NLP4Web Zusammenfassung

Contents

NLP Basics

Anwendungsbeispiele:

- Search Engines
- Spelling Correction
- Question Answering
- Machine Translation
- Speech Recognition
- Plagiarism Detection
- Summarization
- Diachronic Analysis
- Text Generators

Das Web ist der Anwendungsbereich für NLP und auch gleichzeitig die Ressource um NLP zu verbessern.

NLP hat aber auch einige Herausforderungen:

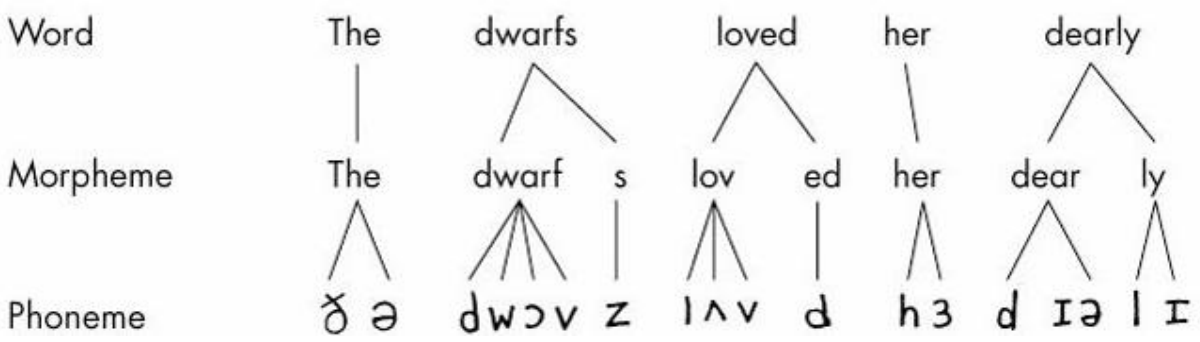
- Dubletten Bereinigung
- Inhaltsqualität
- Verteilter und heterogener Inhalt
- Umgang mit Fehlern (Rechtschreibung, Grammatik)
- Daten bereinigen

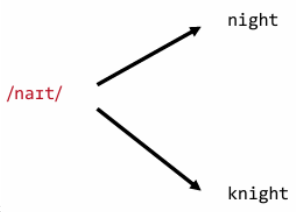
Eine Datenbereinigung ist notwendig, da nutzergenerierter Inhalt Fehler enthält.

Analyselevel von Sprachverständnis

Phonetik und Phonologie

Aufteilen von Wörtern in ihre Klangstämme.

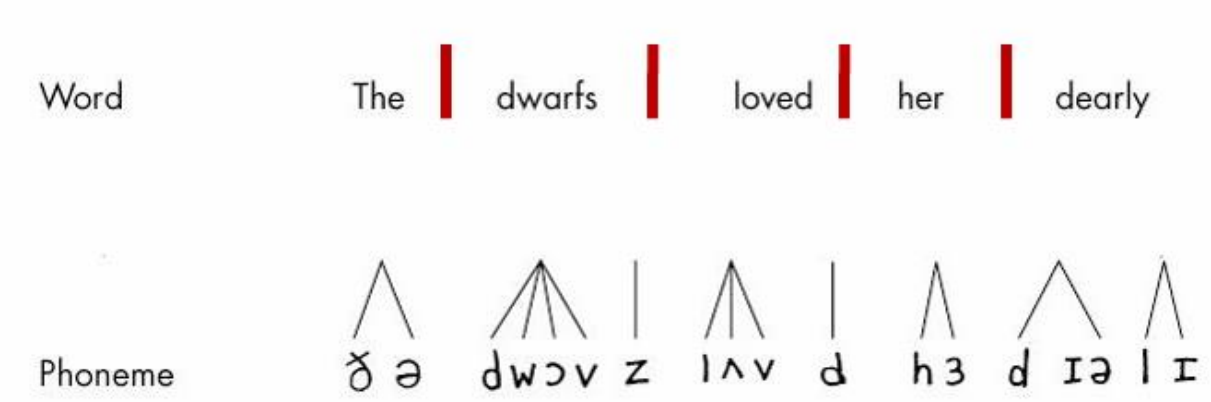




Homophones

Segmentation/ Tokenization

Teilen von Satzen/ Wörtern in einzelne Teile.



Das Segmentieren eines Input Streams in eine geordnete Sequenz von Einheiten nennt man Tokenization.

Ein Token kann eine bestimmte Wortform oder Sub-Wort Form sein und kann zur späteren morphologischen Analyse verwendet werden. Satzzeichen sind auch Tokens. Ein System, welches Text in Tokens teilt nennt man Tokenizer.

Punkt: In den meisten Fällen das finale Zeichen eines Satzes. Kann aber auch Teil einer Abkürzung, Nummer oder Ressource (Web-URL) sein. Außerdem wird bei einer Abkürzung am Satzende nur ein Punkt geschrieben.

Leerzeichen: Kann Teil einer Nummer sein oder auch Teil von Mehr-Wort ausdrücken (“New York”)

Komma: Kann Teil von Nummern sein.

Einzelnes Anführungszeichen: Kann innerhalb eines Tokens zusammengezogene Wörter bedeuten oder Teil eines Tokens (z.B. im Französischen) sein. In den meisten Fällen umschließen von zitierten Wortgruppen.

Bindestrich: Kann ein Trennzeichen oder eine Verbindung zwischen zwei Tokens bedeuten, aber meistens ist es Teil eines Tokens.

Chinesisch: Keine Leerzeichen, mehrere Segmentationen, welche Syntaktisch und semantisch korrekt sind aber die Unsicherheit kann mit kontextueller Information aufgelöst werden.

Deutsch: Keine Leerzeichen zwischen Nomen Zusammenbildung, mehrere Segmentationen welche Semantisch und Syntaktisch korrekt sind. Unsicherheit kann nur mit kontextueller Information aufgelöst werden.

Morphologie

Die Morphologie ist der Zweig von Linguistik, welches sich mit Wortformen und Wortbildung beschäftigt. Wörter werden dabei in Morpheme aufgeteilt. Morpheme sind die kleinsten bedeutungstragenden Einheiten.

Ein gebundenes Morphem ist Teil eines Wortes z.B. „-s“ bei „Cats“. Minimale freie Morpheme nennt man Stems („cat“). Stems beinhalten die zentrale Bedeutung des Wortes. Affixes sind gebundene Morpheme.

Affixe:

- Suffixe: Nach der Wortbasis (cat+s, nice+ly)
- Präfixe: Vor der Wortbasis (un+true)
- Infixe: Innerhalb der Wortbasis (fan+bloody+tastic)
- Circumfixe: Auf beiden Seiten der Wortbasis (ge+sag+t)

Die morphologische Normalisierung besteht in der Identifizierung eines einzigen kanonischen Vertreters für morphologisch verwandte Wortformen Formen. Hierfür verwendet man Stemming und Lemmatization.

Stemming

Stemming ist ein algorithmischer Ansatz, welcher die Endungen von Wörtern abschneidet.

Das Ziel ist es, Wörter in Gruppen zu sortieren mit derselben Morphologischen Familie, indem man diese in dieselbe gestemmte Repräsentation verwandelt. Stemming unterscheidet nicht zwischen Flexion und Derivation und die erhaltenen Stämme entsprechen nicht unbedingt einer realen Wortform.

Regelbasiertes Stemming ist schwer zu erstellen, haben häufig arbiträre Unterscheidungen aber können schnell während der Runtime ausgeführt werden.

Under-stemming: Zu wenig entfernt.

Over-stemming: Zu viel entfernt.

Syntaktische Ambiguität: z.B. Homographs, Wörter die dieselbe Schreibweise aber eine unterschiedliche Bedeutung haben. Solche Fälle können nicht nur mit Stemming gelöst werden, man benötigt zusätzlich die Grammatische Kategorie des Wortes.

Lemmatization

Die Flexionsänderungen einer Grundform „rückgängig“ machen. Braucht lexikalische Ressourcen und part-of-speech Tagging. Benötigt einen Umgang mit Unregelmäßigkeiten.

Syntax

Syntax beschreibt die Art und Weise, wie Wörter zusammen angeordnet sind.

"Syntax is the study of the regularities and constraints of word order and phrase structure" (Manning & Schütze, 2003, p. 93)

Es gibt eine unendliche Anzahl von Möglichkeiten, wie Wörter zu Sätzen zusammengesetzt werden können. Dennoch können wir Sätze verstehen, die wir nie zuvor gehört oder vorher gelesen haben.

