even the word itself is the same.

## 11.3 Decoder

During training, we have some label as input.

**Masked Multi-Head Self-Attention** The key difference here is that given an input, the decoder has to predict the next token, but conventional Multi-Head Self-Attention gives the decoder access to correct predictions; as we do dot product with later inputs in the sequence. The goal is to use only inputs up to and including the current position.

This can be done by setting the dot product between current query and future key to 0. We achieve this by a Mask. In Figure 51 we show how this is done.

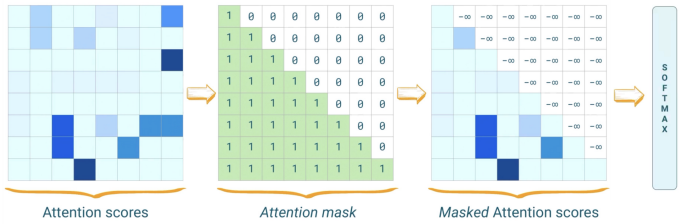



Figure 51: Masked Multi-Head Self-Attention

**Cross-Attention** This is by taking the output from the encoder, and get key, query and query and put the **encoder output's key and query** with vectors in decoder.

There are 6 Decoder blocks in the original transformer is well.

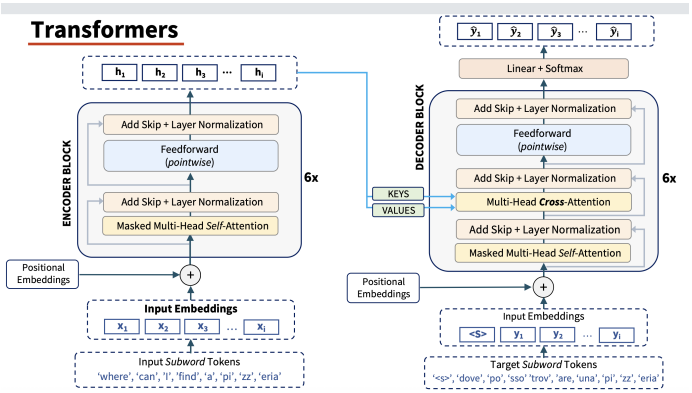



Figure 52: Final Original Transformers

## 11.4 Details in paper

- Byte-pair encodings
- Checkpoint averaging
- **ADAM** optimizer with learning rate changes
- **Dropout** during training at every layer just before adding residual
- Auto-regressive **decoding with beam search** and length penalties

Overall, they are hard to optimise and unlike LSTMs, don't usually work out of the box and don't play well yet with other building blocks on tasks.

## 11.5 Pre-Trained Transformer Categories

- **Encoder-only** (suited for predictive tasks; can technically be used for generative tasks)
  - BERT
  - ALBERT
  - RoBERTa
  - DistilBERT
- **Encoder–Decoder** (suited for sequence transformation tasks)
  - T5
  - Switch
  - BART
  - Pegasus
- **Decoder-only** (suited for generative tasks; can accomplish a wide range of tasks)
  - GPT-1, GPT-2, GPT-3, ...
  - Gopher
  - Gato
  - PaLM

## 11.6 BERT

The name is form Bidirectional Encoder Representations from Transformers (BERT). It is an **Encoder** Transformer, trained from a large text corpus without annotation. The main difference is how the model is been **trained**. There are two training tasks, *Masked Language Model* and *Next Sentence Prediction*.

**Masked Language Model** During training, 15% of input tokens are randomly selected, and of these 15%, replace 80% with a **[MASK]** token.

This result in a [MASK] in side some sentence, so the model need to use left and right context to predict what the [MASK] token is.

**Next Sentence Prediction** So the **idea** here is to predict whole sentence. See in Figure 53, the **CLS** token mark the begging and encoder all the information for the task. We also have **SEP** tokens where they are separator to separate sentence. Then the task is weather the second sentence follow the first sentence, and output T or F.

We can use this for like Sentiment Classification to output class label or Named entity recognition to output if the current token is an entity or a person or others.

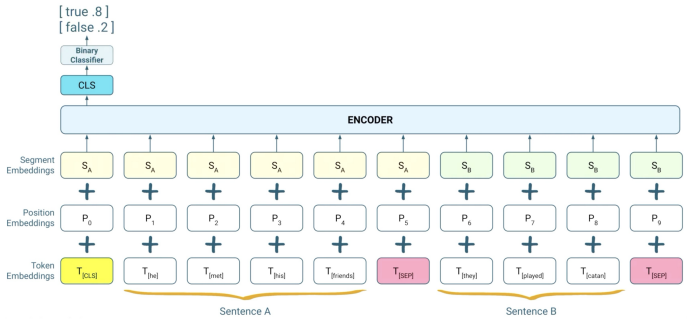



Figure 53: Next Sentence Prediction with BERT

# 12 Large Language Models

- In Context Learning
  - Zero shot
  - Few shot
  - Chain-of-Thoughts
- Instruction Finetuning
- Reinforcement Learning from Human Feedback

Two main new methods introduced from LLM:

- Pretrain models on large and general purpose corpora, learn syntax, semantics, sentiment, phrasal structures ect.
- Fine tune on smaller corpora, specialise the model to our particular task, like question answering in the biomedical domain. By fine tuning it is faster and uses less data.

Three methods related with LLM. *Zero Shot and Few Shot In Context Learning*, *Instruction finetuning*, *Reinforcement Learning from Human Feedback*.

## 12.1 In Context Learning

The ability to perform on many tasks with no examples and no gradient updates.