Verschiedene Interpretationen von Sätzen entstehen meistens durch Syntaktische Ambiguität

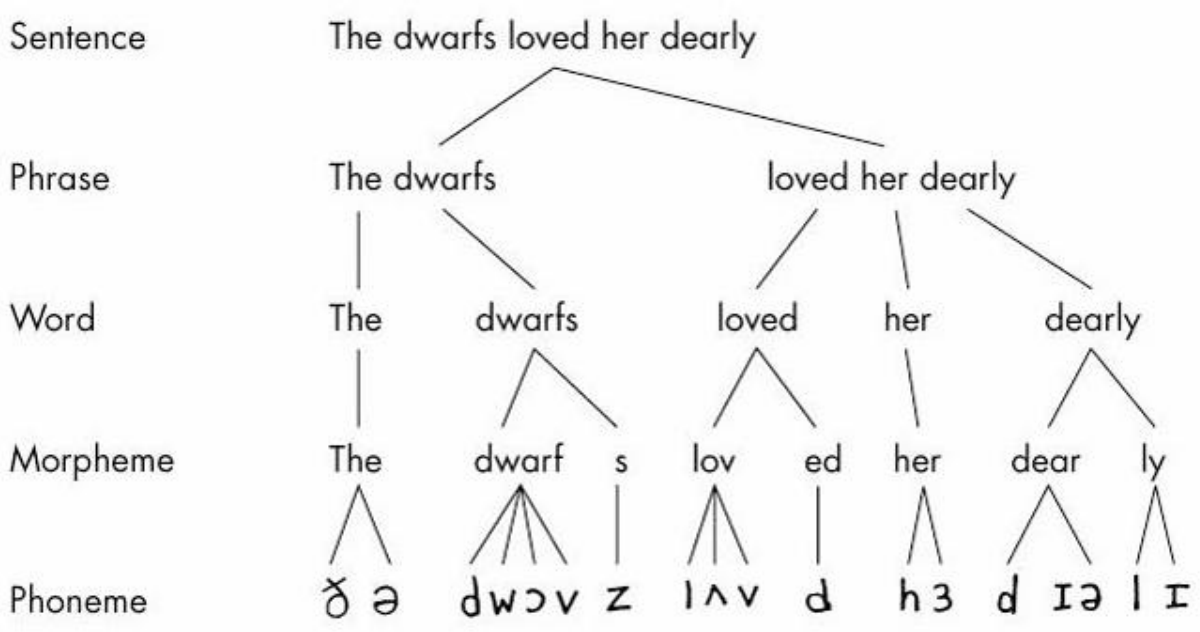
POS Tagging

Der Prozess der Zuordnung einer Wortart- oder lexikalischen Klassenmarkierung zu jedem Wort in einem Korpus. Die Eingabe für einen Tagging-Algorithmus ist eine Folge von Wörtern und ein Tagset, und die Ausgabe ist eine Folge von Tags, ein einzelnes bestes Tag für jedes Wort.

POS-Tagging ist eine Disambiguierungsaufgabe, Wörter sind mehrdeutig und haben mehrere mögliche POS Tags. Eine Zuweisung macht Wörter eindeutig.

Parsing

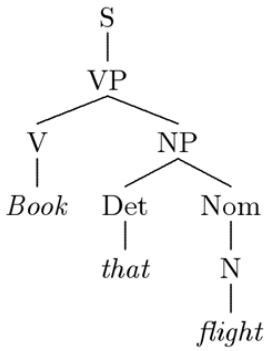
Feststellen der grammatikalischen Struktur mit Bezug zur gegebenen Grammatik.



Alternative Repräsentationen:

- Bracketed: $[_S\ [_{NP}\ [_{Det}$ the] $[_N$ dog]] $[_{VP}\ [_V$ ate] $[_{NP}\ [_{Det}$ a] $[_N$ cookie]]]]
- Paranthesized:

```
(S
    (NP
        (Det the)
        (N dog) )
    (VP
        (V ate)
        (NP
            (Det a)
            (N cookie))))
```

- Parse Tree: 

Aufgrund von Syntaktischen Ambiguitäten, gibt es mehrere Möglichkeiten einen Satz zu parsen.

Semantik

Semantik ist die Untersuchung von Bedeutung von Wörtern, Phrasen, Sätzen oder Dokumenten.

Lexikalische Semantik ist die Bedeutung von Lexikalischen Einheiten (Wörter).

Verschiedene Wortbedeutungen sind verursacht von Lexikalische Ambiguität.

Prgamatik

Die Pragmatik beantwortet die Frage nach dem Zweck einer Äußerung. Also die gemeinte Bedeutung:

Utterance: “Is it cold in here or is it just me?“

Intended meaning: “Please close the window!“

Text Classification

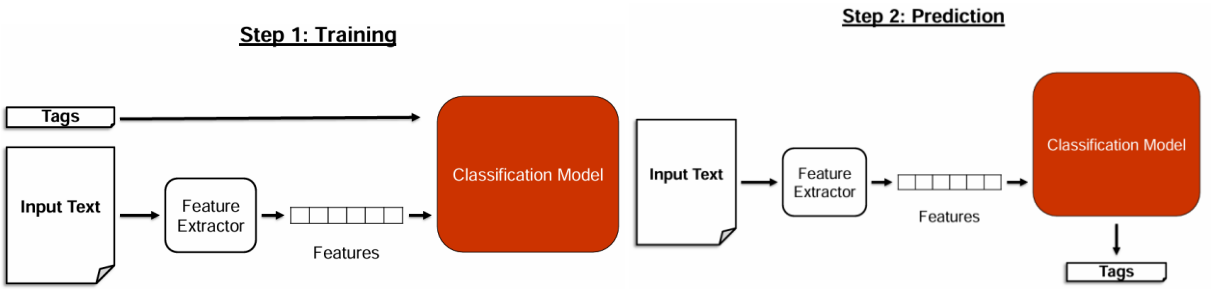
Textklassifizierung besäftigt sich damit, einen Input Text mithilfe eines Klassifikationsmodells in Tags oder Klassen zu klassifizieren.

Beispiele: Spam Detection, Sentiment Analyse (Film Review positiv/negativ), Topic Labeling, Age/Gender Identifizierung, Sprachidentifizierung, Autor Identifizierung.

Ansätze

Regelbasiert: Handgemachte linguistische Regeln, welche vom Menschen verstanden werden können. Regelbasierte Klassifizierung ist sehr Präzise aber sehr teuer zu erstellen und zu unterhalten.

Supervised machine Learning: Klassifikationsmodell basierend auf den Trainingsdaten. Nicht alle Modelle sind vom Menschen verstehbar. Supervised ML ist einfacher zu unterhalten und meistens auch präziser aber benötigt mehr Trainingsdaten.



Naïve Bayes

Abhängige Wahrscheinlichkeit

Abhängige Wahrscheinlichkeit ist die Wahrscheinlichkeit, dass etwas passiert mit der Voraussetzung, dass etwas anderes bereits passiert ist.

Beispiel: Wir nehmen an, wir haben ein Outcome O und Evidence E

$P(O, E)$ ist die Wahrscheinlichkeit, dass wir Outcome O und Evidence E haben, also die Multiplikation von

- $P(O)$ die Wahrscheinlichkeit, dass O auftritt
- $P(E|O)$ die Wahrscheinlichkeit, dass E auftritt, gegeben dass O aufgetreten ist

$$P(O, E) = P(O) \cdot P(E|O)$$

Bayes Rule

Konzeptionell ist dass das Mittel, um von $P(E|O)$ zu $P(O|E)$ zu kommen.

$$P(O|E) = \frac{P(E|O) \cdot P(O)}{P(E)}$$

Nun wollen wir aber ein Outcome O vorhersagen mit mehreren Evidences $E_1, \dots, E_2$. Der Naïve Bayes nimmt an, dass jedes Paar von Evidence unabhängig ist. Wir erhalten:

$$P(O|E_1, \dots, E_2) = \frac{P(E_1|O) \cdot P(E_2|O) \cdot \dots \cdot P(E_n|O) \cdot P(O)}{P(E_1, E_2, \dots, E_n)}$$

$$P(O, E) = \frac{P(Likelihood\ of\ E) \cdot Prior\ prob\ of\ O}{P(E)}$$

Die Intuition hinter der Multiplikation mit dem Prior ist, den häufigeren Ergebnissen eine hohe Wahrscheinlichkeit und den unwahrscheinlichen Ergebnissen eine niedrige Wahrscheinlichkeit zuzuordnen. Diese werden auch als **Base rate** bezeichnet und sind eine Möglichkeit, unsere vorhergesagten Wahrscheinlichkeiten zu skalieren.

Multinomial Naïve Bayes Model: Die Wahrscheinlichkeit, dass Dokument d zu einer Klasse c gehört ist proportional zu dem Produkt der Wahrscheinlichkeiten, dass Terme t zu einer Klasse c gehören und zur Klasse Prior $P(c)$:

$$P(c|d) \propto P(c) \prod_{1 \le k \le n_d} P(t_k|c)$$

Die beste Klasse c für ein Dokument d wird gefunden durch das Wählen der Klasse, für die die maximum a posteriori (map) Wahrscheinlichkeit maximal ist:

$$c_{map} = \underset{c \in \mathbb{C}}{\operatorname{argmax}} \hat{P}(c|d) = \underset{c \in \mathbb{C}}{\operatorname{argmax}} \hat{P}(c) \prod_{1 \leq k \leq n_d} \hat{P}(t_k|c)$$

- Bayes'sche Methoden bilden die Grundlage für probabilistische Lernmethoden
- Mit Bayes'schen Methoden kann die wahrscheinlichste Hypothese aus den Daten zu ermitteln
- Kein Training, nur Wahrscheinlichkeitsberechnung
- Binäre, numerische und nominale Merkmale können gemischt werden
- Naïve Bayes versagt, wenn die Unabhängigkeitsannahme zu stark verletzt wird
    - Besonders identische oder sich stark überschneidende Merkmale stellen ein Problem dar, das mit einer geeigneten Merkmalsauswahl angegangen werden muss

Hidden Markov Models

Das normale Klassifikationsproblem geht davon aus, dass individuelle Fälle nicht verbunden und unabhängig voneinander sind.

Viele NLP Probleme erfüllen allerdings nicht diese Annahme und haben viele zusammenhängende Entscheidungen, von denen jede eine andere Zweideutigkeit auflöst, die aber gegenseitig voneinander abhängig sind. Bessere Learning und Inference Techniken sind notwendig.

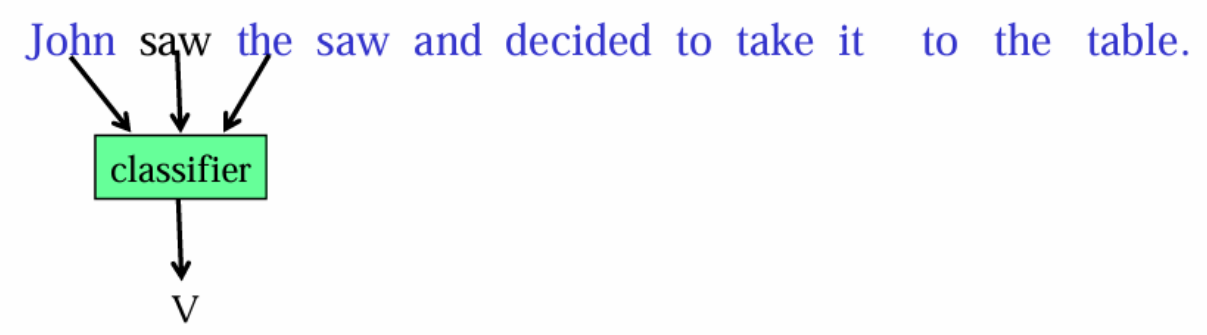
Sequence Labeling

Viele NLP-Probleme können als Sequence Labeling betrachtet werden. Jedem Token in einer Sequenz wird ein Label zugewiesen. Die Label eines Tokens sind abhängig von den Labeln anderer Token in dieser Sequenz, vor allem die Nachbarn. Einer der use cases ist POS Tagging.

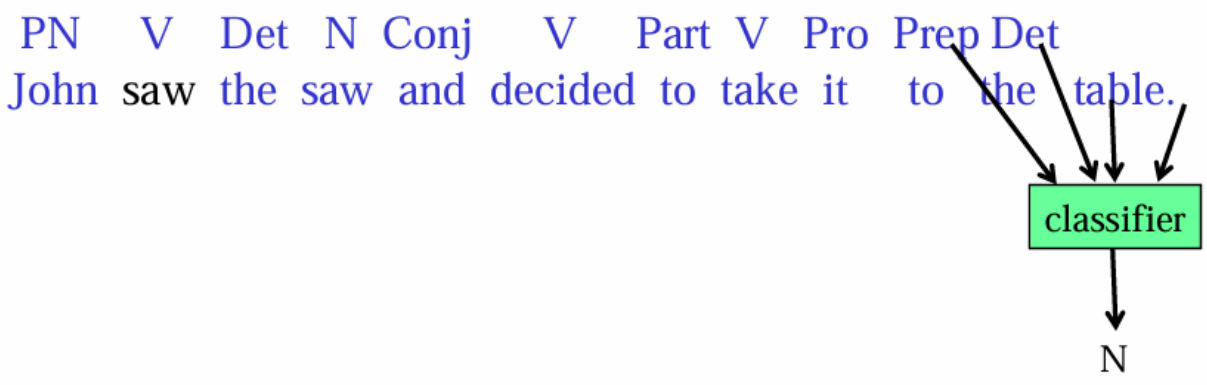
Gegeben einem Satz X wollen wir seine POS Tag Sequenz Y vorhersagen. Hierzu verwendet man z.B.

- Sequence Labeling als Klassifikation verwenden, dabei wird eine Pointwise Prediction gemacht, also jedes Wort einzeln mit einem Klassifikator vorhersagen.
- Generative Sequence Models (Hidden Markov Models)
- Neural Sequence models (RNN/ LSTM)

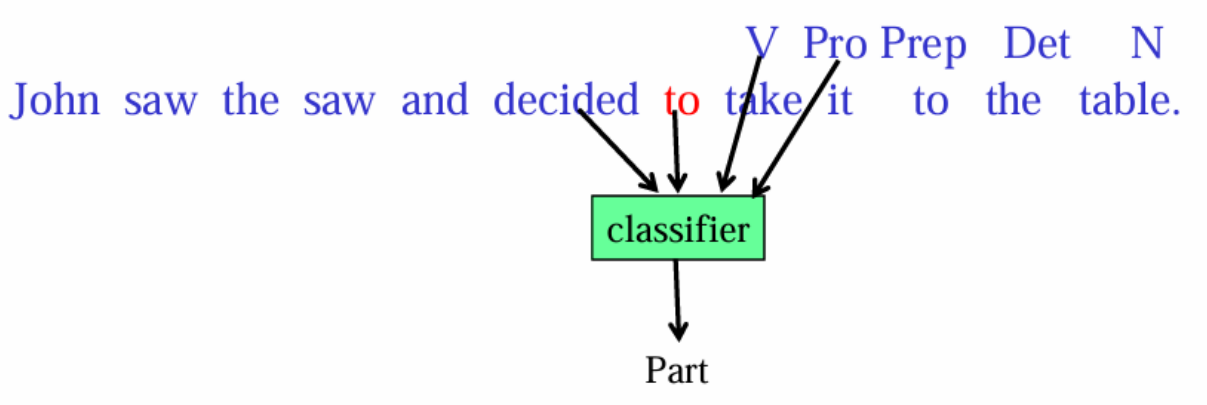
Wir betrachten zunächst Sequence Labeling als Klassifikation: Jeder Token wird hierzu einzeln klassifiziert aber verwendet hierzu die Informationen der benachbarten Tokens als Input Features (sliding window).



Bessere Input features sind meistens die Kategorien der benachbarten Tokens, diese sind aber noch nicht verfügbar. Man nutzt daher entweder den vorhergehenden oder nachfolgenden Token indem man vor- oder zurück geht und den vorherigen Output verwendet.



Die Disambiguität „to“ ist in diesem Fall Rückwärts sogar einfacher. Man kann leichter entscheiden, ob es sich um eine Präposition oder ein Particle handelt.



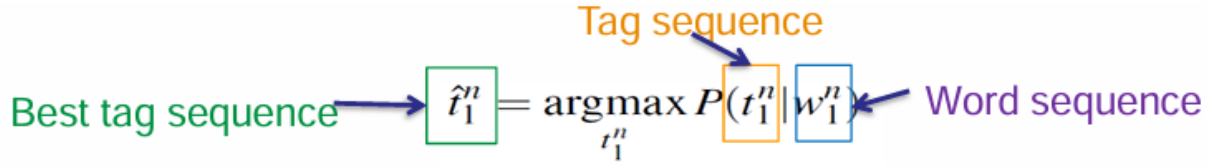
Probleme:

- Es ist nicht einfach, Informationen aus der Kategorie der Token auf beiden Seiten zu integrieren.
- Es ist schwierig, die Unsicherheit zwischen den Entscheidungen zu verbreiten und „kollektiv“ die wahrscheinlichste gemeinsame Zuordnung von Kategorien zu allen Token in einer Sequenz zu bestimmen.

Probabilistic Sequence Models

Probabilistische Sequenzmodelle erlauben es die Unabhängigkeit über mehrere, voneinander abhängige Klassifikation zu integrieren und kollektiv die global wahrscheinlichste Zuweisung festzustellen. Generative Sequenzmodelle sind z.B. Hidden Markov Model.

Beim Hidden Markov Model wählen wir die Tag Sequenz, welche am Wahrscheinlichsten ist gewählt, gegeben der Beobachtungssequenz von n Worten.



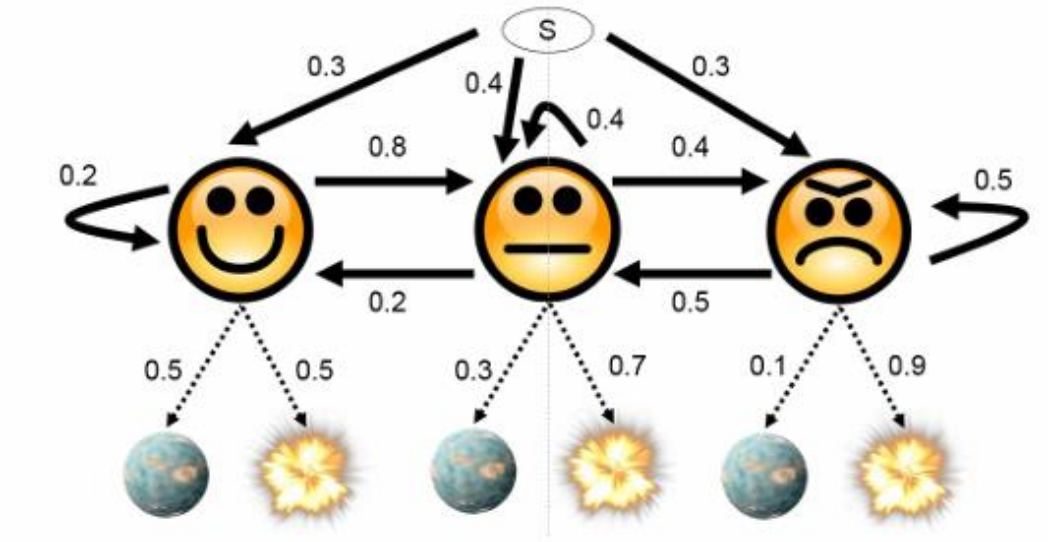
- Bayes Rule $\hat{t}_1 = \underset{t_1^n}{\operatorname{argmax}} \frac{P(w_1^n|t_1^n)P(t_1^n)}{P(w_1^n)}$
- Nenner verwerfen $\hat{t}_1 = \underset{t_1^n}{\operatorname{argmax}} P(w_1^n|t_1^n)P(t_1^n)$
- Annahme 1: Wortvorkommen ist nur vom eigenen Tag abhängig $P(w_1^n|t_1^n) \approx \prod_{i=1}^{n} P(w_i|t_i)$
- Annahme 2: Die Wahrscheinlichkeit eines Tags ist nur abhängig vom vorherigen Tag $P(t_1^n) \approx \prod_{i=1}^{n} P(t_i|t_{i-1})$

Hidden Markov Model

Ein Hidden Markov Model ist ein statistisches Model von versteckten, stochastischen Zustandsübergängen mit beobachtbarem, stochastischem Output. Key Features:

- **Ein festgelegtes Set von Zuständen**: Zu jedem Zeitpunkt ist das HMM in exakt einem Zustand.
- **Zustandsübergangwahrscheinlichkeiten**: Der Startzustand kann festgelegt oder probabilistisch sein.
- **Ein festgelegtes Set von möglichen Outputs**
- **Für jeden Zustand: Eine Wahrscheinlichkeitsverteilung für jeden möglichen Output**. Auch „Emission probabilities“ genannt.