One paper show that some single neuron unit learnt to capture the sentiment of the reviews when trained on a language modeling task on a corpus of reviews.

**Zero shot** Few shot learning via Prompt-based 'learning', by add a prompt to the inputs of the language model so it can perform various tasks, the task can be unseen. Solve traditional tasks without fine-tuning

**Few shot** In context learning, based on Zero shot, with few examples of solution to the task.

Prompt Engineering to make the model do what ever you want it to do.

**Chain-of-Thoughts** Deal with tasks involving multi-step reasoning. This method show the reasoning process to the model.

In summary, in context learning we don't need fine-tuning, we can use prompts. But this method is limited by the length of the context, and complex task may still need gradient descent.

## 12.2 Instruction Finetuning

By design, language models are not used to assist users, therefore it is not aligned with user intent.

The idea is to collect examples of (instruction, output) pairs across many tasks and finetune an language model.

There are datasets to do this, like: Super Natural Instructions dataset.

Two problems:

- Tasks like open-ended creative generation have no right answer.
- Language modelling penalizes all token-level mistakes equally, but some errors are worse than others.

In summary, instruction funetuning is simple and straightforward, generalize to unseen tasks. But we need to collect demonstrations for so many tasks, which is expensive. This method mismatch between LM objective and human preferences.

## 12.3 Reinforcement Learning from Human Feedback

The idea is to explicitly encode human preferences about the task.

For each model generated text $S$, give a human reward $R(s) \in \mathbb{R}$. We want to maximize the (expected) reward of samples from our LM ($p$):

$$\theta^* = \arg\max_\theta \mathbb{E}_{\hat{s}\sim p_\theta(s)}\big[R(\hat{s})\big]$$

This line's **idea** is: On average, maximize the reward from the summaries generated using the parameter $\theta$.

This is basically reinforcement learning and with language model, it is hard. There are new RL algorithms that work for large neural models, including language models.

If the reward function is **nondifferentiable**, we use **Policy gradient** methods in RL, which estimating and optimizing this objective.

$$\mathbb{E}_{\hat{s}\sim p_\theta(s)}\big[R(\hat{s})\big]$$

There are two main problems:

**Problem 1** Human-in-the-loop is expensive. The solution is to train a Reward Model to predict human preferences from an annotated dataset, then optimize for $\mathrm{RM}_\phi(s)$ instead.

**Problem 2** human judgments are noisy and miscalibrated. The solution is to instead of asking for direct ratings, ask for **pairwise comparisons**, which can be more reliable.

Finally, we have the formulation for RLHF:

- A **pretrained LM** $p^{PT}(s)$, possibly instruction-finetuned
- A **reward model** $RM_\phi(s)$ that produces scalar rewards for LM outputs, trained on a dataset of human comparisons
- A method for **optimizing LM parameters** towards an arbitrary reward function

With RL, the model will tend to follow the reward blindly and greedily, so we need to apply some penalty which prevents the model from diverging too far from the original pretrained model.

So we need to initialize a copy of the model $p_{\phi}^{RL}(s)$, with parameters $\phi$ we would like to optimize:

$$R(s) \;=\; \mathrm{RM}_{\phi}(s) \;-\; \beta \log\!\left(\frac{p_{\theta}^{\mathrm{RL}}(s)}{p^{\mathrm{PT}}(s)}\right)$$

In summary, RLHF Model human preferences directly which is good, but RL is difficult to make it work.

# 13 Information Extraction

- 

There are mainly two tasks in IE, Identify the concepts (Named Entity Recognition) and Identify the relations and other properties (Relation Extraction).

## 13.1 Named Entity Recognition

NER is good for reducing sparseness in text classification, identify target in sentiment analysis and Question answering.

With NER, We have:

- **Named Entity** (NE): anything that can be referred to with a proper name
- Usually 3 categories: person, location and organisation
- Can be extended to numeric expressions like price, date, time
- The NER **pipeline**:
  1. **Identify** spans of text that constitute proper names
  2. Categorise by entity type.
- There are two **problems**:
  1. Ambiguity of segmentation (is it an entity? how to find boundaries?).
  2. Ambiguity of entity type
- To solve the problems, we have **weak supervision**.

- External **knowledge base**, like some knowledge that tell us "Amazon" is a "company"
- Word embedding, maybe more advanced embedding is needed like embedding from Bert. here we can see that embedding of "Amazon" is closer to embedding of "company" than "river"
- Use pretrained LM, like "Amazon is a [MASK]"
- Linguistic patterns

Standard NER algorithm is a *word-by-word sequence labelling task*, which is also a classification task. With this sequence labelling task, we have **BIO format**, which have ***B*** *as beginning*, ***I*** *as in*, ***O*** *is out*. For example, we have "American Airline", "American" is *B-ORG*, which is beginning of organisation, "Airline" is *I-ORG*, which is in organisation,

We can have a **bidirectional LSTM or Bert** that output the label each token in a text as being part of a named entity or not. We can use *Gazetteers* to assign labels to the tokens, *Gazetteers* are lists of place names, first names, surnames, organisations,products ect.

For **Evaluation**, we can use Precision, Recall or F-measure for it is a classification task. Note for **calculation**, we use number of entities, not tokens. **Segmentation** can cause problem. Be careful that we are using *words for training but entities as units of response*; if we have *Leamington Spa* in the corpus, and our system identifies *Leamington* only, that's a mistake.

In research, Statistical sequence models are the main focus. In Commercial use, rules is also used. These rules have *high precision* ($\frac{TP}{TP+FP}$) but *low recall* ($\frac{TP}{P}$), the rules are used to tag unambiguous entities. The use of ML is to increase recall.

## 13.2 Relation Extraction

Easy initiation, that we want relation with the entities we found.

Naive way to use **patterns**, some entities have some patterns with others, but this is a high precision but low recall method and it is hard to generalise.

The general pipeline of RE:

1. A **fixed set** of relations and entities is chose
2. A corpus is **manually annotated** with the entities and relations
3. A **classifier** is trained on the corpus and tested on an unseen text to classify potential relations between entity pairs

- The classifier need to first find pairs of named entities
- This classifier is trained to make a binary decision as to whether a given pair of named entities are related
- Then classifier need to assign a label to the relations that were found

**Supervised Learning** is helpful. But require **annotated dataset** which is **expensive**. The solution to expensive dataset is **Semi-Supervised Learning** or **Distance Learning**.

**Semi-Supervised Learning** The **idea** is to use Bootstrapping. We annotated pair of words with relation, then **expand** our training data with instances (pair) where our classifier's predictions are very **confident**, these pairs are called **seed**. We then use all sentence with this pair as training examples.

**Distance Learning** The **idea** is that if we know the relation between two entities, then any sentence that includes these two entities is likely to express the same relation.

Say we have "<Albert Einstein, born, Ulm>", then the relation of "Albert Einstein" with "Ulm" is "born", so any sentence with these two are expressing the relation of "born".

There are tools to establish these kind of dependencies.

And the **evaluation** is by Precision, Recall, F-measure. Where we measure the number of extracted relation tuple.

### 13.3 Other tasks

**Coreference Resolution** Finding all expressions that refer to the same entity in a text.

**Probing pre-trained LMs** Find words that have high relation with some entities.

**Entity Linking** Task of taking ambiguous entity mentions, and "link" them with concrete entries in a knowledge base.

## 14 Text Summarisation

- Extractive
  - Single-Document
  - Multi-Document
- Abstractive
- Evaluation

Difficultly in doing text summarisation can be: **Selecting** the most relevant information from a source document. **Expressing** that key information in the final summary

Two main methods:

- Extractive summarisation:
  - Identify most important segments (e.g. sentences, paragraphs) in source text.
  - Those segments will compose the target text.
  - **Challenge** lies in making the resulting text **cohesive**. When using baseline like methods like **Extract top N sentences**, the result is gap between sentences. And we sometimes want optimise topic coverage, where we want one sentence per topic.
- Abstractive summarisation:
  - Identify the key information of the source text.
  - Using that information, write new sentences that will form the target text.
  - **Challenge** lies in generating sentences (NLG, natural language generation).
- The **base line** is to take the first and last sentence as the summary.

The general pipeline for summarisaton:

- **Content Selection**: Choose sentences to extract.
- **Information Ordering**: Decide how to order the sentences.
- **Sentence Realisation**: Clean up sentences, remove non-essential phrases from sentences, fuse several sentences into one, fix problems in coherence.

## 14.1 Extractive

### 14.1.1 Single-Document

**Content Selection** This is usually treated as a classification task, if something is "extractworthy" or "non-extractworthy". This can be **unsupervised** or **supervised**.

For **unsupervised** methods, we can use TF-IDF (Term Frequency - Inverse Document Frequency) to determine the saliency of a word $w_i$ in a sentence, we do this for each word in the source text.

$$\text{weight}(w_i) \;=\; \text{TF}_{i,j} \;\times\; \text{IDF}_i$$

Each of the $k$ sentences in the input is then assigned a centrality score,that is the average cosine with all other sentences:

$$\text{centrality}(x) \;=\; \frac{1}{K}\sum_{y} \text{TF-IDF-Cosine}(x, y)$$

We can then choose to include the most **central** sentences in summary.

For **supervised** methods, we need documents and human-created summaries (label sentences **in** the summary as 1, all **other** sentences as 0). Train models like Maximum Entropy or Naive Bayes classifier and output the probability that each sentence is **extractworthy**. We can use features like:

- Position (e.g. position of sentence, is it first sentence?).
- Length of sentence.
- Informativeness of its words (e.g. tf-idf).

**Ordering of Sentences** We can use **Rhetorical Structure Theory** (RST), which is used for determine discursive roles of sentences. This can decide the order and ensure cohesive.

An other way is just to keep the **original order**. We assume that the content was well structured and ordered in the original document.

**Sentence Realisation** Things to do here can be Simplification of long sentences, and apply rule-based removal. Like Initial adverbials: for example, on the other hand, as a matter of fact, at this point. Or we can use ML instead of rules.

### 14.1.2 Multi-Document

**Content Selection** Problem: **Scarcity** of labelled corpora to train supervised classifier. So we stuck with unsupervised learning. Now the problem is to getting rid of **redundancy** across documents.

The solution is to use **Maximal Marginal Relevance** (MMR) for redundancy removal. It is a **Penalisation** factor based on similarity between sentence $s$ and the set of sentences already selected for the summary. We have $\lambda$ is a predefined parameter, and $Sim$ is some similarity scoring function:

$$\text{MMR\_penalisation}(s) \;=\; \lambda \max_{s_i \in \text{Summary}} \text{Sim}(s,\, s_i)$$

**Information Ordering** We can no longer keep the **original order**. We can use **chronological ordering**, the timestamps of the documents, but it result in **low cohesion**. Or we can use **lexical cohesion** to pairs sentences with low TF-IDF cosine distance next to each other.

**Sentence Realisation** Same methods form single document work.

We also need **coherence in named entity mentions**. Like "UK Prime Minister Rishi Sunak" is mentioned first then the rest are just "Sunak".