Aufgabe: Für eine beobachtete Output Sequenz, was ist die (versteckte) Zustandssequenz, welche die höchste Wahrscheinlichkeit hat, diese Output Sequenz zu produzieren.



Beispiel: Was ist die Wahrscheinlichkeit für die folgende Mood Sequenz und die folgenden Observationen?

	Tag 1		Tag 2		Tag 3	
Transition Probability	S ▶	0.3	▶	0.5	▶	0.4
Emission Probability		0.1		0.7		0.9

Joint Probability	0.3*0.1=0.03	0.5*0.7=0.35	0.4*0.9=0.36

Wahrscheinlichkeit für die Sequenz 😠▶😐▶😠: 0.03*0.35*0.36=0.00378

Wenn wir nun HMMs auf POS Tagging anwenden, sind die (hidden) Zustände die POS Tags und die (beobachtbaren) Outputs die Tokens. Außerdem benötigen wir die Transition und Emission Probabilities.

Die Wahrscheinlichkeiten werden durch Zählen auf einem getaggten Training Korpus geschätzt:

- Transition Probability: wie oft auf das erste Tag das zweite Tag folgt, geteilt durch durch die Anzahl der Male, die das erste Tag in einem markierten Korpus vorkommt
$$P(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i)}{C(t_{i-1})}$$
- Emission Probabilities: die Anzahl der Assoziationen zwischen dem Wort und dem Tag im markierten Korpus geteilt durch die Anzahl der Male, die das erste Tag in einem markierten Korpus gesehen wurde
$$P(w_i|t_i) = \frac{C(t_i, w_i)}{C(t_i)}$$

Complexity: Was ist die wahrscheinlichste Zustandssequenz gegeben einer Outputsequenz?

$$\hat{t}_1 = \underset{t_1^n}{\operatorname{argmax}} P(t_1^n|w_1^n) \approx \underset{t_1^n}{\operatorname{argmax}} \prod_{i=1}^{n} P(w_i|t_i)P(t_i|t_{i-1})$$

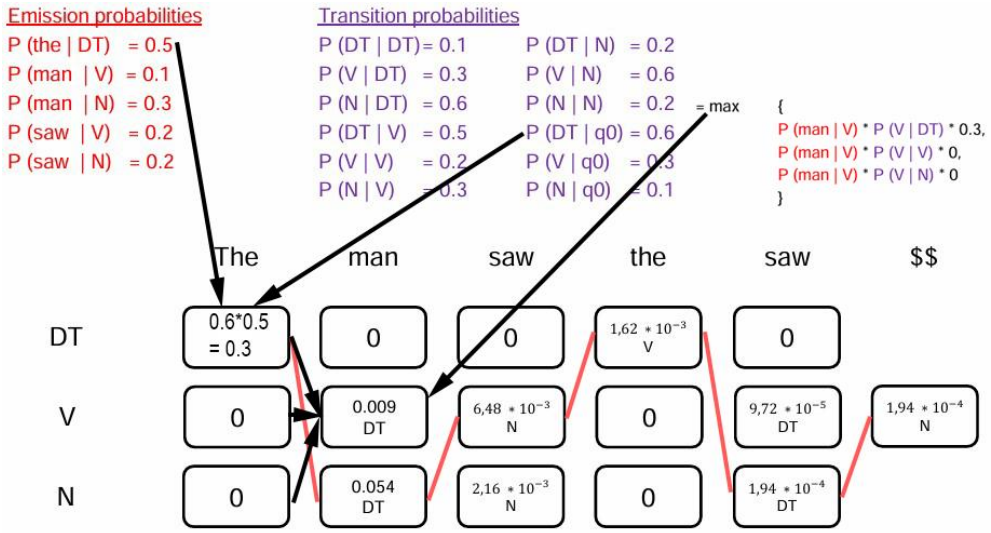
Naïve Lösung: Brute Force Suche durch alle möglichen Zustandssequenzen: $O(S^m)$ mit m, der Länge des Inputs und s die Anzahl der Zustände.

Bessere Lösung (Dynamic Programming): Viterbi Algorithmus ist der Standard: $O(ms^2)$

Viterbi Algorithmus

- Initialisierung: Setzt die Anfangswahrscheinlichkeiten für jedes mögliche Tag des ersten Wortes.
- Rekursion: Berechnet rekursiv die wahrscheinlichsten POS-Tags für jedes Wort, basierend auf Übergangswahrscheinlichkeiten (Tag-zu-Tag) und Emissionswahrscheinlichkeiten (Wort-zu-Tag).

- Backtracking: Rekonstruiert den optimalen Pfad der Tags durch Rückverfolgung der gespeicherten Entscheidungen.



Ein umfangreiches Beispiel findet man zu diesem Algorithmus im 2. Foliensatz auf Folie 90-107

Information Retrieval

Intuition: Wir wollen wissen, wie relevant ein Dokument für einen Suchterm ist.

Dokument: Kann alles sein, eine Website, ein Word-Dokument, eine Text-Datei, ein Artikel, usw. Man muss aber auf das Encoding, die Sprache, Hierarchie, Felder, usw. achten.

Collection: Eine Menge an Dokumenten.

Relevanz: Befriedigt ein Dokument den Informationsbedarf des Benutzers und trägt es zur Erfüllung der Aufgabe des Benutzers bei?

Einfachste Möglichkeit: Wenn ein bestimmtes Wort öfter auftaucht, ist das Dokument relevanter, also einfach die Wörter zählen. Wenn aber ein Dokument länger ist, wird ein Wort auch öfter darin auftauchen, daher die Länge des Dokuments mit einbeziehen. Aber Zählen, nur falls wir eine Query haben, ist sehr ineffizient.

Inverted Index

Der Inverted Index erlaubt es effizient Dokumente aus einer großen Collection zu entnehmen.

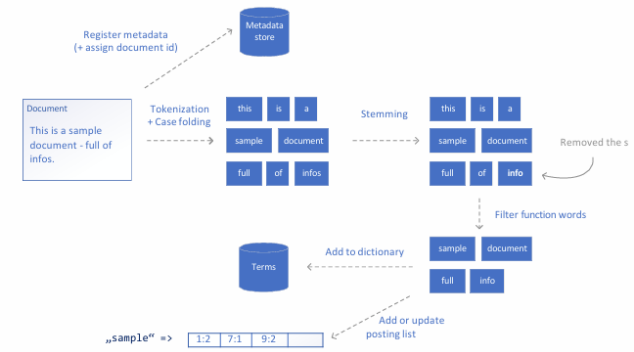
Der Inverted Index speichert alle Statistiken pro Term:

- Document frequency: Wie viele Dokumente beinhalten diesen Term
- Term frequency per document: Wie oft ist der Term in jedem Dokument
- Document length
- Average document length

Die Statistiken müssen in einem Format gespeichert werden, sodass es zugreifbar für einen gegeben Term ist und die Metadaten eines Dokuments (Name, location of full text, ...) werden auch gespeichert.

Jedes Dokument erhält eine interne Dokumenten ID. Das Term Dictionary wird in einer Suchfreundlichen Datenstruktur und die Term Frequencies werden in einer „posting list“ (Eine Liste von Dokument ID und Frequenz) gespeichert.

Rechts sieht man eine vereinfachte Beispiel Pipeline. Linguistische Modelle sind Sprachabhängig. Ein Querytext und ein Dokument müssen beide die selben Schritte durchlaufen.



Search and Relevance

Wenn man den Inverted Index abfragt, braucht man nicht das gesamte Dokument lesen. Man arbeitet nur mit den Frequenznummern von potenziell relevanten Dokumenten. Die Dokumente werden basierend auf dem Relevance Score sortiert, um das relevanteste Dokument zu erhalten.

Querytypes:

- **Exact matching:** Ganze Wörter abgleichen und verketten von mehreren Wörtern mit OR
- **Boolean queries:** AND, OR, NOT Operatoren zwischen Wörtern
- **Expanded queries:** Automatisch Synonyme oder andere ähnliche oder relevante Wörter der Query hinzufügen.
- **Wildcard queries, phrase queries, phonetic queries**

Dictionary: Ein Dictionary<T> mappt Text zu T. T ist eine Posting List oder potenziell andere Daten über den Term in Abhängigkeit zum Index. Wir wollen zufälliges, schnelles und speichereffizientes Nachschlagen.