## 14.2 Abstractive

Use of LLM introduce factual issue, but traditional models tend to repeat itself.

We can use a **Pointer Network** from 10.4. Here we see that *cross attention* is for Extractive summarisation, for we have information about the original sentence, we can check how relevant are the original sentence and copy to generated text. And the self-attention is for Abstractive summarisation where we generate new words.

## 14.3 Evaluation

There is limited amount of datasets. Two main ways, **extrinsic** (task-based) or **intrinsic** (task-independent).

Extrinsic mainly focus on human, so like how many questions can the person answer after reading the summary.

For intrinsic, one is **ROUGE** (Recall Oriented Understudy for Gisting Evaluation), which is based on BLUE for Machine Translation. This metric computes the (word) overlap of (all of) unigrams, bigrams, trigrams and quadrigrams between reference and automatic translations. We take ROUGE2 (bigrams) as an

example:

$$\text{ROUGE2} = \frac{\sum_{S \in \mathcal{S}} \sum_{\text{bigram} \in S} \text{Count}_{\text{match}}(\text{bigram})}{\sum_{S \in \mathcal{S}} \sum_{\text{bigram} \in S} \text{Count}(\text{bigram})}$$

The problem is obvious, we don't have measure of cohesion, coherence, grammaticality, etc.

Or, we have the The Pyramid Method. Based on annotations of the reference summaries called **Summary Content Units** (SCU), i.e. sequences of words that we want in the summaries, basically the meaning of the sentence. We can measure how many SCUs are in each generated summary, with each SCU having a weight $w$, describing its relevance, bigger is more relevant. This $w$ is the number of times some sentence has appeared in reference summaries.

# 15 Recommender Systems

- 

## 15.1 Content-based Filtering

The main idea to to recommend some user $x$ with with similar stuff that is highly rated by $x$ in the past. This method focus on the **item and its content** that is been recommended, we fully **ignoring other users'** information.

The **pipeline** for Content Based recommendation:

- **Preprocessing** and feature extraction.
- Offline training from labelled data (e.g. ratings).
- Online generation of recommendations. (Need to be efficient here, near real-time recommendations)

**Preprocessing** Standard steps like **Stop Word removal**, **stemming** and **Phrase extraction** for stuff like extracting the name of some movie.

**Offline Training** Mainly two features to use, **Metadata** like director of some movie. And NLP over content like similar lyrics for music.

Combine the two features we can have something like:

| | Actor 1 | 2 | 3 | 4 | Comic | Pirate | Fantasy | Avg Rating |
|---|---|---|---|---|---|---|---|---|
| Movie X | 0 | 1 | 1 | 1 | 1 | 0 | 1 | $3\alpha$ |
| Movie Y | 1 | 1 | 0 | 0 | 1 | 1 | 0 | $4\alpha$ |

Table 9: One-hot feature vectors plus average-rating for two movies

Note the $\alpha$ is to scale the rating as all other feature is upper bounded by 1. Different $\alpha$ will result in different scores.

We then can get **cosine-similarity** between Movie X and Movie Y:

$$\cos(\mathrm{X}, \mathrm{Y}) = \frac{2 + 12\,\alpha^2}{\sqrt{5 + 9\,\alpha^2}\,\sqrt{5 + 16\,\alpha^2}}.$$

An other feature can be **Opinion Mining**, which is a subfield of sentiment analysis. Say we don't know if $x$ likes *romantic movies* or not, and $x$ has an review say "I hate romantic movies", then we know $x$ don't like it. An other example of feature is word **embeddings**.

**Pro**:

- No need for data on other users. **No cold-start problem** (no much ratings, for we only look at features of item).
- Able to recommend to users with **unique tastes**.
- Able to recommend **new and unpopular items**. No first-rater problem.
- We can provide better explanations of the **reasons behind a recommendation**, i.e. the features that the recommended item has in common with what they've viewed before.

**Con**:

- **Difficult to extract** features from content like for video.
- Difficult to implement **serendipity** (content-based systems tend to recommend very similar items).
- Even though content-based systems help in resolving cold-start problems for new items, they do not help in resolving these problems for new users (not enough information to check similarity between users)

## 15.2 Collaborative Filtering

The **idea** is find similar users and recommend items that they like.

We can build a **matrix** for $M$ *users* and $N$ *items* in our system. We can have 1 is viewed and 0 otherwise, or we can have a 5 star rating which is better. The **problem** now is how to give ratings to entries which the user has no rating of.

There are two ways to do this, one is **Model-based** Collaborative Filtering, which is use ML to do predictions, which will **not be covered**. The other is **Memory-based Collaborative Filtering**, where we use **patterns in past behaviour**. Two ways to do this. **User based Collaborative Filtering**, recommend based on items similar users liked. **Item based Collaborative Filtering**, recommend based on similar items the users liked.

### 15.2.1 User-based Collaborative Filtering

The pipeline:

1. **Retrieve** set of items $D$ the user $U$ liked.
2. **Find users** with similar items liked.
3. **Retrieve sets** of items $L_1..L_u$ those similar users liked.
4. Make **predictions** on items in $L_1..L_u$ that are not in $D$ based on how much user $U$ will like them.
5. **Recommend** top $N$ items to user $U$ based on above predictions.

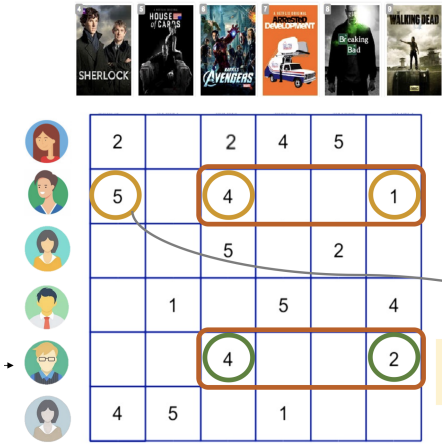



Figure 54: We see two users have similar rating to two movies, so they should have similar rating to the first movie.

### 15.2.2 Item-based Collaborative Filtering

The **idea** is that some movie (or other) should be **liked by the same set of users**, so we want to find this set of users. So we use **similarity between sets of ratings by two users or for two items**. We **don't care about the content** like categories or actor.

Here is how we calculate the similarity between two users $u$ and $u'$, so we can see if they are from the same set or not.

$$\text{sim}(u, u') = \cos\theta = \frac{\mathbf{r}_u \cdot \mathbf{r}_{u'}}{\|\mathbf{r}_u\| \, \|\mathbf{r}_{u'}\|} = \sum_i \frac{r_{ui} \, r_{u'i}}{\sqrt{\sum_i r_{ui}^2} \, \sqrt{\sum_i r_{u'i}^2}}$$

Here we can predict new ratings for $u$. This is basically the **Weighted sum of ratings** (normalised), and **Weights** are derived via **cosine similarity** (top).

$$\hat{r}_{ui} = \frac{\sum_{u'} \text{sim}(u, u') \, r_{u'i}}{\sum_{u'} |\text{sim}(u, u')|}$$

Say for the two users in Figure 54, we can calculate similarity between the

two users as:

$$\text{sim}(u, u') = \frac{5 \cdot 0 + 4 \cdot 4 + 1 \cdot 2}{\sqrt{5^2 + 4^2 + 1^2}\sqrt{4^2 + 2^2}} = \frac{18}{\sqrt{42}\sqrt{20}} = \frac{18}{28.98} \approx 0.62$$

We do this for all users, so we get the similarity of the target users and all other users. We then can predict the missing weighting, say for "Breaking Bad":

$$r_{ui} = \frac{0.26 \times 5 + 0.83 \times 2}{0.26 + 0.62 + 0.83 + 0.28} = 1.49$$

Not 0.26, 0.62 are similarity between the users and it is in order. We see here the rating 2 has higher weight 0.83 than rating 5, which make sense as this 0.83 similar user has similar rating for other movies with the target user.

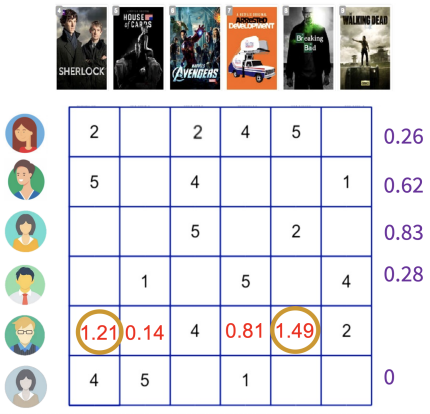



Figure 55: Calculation example for Item-based Collaborative Filtering. Here we can recommend movie 1 and 5.

In **practice**, this method work better than user-based. This can be caused by simplicity of items, where users have multiple tastes and can change over time.

**Good and Bad** about Collaborative Filtering:

**Pro**:

- **Easy to implement**: users and items can be treated as IDs, no content is processed.
- Performs reasonably well when we have a history of likes.

**Con**:

- **Cold start** and **popularity bias**.
  - **Cold start**: We need other users already in the system to find a match (no user with similar rating). Likewise, new items need to get enough ratings (if no rating for a new movie, no idea who will like it).
  - **Popularity bias**: We can end up having a **tendency** to recommend items that **everybody likes**. Items from the tail only **rated by a few** will **never be recommended**. Indeed, we can end up **recommending the same stuff over and over**.
- Can't perform well when we **lack sufficient history**. **Context** is ignored (e.g. if I move to a different city, don't recommend me restaurants in the previous city).

## 15.3 Hybrid Recommender Systems

Combine content-based methods and collaborative filtering methods. Use **Content based** methods will **help overcome the cold-start problem**. Once enough ratings, we use **Collaborative filtering**, avoid providing the user with recommendations that are always very similar in content, with no novel stuff.

Can also use other information:

- **Social recommendations**: what do my **friends/followers** like?
- **Demographics**: what do people of my **age** or from my **city** like?
- **Serendipity**: include **something new**, from the long tail of **unpopular items**.

## 15.4 Evaluation

There are two sections, one is about how to conduct evaluation, one is about how to assign scores.

### 15.4.1 How to conduct

When we have a platform with a **user base that is large**, we can do an **AB test**. Try different recommendation algorithm on different group of users. Say group A has one and group B has an other algorithm, see which group clicks more on the recommended items, A or B?

When we **don't have such a large user base platform**, we can use a **dataset for evaluation**. This will be **limited** to the data for items the user have already rated.

See in Figure 56, we can use part of the dataset as test set. Or we can hold out entire users or entire items, especially if we need to **test with cold-start** and/or introducing new products.