Scoring model: Paarweise Evaluierung (1 query, 1 document): $score(q, d)$. Wir wollen also die Relevanz in einem mathematischen Modell festhalten.

```
float Scores={}
for each query term q
    fetch posting list for q
    for each pair(d, tf_{t,d}) in posting list
        if d not in Scores do Scores[d]=0
        Scores[d] += score(q, d, tf_{t,d}, ...)
return Top K entries of Scores
```

Suchalgorithmus

Beschränkungen:

- **Relevanz** bedeutet relevant zum Zweck und nicht zur Query
- **Query** ist eine Abkürzung für einen Informationsbedarf, seine erste verbalisierte Präsentation durch den Benutzer
- Es wird, angenommen dass Relevanz ein **binäres Attribut** ist. Ein Dokument ist entweder relevant oder nicht.
- Wir brauchen diese Vereinfachungen um mathematische Modelle zu erstellen und evaluieren.

Term frequency, inverse document frequency (TF-IDF, BM25)

- **Bag of words:** Wortreinfolge ist nicht wichtig.
- Der erste Schritt für das Retrieval Model: Anzahl der Vorkommen zählen.

Term frequency: $tf_{t,d}$ Anzahl der Vorkommen von Term t in Dokument d

- Der Inverted Index speichert nur nicht-0 Einträge, also nicht die gesamte Matrix, da das sehr ineffizient ist
- Daher schlecht für zufälliges Nachschlagen in einer Dokumentenspalte, da man erst durch die Posting Liste nach dem korrekten Dokument suchen muss, aber man kann Dokumente mit $tf_{t,d} = 0$ überspringen.
- TF ist ein starker Startpunkt für viele Abfragemodelle aber die rohe Frequenz ist nicht die beste Lösung, es macht mehr Sinn relative Frequenzen zu verwenden und die Werte mit dem Logarithmus zu dämpfen
- Experimente zum Abruf von Daten zeigen, dass die Verwendung des Logarithmus der Anzahl der Termvorkommen effektiver ist als die reine Anzahl: $\log(1 + tf_{t,d})$

Document frequency: df_t ist die Anzahl der Dokumente, in denen Term t vorkommt. Seltene Terme haben einen höheren Informationsgehalt als häufige Terme (z.B. und, oder, die, …). Wir wollen ein hohes Gewicht für seltene Terme. Die document frequency ist ein inverses Maß für die „Informativität“ eines Terms.

Inverse Document frequency: $idf(t) = \log(\frac{|D|}{df_t})$, wobei $|D|$ die gesamte Anzahl von Dokumenten und $df_t \leq |D|$.

TF-IDF: $TF\_IDF_{(q,d)} = \sum_{t \in T_d \cap T_q} \log(1 + tf_{t,d}) \cdot \log(\frac{|D|}{df_t})$ wobei über alle Queryterme summiert wird, die im Index sind.

- Ein seltenes Wort (in einer Collection), welches häufig in einem Dokument auftaucht bekommt einen hohen Score
- Häufige Wörter bekommen einen niedrigen Score
- TF-IDF nicht nur als alleinständiges Modell gut sondern die Gewichte werden für viele Abfragemodelle als Basis verwendet.
- Auch gut als generische Wortgewichtungsmethode für NLP

BM25: $BM25(q,d) = \sum_{t \in T_d \cap T_q} \frac{tf_{t,d}}{k_1\left((1-b)+b\frac{dl_d}{avgdl}\right)+tf_{t,d}} \cdot \log(\frac{|D|-df_t+0.5}{df_t+0.5})$, wobei dl_d die Dokumentlänge, $avgdl$ die durchschnittliche Dokumentenlänge im Index und k_1, d Hyperparameter sind.

- k_1 kontrolliert wie die Termfrequenz skaliert
- b kontrolliert die Normalisierung der Dokumentenlänge
- Gängige Werte: $0.5 < b < 0.8, 1.2 < k_1 < 2$
- Es wurde gezeigt, dass die komplexeren Teile (k1, b, log und +0.5) nicht notwendig sind.
- Die einfache BM25 Variante ist TF-IDF sehr ähnlich, mit dem großen Unterschied, dass die Termfrequenz von BM25 eine Sättigungsfunktion beinhaltet.
- BM25 Varianten können angepasst werden auf längere Queries, mehrere Felder und zusätzliche Referenzinformationen
- TF-IDF: Gewicht wird immer größer (trotz log)
- BM25: BM25 nähert sich $k_1 + 1$ an

Evaluation

Wir evaluieren Systeme, um konkrete Beweise für eine Hypothese zu finden. IR Systeme sind schwer zu evaluieren (Ambiguität, Collection size). Es gibt verschiedene Typen um die Qualität von Ergebnissen zu evaluieren:

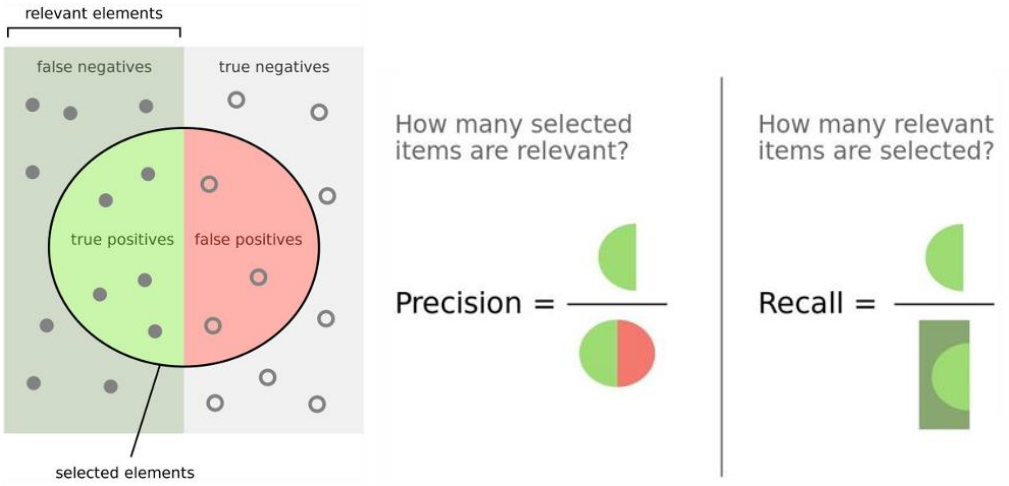
- Intrinsic: Festes Set, gleiche Collection, Query Set und Label
- Extrinsic: Nutzerverhalten beobachten (Im Produktivsystem)

Wir wollen bei der Evaluation von unseren IR Systemen folgende Frage beantworten: Beinhaltet ein Dokument die Antwort zu unserer Query?

Es gibt aber auch noch andere Möglichkeiten wie Effizient, Fairness, Diversität, Inhaltsqualität, Quellenauthentizität, Mühe, etc.

Bei einer Extrinsic Evaluation vergleichen wir die Qualität von Systemen, die eine Rangliste von Dokumenten produzieren. Diese werden durch einen Pool von Urteilen verglichen.

Precision & Recall



Man kann den Recall verbessern in dem man einfach mehr Dokumente zurück gibt, es ist aber auch einfach eine hohe Präzision mit einem geringen Recall zu bekommen. Der F-score kombiniert die beiden und gobt ein harmonisches Mittel über P und R.

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R}$$

Mean Reciprocal Rank & Mean Average Precision

MRR und MAP sind binäre Labels für Ranking List Evaluation Metrics.

Typischerweise wird an einem Cutoff @k von Top gerankten Dokumenten gemessen.

$$MRR(Q) = \frac{1}{|Q|} \cdot \sum_{q \in Q} \frac{1}{FirstRank(q)}$$

- MRR setzt den Fokus auf das erste relevante Dokument
- Nur anwendbar mit spärlicher Beurteilung oder der Annahme, dass der Nutzer damit zufrieden ist, nur das relevanteste Dokument zu erhalten

$$MAP(Q) = \frac{1}{|Q|} \cdot \sum_{q \in Q} \frac{\sigma_{i=1}^{k} P(q)_{@i} \cdot rel(q)}{|rel(q)|}$$

- MAP quetscht eine komplexe Evaluation in eine einzige Nummer und ist schwer zu Interpretieren
- MAP entspricht dem Flächeninhalt unter der Precision-Recall Kurve

- $P(q)_@$ ist die Präzision der Query q nach den ersten i Dokumenten, $rel(q)$ ist die Binäre Relevanz des Dokuments an Stelle i

Normalized Discounted Cumulative

Graded Relevanz erlaubt es uns verschiedene Relevanzwerte zu vergeben. Diese können eine Fließkommazahl sein oder eine Menge von Klassen für manuelle Annotation.

Common Graded TREC Relevance Labels:

- [3] Perfekt Relevant: Dokument ist der Anfrage gewidmet, es ist würdig, ein Top-Ergebnis in einer Suchmaschine zu sein.
- [2] Hoch Relevant: Der Inhalt dieses Dokuments liefert umfangreiche Informationen über die Abfrage.
- [1] Relevant: Das Dokument enthält einige für die Abfrage relevante Informationen die für die Abfrage relevant sind, die jedoch minimal sein können.
- [0] Irrelevant: Das Dokument enthält keine nützlichen Informationen über die Abfrage.

$$DCG(D) = \sum_{d \in D, i=1} \frac{rel(d)}{\log_2(i+1)}$$

$$nDCG(Q) = \frac{1}{|Q|} \cdot \sum_{q \in Q} \frac{DCG(q)}{DCG(sorted(rel(q)))}$$

- nDCG vergleicht tatsächliche Ergebnisse mit dem Maximum pro Query. Die Relevanz wird bewertet.
- nDCG@10 ist die am meisten verwendete Offline Web Search Evaluation

Word Representation

Gradient Descent basierte Neurale Netzwerke arbeiten auf kontinuierlichen Räumen. Das bedeutet, wir können nicht einfach die Wörter als Input in ein lineares Algebra Netzwerk geben und erwarten, dass es funktioniert, wir müssen die Wörter oder Wortteile auf Vektoren abbilden.

Die einfachste Methode ist One-Hot Encoding, hierbei gibt es eine Vektordimension für jedes Wort in unserem Vokabular. Dies ist aber sehr spärlich, da jeder Wortvektor nur eine 1 hat und sonst nur 0en.

Word Embeddings

Word Embeddings bieten eine dichtere Vektorrepräsentation für Wörter. Typischerweise 100-300 (abstrakte) Dimensionen. Dieser Vektorraum erlaubt uns mathematische Operationen.

Word Embeddings können unsupervised auf großen Text Datensätzen vortrainert werden und im Model fein abgestimmt werden.

Werden in einer einfachen Datenstruktur gespeichert: Dictionary<string, float[]>

Es gibt mehrere unsupervised (creation) Methoden:

- 1-Wort-1-Vector
  - Word2Vec
  - Glove
  - Viele spezialisierte Varianten von den Beiden

- 1-Wort-1-Vector+Char-n-grams
    - FastTex (basiert auf Word2Vec)
- Kontextualisiert, Kontextabhängig, Komplexe Struktur
    - ELMo
    - Transformer a la BERT und Varianten davon

Unsupervised Training: Training ohne explizite Label aber mit echtem Text (dieser wird modelliert). Die Aufgabe ist es das nächste Wort vorherzusagen, gegeben einer Sequenz von Wörtern. Varianten: Kontextwörter vorhersagen (-2, -1, +1, +2, ...), Maskierte Wörter vorhersagen

Word2Vec: Ein 1-Layer Hidden Netzwerk trainieren um Kontextwörter vorherzusagen. Zielwörter über 1-Hot Encoding.

- Die Wortvektoren erhalten wir aus dem Netzwerk (Matrix nehmen): Jede Zeile entspricht der 1-Hot Position eines Wortes.
- Die Outputmatrix wird meistens ignoriert
- Training mit einem Sliding Window über den Inputtext
- Negative Log Likelihood Loss berechnen (aber nicht über alle sondern random gewählte Terms)

Word Ordering / N-Grams: Die Reihenfolge und der lokale Kontext ist wichtig. Betrachtet man N Worte gleichzeitig nennt man es N-Gram. Bi-Grams oder Tri-Grams sind nicht lohnend.

Limitationen:

- Word2vec, Glove & FastText bilden immer 1 Wort auf 1nen Vektor ab (Gut für Analyse und Umgebungen mit wenigen Ressourcen)
- Keine Kontextualisierung im Vektor nach dem Training. Der Vektor eines Wortes wird nicht durch die anderen Wörter im selben Satz beeinflusst. Wörter mit mehreren Bedeutungen werden anhand des Kontextes zusammengefügt.
- Viele Wörter haben viele Bedeutungen anhand des Kontextes

Query Expansion:

- Suche mit ähnlichen Worten erweitern.
- Collection Statistiken aktualisieren
- Relevanzmodell anpassen, sodass ein Dokument mit mehreren ähnlichen Worten zusammen einen Score erhält

Byte-Pair Encoding

Wie gehen wir mit unbekannten Wörtern um? Wir betrachten nicht Wörter als Tokens sondern Sub-Wörter.

Vorteile:

- Kompressionseffizienz: Häufige Sequenzen werden als ein Token behandelt
- Adaptivität: Encoding kann für verschiedene Texttypen optimiert werden
- Flexibilität: Kann Out-of-Vocabulary Text behandeln

Initialisierung: Vokabular (Set von allen individuellen Zeichen)

Wiederholen bis k Zusammenführungen:

- Zwei Zeichen wählen, die am meisten als Paar auftreten (a,t)

- Neues Zusammengesetztes Zeichen hinzufügen (at)
- Jedes Vorkommen mit dem neuen Zeichen ersetzen (t,h,a,t => t,h,at)

Beispiel: siehe Foliensatz 4, Folie 18-20

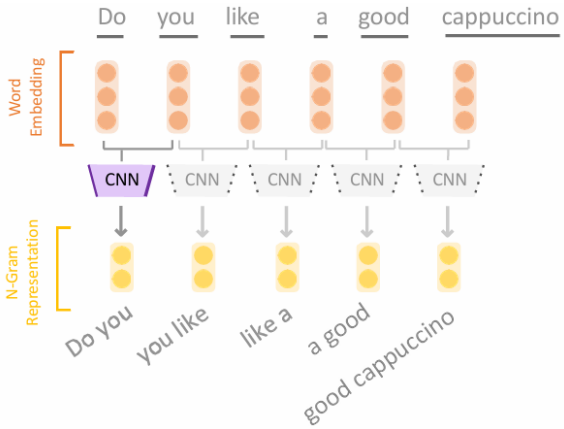
Simple Neural Techniques

Convolutional CNN

- 2D CNNs werden vorallem in Computer Vision verwendet.
- 1D CNNs haben einen eindimensionalen Filter und arbeiten auf eindimensionalen Input.

CNNs wenden einen Filter mit einem Sliding Window auf den Input daten an. Die Werte in der Filterregion werden zu einem Outputwert zusammengeführt. Die Filterparameter werden während dem Training gelernt. Typischerweise werden mehrere Filter parallel angewendet.

Für die N-Gram Modellierung wenden wir ein 1D CNN auf eine Sequenz von Wortvektoren an. Dabei wird das N als Filtergröße verwendet. Das Output ist eine Sequenz von N-Gram Repräsentationen. WE & CNN können end-to-end trainiert werden.



N-Gram Modellierung mit 1D CNN

```
conv = nn.Sequential(
         nn.ConstantPad1d((0,2 - 1), 0),
         nn.Conv1d(kernel_size=2,
                   in_channels=300,
                   out_channels=200),
         nn.ReLU())
```

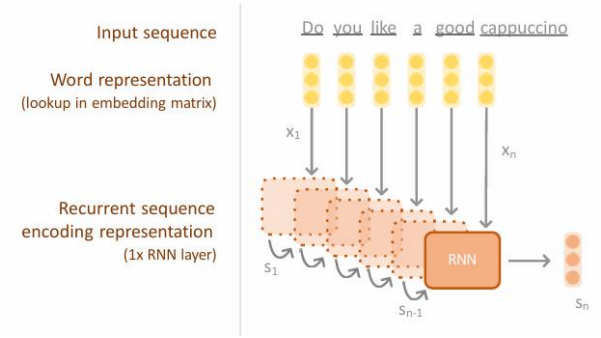
```
embeddings_t = embeddings.transpose(1, 2)
n_grams = conv(embeddings_t).transpose(1, 2)
```

Recurrent NN

Recurrent Neural Networks modellieren globale Muster in sequenziellen Daten (erlaubt arbiträre Inputgröße). RNN sind trainierbar mit Backpropagation und Gradient Descent. Außerdem haben RNNs die Fähigkeit den nächsten Output auf einer gesamten Satzhistorie zu konditionieren (erlaubt conditional generation models). Auch RNNs können nicht nur mit Textdaten verwendet werden.

$$s_i = R_{SRNN}(x_i, s_{i-1}) = g(s_{i-1}W^s + x_iW^x + b)$$

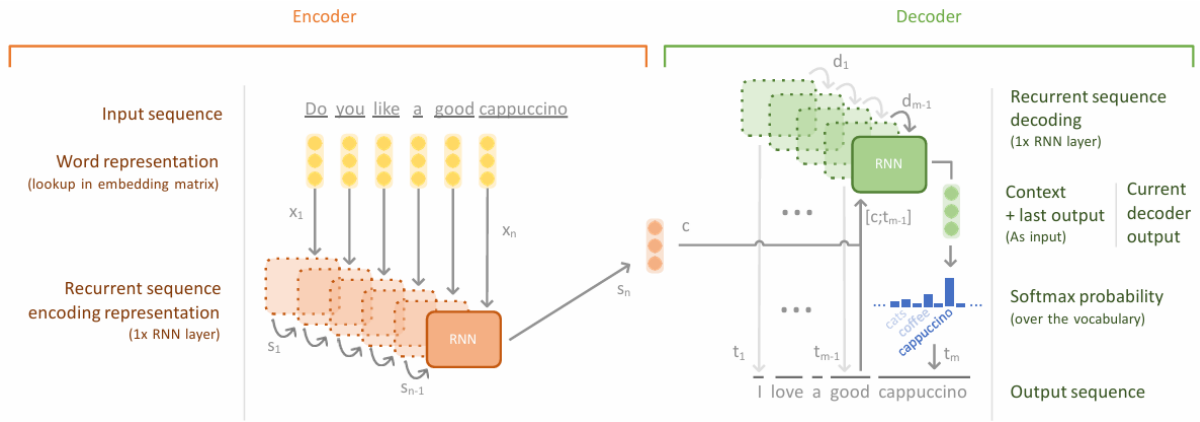
Wobei s_i der Zustand des RNN an Position i, x_i der Input Vektor an Position i, b der Bias Vektor, W die Gewichtsmatrizen und g eine nichtlineare Aktivierungsfunktion.



Verwendet man ein RNN als Encoder, nimmt man die Sequenz als Input, und erhält einen einzigen Output Vektor. Der Computation Graph ist durch die gepunkteten überlappten Boxen dargestellt. Das RNN ist immernoch ein Set an lernbaren Parametern. Optimal repräsentiert s_n die gesamte Sequenz.