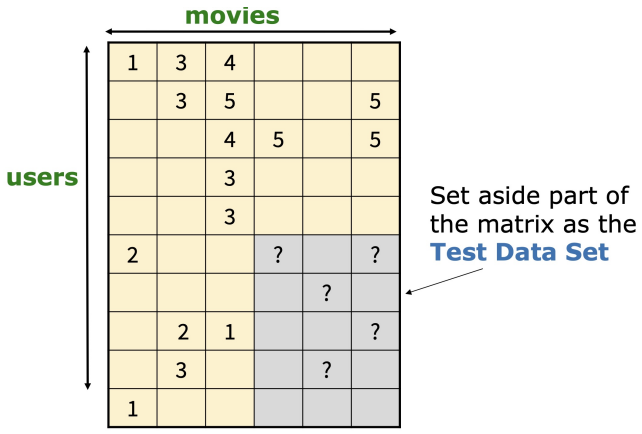



Figure 56: Dataset Example

### 15.4.2 Assign Scores

Dependent on type of recommender (rated, ranked, or not) we may evaluate as:

- **Rated**: A rating comparison problem (RMSE, MAE)
- **Ranked**: An information retrieval problem (MAP, NDCG)
- **Unranked, unrated**: A classification problem (Precision, Recall, F1,

With **Ratings** data, we have some numerical data so quite standatrd.

**Root-mean-square error** (RMSE), comparing predicted ratings vs actual ratings i.e. how close are we from the actual ratings?:

$$\text{RMSE} = \sqrt{\frac{\sum_{t=1}^{n}(\hat{y}_t - y_t)^2}{n}}$$

**Mean Absolute Error** (MAE)

$$\text{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n} = \frac{\sum_{i=1}^{n} |e_i|}{n}$$

With **Ranking** data, we have:

- **Precision@K** (Precision evaluated only up to the k-th prediction)
- **MAP** (Mean Average Precision)
- **NDCG** (Normalized Discounted Cumulative Gain)

**Precision@K** This is the ratio of relevant items within the top $k$ results.

$$\text{Precision@}k \;=\; \frac{\text{true positives@}k}{\text{true positives@}k \;+\; \text{false positives@}k}$$

Take example form Figure 57, system 1 has precision@5 of 0.4, and same as system 2. This method should be used when **ranking is not important**. So regardless of relevant items being the top 2 or the bottom 2 items. In some cases where the top ranked item has more importance.

| k | Query 1 | Query 2 |
|---|---|---|
| 1 | Relevant | Not-relevant |
| 2 | Relevant | Not-relevant |
| 3 | Not-relevant | Not-relevant |
| 4 | Not-relevant | Relevant |
| 5 | Not-relevant | Relevant |

Figure 57: Precision@5 (K=5). This graph show the top 5 ranked items and if they are actually relevant. If it is marked as *Not-relevant*, then it is a miss classify.

When **ranking is important**, use MAP or NDCG.

**MAP**

We first calculate the **AP** (Average Precision): up to position $k$, compute the average Precision@K for all positions $1..k$, then get the **mean** of all these $k$ scores, which is **MAP** (Mean Average Precision): average of **AP** over all examples (or queries) in the test set.

It weighs performance on top positions higher (precision on top 1 item is being counted in every iteration $1..k$).

From the same Figure 57, we show an example of calculation.

We first get all the **Precision** to $k$ for each query:

**Query 1:**

$$\begin{aligned}
\text{Precision@1} &= \frac{1}{1} = 1.00, \\
\text{Precision@2} &= \frac{2}{2} = 1.00, \\
\text{Precision@3} &= \frac{2}{3} \approx 0.67, \\
\text{Precision@4} &= \frac{2}{4} = 0.50, \\
\text{Precision@5} &= \frac{2}{5} = 0.40.
\end{aligned}$$

**Query 2:**

$$\begin{aligned}
\text{Precision@1} &= \frac{0}{1} = 0.00, \\
\text{Precision@2} &= \frac{0}{2} = 0.00, \\
\text{Precision@3} &= \frac{0}{3} = 0.00, \\
\text{Precision@4} &= \frac{1}{4} = 0.25, \\
\text{Precision@5} &= \frac{2}{5} = 0.40.
\end{aligned}$$

We then get the **Average Precision** up to $k$:

$$\begin{aligned}
\text{AP}_{Q_1}@5 &= \frac{1 + 1 + 0.67 + 0.50 + 0.40}{5} = \frac{3.57}{5} = 0.714, \\
\text{AP}_{Q_2}@5 &= \frac{0 + 0 + 0 + 0.25 + 0.40}{5} = \frac{0.65}{5} = 0.130,
\end{aligned}$$

We then take the **Mean**:

$$\text{MAP@5} = \frac{\text{AP}_{Q_1}@5 + \text{AP}_{Q_2}@5}{2} = \frac{0.714 + 0.130}{2} = 0.422.$$

**NDCG** This is **graded degrees of relevance**, normalised wrt ideal/perfect output.

We first compute **DCG@k**, where $rel_i$ is 1 if it is relevant and 0 other wise.

$$\text{DCG@}k \;=\; \sum_{i=1}^{k} \frac{\text{rel}_i}{\log_2(i+1)}$$

We then normalise it to get **NDCG@k**. Here $IDCG@k = DCG@k$ for the ideal system (all $k$ elements are relevant). The **Ideal DCG** (IDCG) is the maximum possible DCG that could be achieved if the results were ranked in the perfect order of relevance.

$$\text{NDCG@}k \;=\; \frac{\text{DCG@}k}{\text{IDCG@}k}$$

By the same example from Figure 57:

We first get **DCG** of both query:

$$\text{Query 1:}\quad \text{DCG}_{5,Q_1} = \frac{1}{\log_2(2)} + \frac{1}{\log_2(3)} + 0 \times 3 = 1 + 0.631 = 1.631,$$
$$\text{Query 2:}\quad \text{DCG}_{5,Q_2} = 0 \times 3 + \frac{1}{\log_2(5)} + \frac{1}{\log_2(6)} = 0.431 + 0.387 = 0.818$$