Encoder-Decoder Architektur

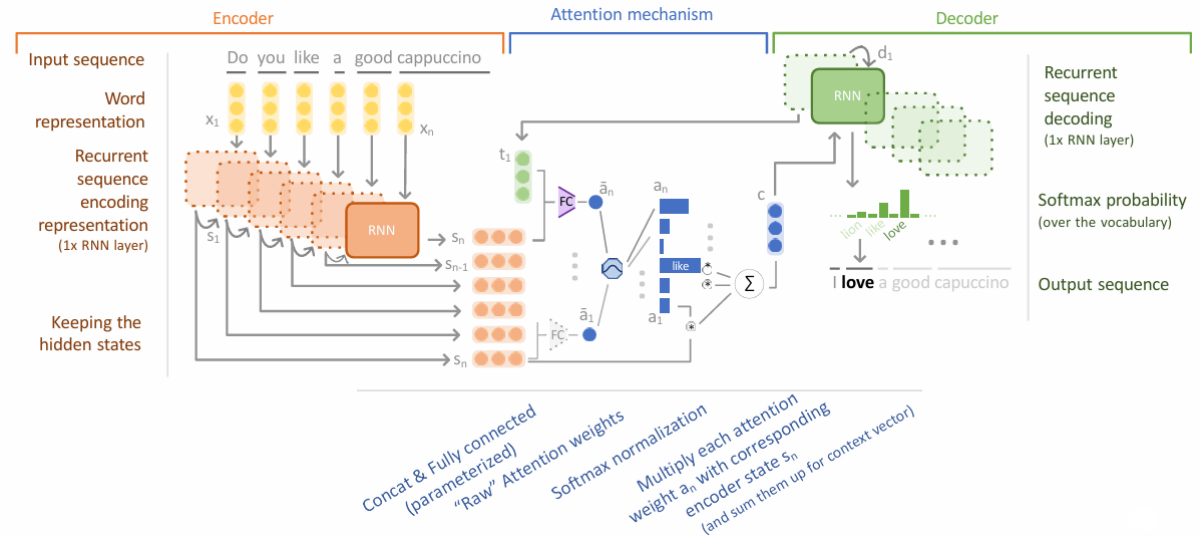
Vielseitige Architektur, die die Eingabe und Ausgabe von Sequenzen unterstützt. Basierend auf den Trainingsdaten (und einigen Optimierungen) für verschiedene Aufgaben verwendbar.



Attention Mechanismen erlaube es relevante Teile des Inputs zu suchen. Dafür wird ein gewichteter Durchschnitts-Kontext-Vektor erstellt. Die Gewichte basieren auf einem Softmax und summieren zu 1. Attention ist parametrisiert und end-to-end mit dem Modell trainiert.

Attention ist sehr effektiv, vielseitig und bietet ein wenige Interpretierbarkeit (Man kann zumindest sehen, welche Worte mehr Effekt auf den Output haben).

Jeder neue Decoder Zustand produziert einen neuen Kontextvektor. Die Encoder Zustände sind read-only und der fully connected Layer beinhaltet die gelernten Parameter und eine nichtlineare Aktivierung.



Transformer Architektur

Die Transformer Architektur ist nicht Aufgabenspezifisch, sondern arbeitet auf einer Sequenz von Vektoren, was wir mit diesen machen, ist uns überlassen.

Transformer sind schnell populär geworden und sind heutzutage notwendig in NLP und IR-Forschung.

Transformer lernen die Bedeutung basierend auf dem benachbarten Kontext für jedes Wortvorkommen. Diese Kontextualisierung verbindet Repräsentationen. Der Kontext ist hier

lokal zur Sequenz. Aber Transformer sind rechnerisch intensiv $O(n^2)$. Jeder Token ist von jedem anderen Token abhängig.

Transformer kontextualisieren mit mehreren Self-Attention Einheiten (units). Gängigerweise stapeln Transformer viele Layer und können als Encoder-only oder Encoder-Decoder eingesetzt werden. Außerdem benötigen sie keine Wiederholungen (Die Attention reduziert sich auf eine Reihe von Matrixmultiplikationen über die Sequenz).

```
encoder_layer = nn.TransformerEncoderLayer(d_model=300,nhead=10,dim_feedforward=300)
transformer = nn.TransformerEncoder(encoder_layer, num_layers=2)

src = torch.rand(10, 32, 300)
out = transformer(src)
```

BERT Pre-Training

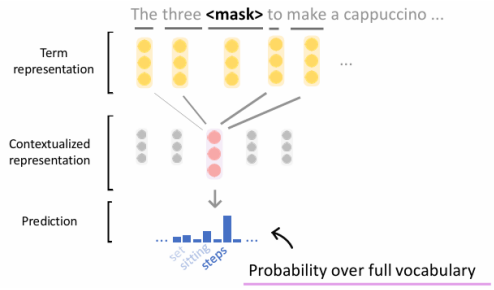
Bidirectional **E**ncoder **R**epresentations from **T**ransformers. Großer Effektivitätszuwachs über alle NLP Aufgaben.

Bestandteile:

- WordPiece Tokenization & Embedding (ähnlich zu BPE)
- Großem Modell (viele Dimensionen und Layer, Basis: 12 Layer und 768 Dimensionen)
- Spezielle Tokens (geteilte Nutzung zwischen Pre-Training und Fine Tuning)
  - **[CLS]**: Klassifizierungstoken, wird als pooling Operator verwendet, um einen einzelnen Vektor pro Sequenz zu erhalten
  - **[MASK]**: Verwendet im maskierten Sprachmodell um das Wort vorherzusagen
  - **[SEP]**: Markiert einen zweiten Satz (und Sequenz Enkodierung)
- Langes maskiertes Sprachmodellierungs Pre-Training (Wochen auf 1ner GPU)

Trainingsablauf:

- Text nehmen und zufällige Wörter maskieren
- Originales Wort mithilfe der Kontextwörter vorhersagen
- Gewichte basierend auf der Vorhersage und dem tatsächlichen Wort aktualisieren



Input: Entweder 1 oder 2 Sätze immer mit [CLS] vorangestellt. BERT fügt trainierte Position Embeddings und Sequenz Embeddings hinzu.

Model: Model ist relativ einfach mit n Layern von gestapelten Transformern. Jeder Transformerlayer bekommt den Output des vorherigen als Input. Der [CLS] Token ist nur speziell, weil wir ihn so trainieren. Im Modell wird dieser nicht von anderen Token unterschieden.

Workflow: Jemand mit viel Rechenleistung oder Zeit trainiert ein großes Modell vor, wir verwenden dieses und passen es auf unsere Aufgabe an.

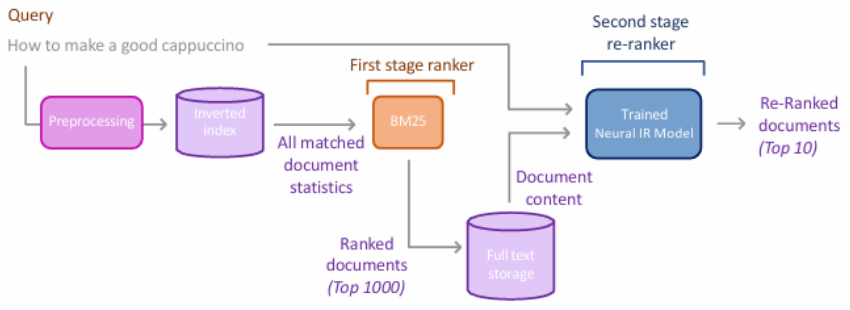
BERT++: Es gibt viele BERT Varianten für unterschiedliche Anwendungen.

SPLADE: Siehe Vorlesungssatz 4 Folie 21-23.

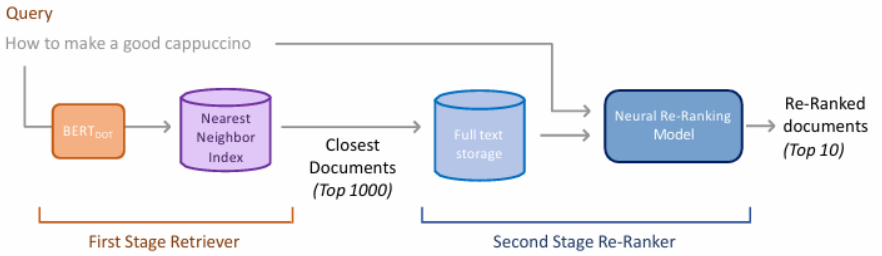
Dense Retrieval

Re-Ranking ist abhängig von einer Kandidatenauswahl (bottleneck). Dense Retrieval als BM25 alternative. Aber auch andere neurale Ansätze für first-stage Retrieval: Doc2query, DeepCT, COIL

Re-Rankers: Passen das Ranking von einer Vorausgewählten List von Ergebnissen an. Gleiches Interface wie die klassischen Ranking Methoden.



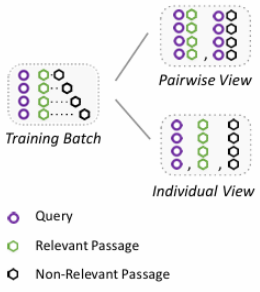
Dense Retrieval ersetzt die traditionelle erste Stufe. Mit neuralem Encoder und Vektor Index des nächsten Nachbarn und kann als Teil einer größeren Pipeline verwendet werden.



Wenn Dense Retrieval effektiv genug für unser Ziel ist, können wir es auch als alleinstehende Lösung betrachten, das ist deutlich schneller und weniger komplex, als mit dem Re-Ranking.