The **IDCG** should be:

$$\text{IDCG}_5 = 1 + \frac{1}{\log_2(3)} + \frac{1}{\log_2(4)} + \cdots + \frac{1}{\log_2(6)} \;\approx\; 2.948$$

We then can calculate the **NDCG** for each query:

$$\text{Query 1:}\quad \text{NDCG}_{5,Q_1} = \frac{\text{DCG}_{5,Q_1}}{\text{IDCG}_5} = \frac{1.631}{2.948} \approx 0.553,$$
$$\text{Query 2:}\quad \text{NDCG}_{5,Q_2} = \frac{\text{DCG}_{5,Q_2}}{\text{IDCG}_5} = \frac{0.818}{2.948} \approx 0.277$$

We can also get the Global **NDCG**:

$$\text{NDCG}_5 = \frac{\text{NDCG}_{5,Q_1} + \text{NDCG}_{5,Q_2}}{2} = \frac{0.553 + 0.277}{2} = 0.415$$

**Unranked, unrated** With **Unranked, Unrated** data, we can just use a **binary classification model**.

## 15.5 Other aspects

There can be many others things you want to evaluate, like **Coverage**, Number of items/users that the user recommends, i.e. we count for instance how many of the items in our database are being recommended to somebody.

There are other stuff in the slide that are too random, skip for now.

# 16 Question Answering

- 

Three parts here, **Information Retrieval (IR) based QA**, **Knowledge based QA** and **Hybrid QA Models**.

Two Paradigms for QA, one is **Information Retrieval** based approaches, one is **Knowledge-based and Hybrid** approaches.

## 16.1 Information Retrieval (IR) based QA

Three steps, **Question preprocessing**, **Passage Retrial**, **Answer Processing**.

### 16.1.1 Question preprocessing

Three steps:

- **Answer Type Detection**: Find the named entity type (person, place) of the answer
- **Query Formulation**: Choose query keywords for the IR system
- **Relation Extraction**: Find relations between entities in the question

**Answer Type Detection** Detect Named Entities like: "Who founded Virgin Airlines? - PERSON". This can be further fine-grained into some **Taxonomy** of Entities, like some coarse classes each with finer classes like LOCATION: city, country, mountain...

Different ways to detect, like **rule based** which is low in recall put high precision, and difficult to construct. Or by **ML**, define taxonomy of question types, Annotate training data for each question type then Extract features to train the classifier. Or we can do a hybrid of both.

We can use different features like:

- Question words and phrases
- Part-of-speech tags
- Parse features (headwords)
- Named Entities
- Semantically related words

**Query Formulation** Remove low IDF (inter document frequency) or stop words. So the **idea** here is query given by human may not be best for models, we need to make the query so that the model can perform the best.

### 16.1.2 Passage Retrieval and Answer Extraction

**Passage Retrieval** Three steps process:

1. Step 1: IR engine retrieves documents using query terms
2. Step 2: Segment the documents into shorter units, Something like paragraphs
3. Step 3: Passage ranking. Use answer type to help rerank passages

Here are many features that can be used for ranking the passage, like *Number of Named Entities* of the right type in passage, *Number of query* words in passage, Number of *question N-grams also in passage*, Cosine similarity of their **embeddings** ect.

We need to look in documents and form a **chunk vector**, which contain sentence from the documents in some embedding, and these sentence may contain information we need. We then **reorder** the vectors from the chunk with the chunk that is the most promising on top.

**Answer Extraction** The main **idea** here is to use **Retrieval Augmented Generation** (RAG). So we use the information retried, the chunks, and we use them to generate answer from LLM.

One naive way is to put the chunks into a prompt as given information and ask the model to generate the answer.

> Form Figure 58 gives idea to how is this QA formed. Same as before, we use **bidirectional LSTM** to get the vector from of the input **Question** and **Passage**. Note here not only we have **embedding** for the direct text of the passage, we also have PoS tag of the word as input, and the 1 and 0 mean if this **N-gram** is overlapped with some part of the Question. Then we

have **self attention** in question, cross attention between embedding. The output is **weighted sum** of Question and output of passage from LSTM. The output is a probability of this token been the **Start** of the Answer or the **End**. We have this start and end for each of the token, and we can use this to control the length of the output, like 5 words.

Sometimes we can have the **End** is before **Start**, then we need to filter this classification out and go to the next one until there is consistency.

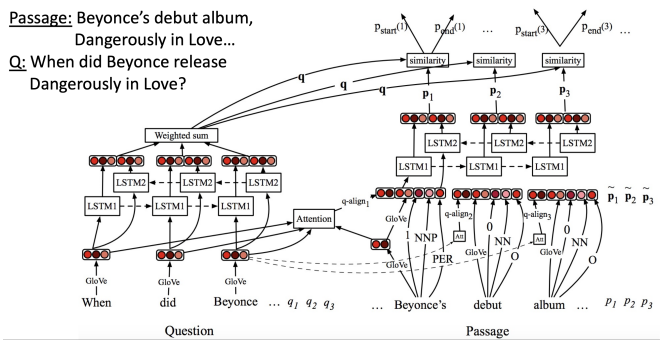



Figure 58: Neural Approach to QA, Seq2Seq as example. At high level, this is the same idea with LLM

Take reference from Figure 58.

We have the **question** represented as a single embedding $q$, computed by passing the series of embeddings of question words through a BiLSTM

**Attention**: the weight $b_j$ is a measure of the relevance of each question word, and relies on a learned weight vector $w$. Then apply SoftMax on attention.

$$q = \sum_j b_j h_j^q, \qquad b_j = \frac{\exp(w \cdot h_j^q)}{\sum_k \exp(w \cdot h_k^q)}.$$

To compute **passage embeddings**, for each word, concatenating four components to form the input:

- An **embedding** for each word such as from GLoVE
- **Token features** like the part of speech of the word, or the named entity tag of the word

- **Exact match features** representing whether the passage word $p_i$ occurred in the question: $\mathbb{I}(p_i \in q)$
- **Aligned question embedding**: use an attention mechanism to give a more sophisticated model of similarity between the passage and question words

$$a_{i,j} = \frac{\exp(\mathbf{p}_i \cdot \mathbf{q}_j)}{\sum_k \exp(\mathbf{p}_i \cdot \mathbf{q}_k)}.$$

The concatenated input components for each word in the passage is passed through a BiLSTM to derive the final passage word representation $p_i$ In order to find the answer span, we can train **two separate classifiers**:

- One to compute for each $p_i$, the probability $p_{start}(i)$ that $p_i$ is the **start of the answer span**.
- Another to compute the probability $p_{end}(i)$ that $p_i$ is the **end of the answer span**.

$$p_{\text{start}}(i) \propto \exp(\mathbf{p}_i W_s \mathbf{q}), \qquad p_{\text{end}}(i) \propto \exp(\mathbf{p}_i W_e \mathbf{q}).$$

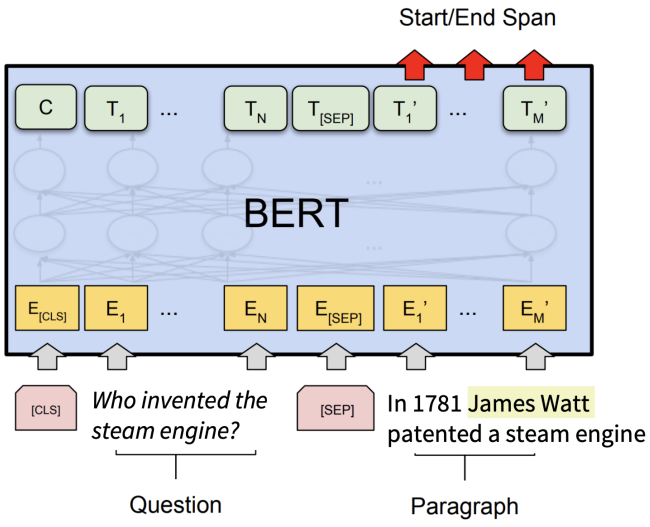



Figure 59: We can use Bert instead of Seq2Seq with the same idea.

## 16.2 Knowledge-based QA

So the **idea** is to use **Relation** form the database to extract the answer.

First we extract relation from question, like: "Whose granddaughter starred in E.T.?", we have relation of "(acted-in ?$x$ 'E.T.')" which try to find actor in E.T.. Then use this $x$ with relation "(granddaughter-of ?$x$ ?$y$)" to find the answer $y$.

We can do some **reasoning** like Temporal Reasoning, like some actor is born after the film came out is it is not possible.

Overall, this knowledge based QA system are **based on different types of entities and relations**, which are usually **encoded in the knowledge base**, and you have plenty of those depending on what you need a geospatial database, ontologies, restaurants, scientific, ect.

## 16.3 Hybrid QA Models

- Build a shallow semantic representation of the query
- Generate answer candidates using IR methods, Augmented with ontologies and semi-structured data
- Score each candidate using richer knowledge sources, like Geospatial or Temporal reasoning ect.

## 16.4 Datasets and Evaluation

**Dataset** We can manually collect dataset like from wiki or use existing ones.

**Evaluation** We can use accuracy, ratio of correct predictions.

Or **Mean Reciprocal Rank** (MRR):

- For each query return a **ranked list** of $M$ candidate answers.
- Query score is $\frac{1}{Rank}$ of the first correct answer.
  - If **first** answer is correct: 1
  - else if $nth$ **answer** is correct: $\frac{1}{n}$
  - score is 0 if **none of the** $M$ **answers are correct**
- Take the **mean** over all $N$ queries.

Or we can use **Exact match**: The percentage of predicted answers that match the gold answer exactly.

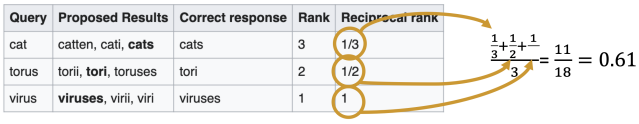



Figure 60: Example calculation of MRR

Or **F1 score**: Measures token-level overlap between predicted and gold answers, so **Precision** (fraction of predicted tokens in gold answer) and **Recall** (fraction of gold tokens in predicted answer).

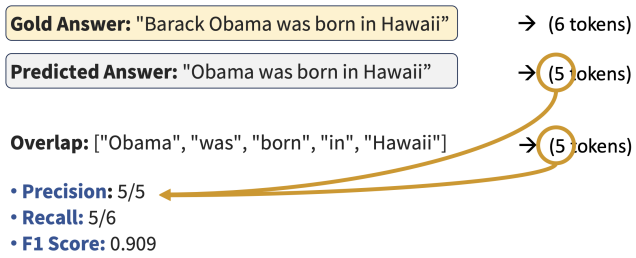



Figure 61: F1 calculation example