Neuronale IR Modelle werden typischerweise auf Triples (paarweise, +, -) trainiert. Ein Triple besteht aus 1ner Query und einem relevantem und einem unrelevantem Dokument und generiert Embeddings für diese drei Teile. Die Lossfunktion maximiert die Margin zwischen dem relevanten und unrelevanten Dokument. Alle Modellkomponenten werden end-to-end trainert.

Training Batches: Wir formen die Batches indem wir so viele Triples sampeln, wie unsere GPU erlaubt, typischerweise 16-128, wir mixen verschieden Queries zusammen. Abhängig vom Model müssen wir Query-Passage Pairs erstellen oder jede der drei Sequenzen einzeln durch das Modell schicken. Wir führen den Backward pass und das Gradient Update pro Batch durch. Sequenzielle Inputs kommen als einzelne Matrix, sodass wir die verschiedenen Inputlängen padden müssen.



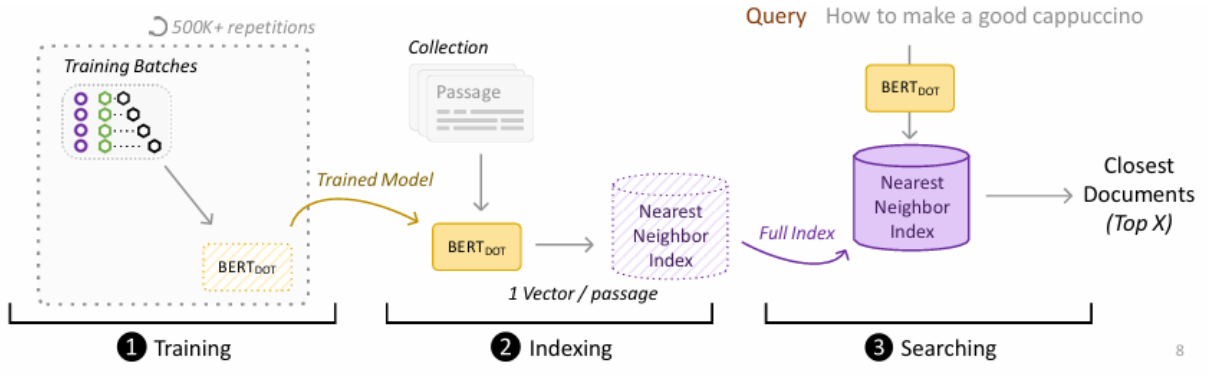
Die meisten Sammlungen enthalten nur Beurteilungen der **Relevanz** (oder falsch-positive Auswahlen aus anderen Modellen) und nicht wirklich nicht relevante Beurteilungen. Wir müssen dem Modell also mitteilen, was nicht-relevant ist. Einfache Methode um nicht relevante Passagen zu finden: BM25 ausführen und Top-1000 Ergebnisse, davon einige zufällig als nicht-relevant wählen.

Loss Funktion: Wahl verschiedene Methoden mit dem Ziel, den Abstand zwischen dem relevanten und nicht relevanten Dokument zu maximieren. Die folgenden beiden Methoden gehen von binärer Relevanz aus.

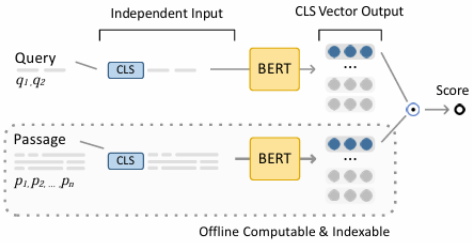
- Plain Margin Loss: $loss = \max(0, s_{nonrel} - s_{rel} + 1)$
- RankNet: $loss = BinaryCrossEntropy(s_{rel} - s_{nonrel})$

Dense Retrieval Lifecycle

3 primäre Phasen, jede mit jeweils komplexen Entscheidungen und notwendigen Techniken, Schritt 1 kann übersprungen werden, wenn man mit einem pre-trained Modell arbeitet.



$BERT_{DOT}$ Model: Passagen und Queries werden in einen einzigen Vektor zusammengefügt. Die Passagen sind komplett unabhängig. Query Encoding wird erst während der Laufzeit benötigt. Die Relevanz wird mit dem Dot-Produkt gescored.



Einfache Formel, solange wir BERT abstrahieren.

Encoding: $\hat{q} = BERT([CLS]; q_{1..n})_{CLS}, \hat{p} = BERT([CLS]; p_{1..m})_{CLS}$

Matching: $s = \hat{q} \cdot \hat{p}$

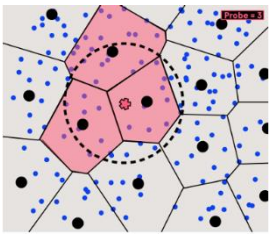
Nearest Neighbor Search: Wenn wir ein trainiertes DR-Modell haben, können wir jede Passage in unserer Collection enkodieren. Wir speichern die Passagen in einem angenäherten Nearest Neighbor Index. Während der Suche enkodieren wir die Query „on the fly" und suchen für die nearest Neighbor Vektoren im Passagen Index.

Indexing Techniken

Flat Index:

Einfach Brute Force, keine zusätzliche Verarbeitung, rohe Vektor Embeddings, berechnet die Distanz zu jedem Paar und ist daher langsam, aber eine vollumfangende Suche mit bester Genauigkeit.

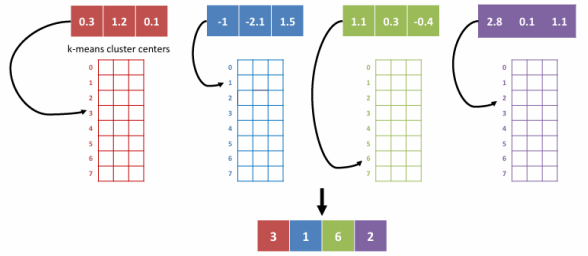
Inverted File Index (IVF):



Teilt den Datensatz in Cluster, mithilfe eines Cluster-Algorithmus (k-Means). Dann Centroid von jedem Cluster berechnen und diesen Centroid-Vektor speichern mit einer Inverted Index List von allen Vektoren, die zu diesem Cluster gehören.

Während der Query-Time wird die ähnlichkeit zwischen Query und Centroid berechnet, man wählt die top n Cluster und berechnet die Ähnlichkeit für alle Vektoren in diesen Clustern. Führt zu mehr Overhead aber deutliche Verkleinerung des Suchraumes.

Product Quantization:



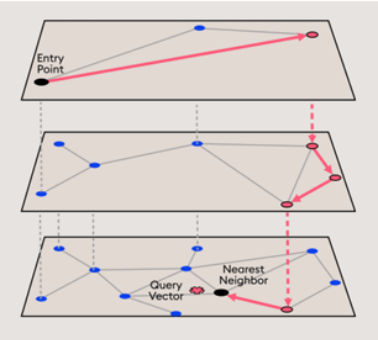
Idee: Den hochdimensionalen Float Vektor mit einem niedrigdimensionalen Integer Vektor ersetzen. K-Means Clustering auf jedem Subspace anwenden.

Reduziert n*d (embedding space) float Matrix zu n*m Integer Matrix. Man muss zusätzlich die Distanz von den Sub-Vektoren zu den Centroids speichern.

Zu Query-Time enkodieren wir die Query Vektoren auf den selben Weg. Wir approximieren die Distanz von der Query zum Dokument indem man die Summe der gespeicherten Distanz vom Dokument Sub-Vektor zum Query Cluster Centroid berechnet.

Product Qunatization ist sehr schnell und speichereffizient aber liefert nur angenäherte Ergebnisse und die Qualität ist von den Split und Clustering Parametern abhängig.

Graph Indices:



z.B. HNSW (Hierarchical Navigable Small Worlds)

Ein Proximity Graph, bei dem Vektoren mit ähnlichen Freunden verbunden sind. Die Suche startet an einem festgelegten Entry Point und besucht die Freunde, bis kein näherer Vertex gefunden wird. Der Suchraum ist in mehrere hierarchische Ebenen geteilt. Die oberste Ebene hat die längsten Distanzen. Ist man an einem lokalen Minimum, geht man eine Ebene weiter nach unten und sucht weiter, dies wiederholt man so lange, bis man am nächsten Nachbarn der niedrigsten Ebene ist.

Braucht zusätzliche Berechnungen und Speicher, aber skaliert besser auf sehr großen Datensätzen.

Brute-Force Suche skaliert nicht weiter als ein paar Millionen Vektoren, aber nearest Neighbor ist sehr verbreitet in ML, es gibt viele Techniken und Bibliotheken um die Suche zu beschleunigen. Eine populäre Bibliothek ist FAISS (bietet viele Algorithmen und CPU und GPU Unterstützung). Angenäherte Suche ist ein weiterer Tradeoff zwischen Latenz und Effektivität, wir fügen viel Komplexität in das Suchsystem, um eine niedrige CPU Latenz zu erhalten. Dense Retrievel bekommt immer mehr Unterstützung in Produktiv-Systemen (Viele Modelle auf HuggingFace verfügbar). Such Engines müssen Indizierung, Query Encoding und Nearest Neighbor implementieren. Bekannte Projekte: Vespa.AI & Pyserini.

$Bert_{DOT}$ kann für semantische Vergleiche von allen Suchen verwendet werden. S-BERT bietet viele Modelle un Szenarien. Adaptionen basierend auf Dot-Product Ähnlichkeit erlaubt Multi-Modale Vergleiche.

Knowledge Destillation

Die meisten Trainingsdaten sind verrauscht und nicht fein. Im Fall von MSMARCO haben wir nur eine relevant markierte Information pro Query, aber haben viele False-Negatives.

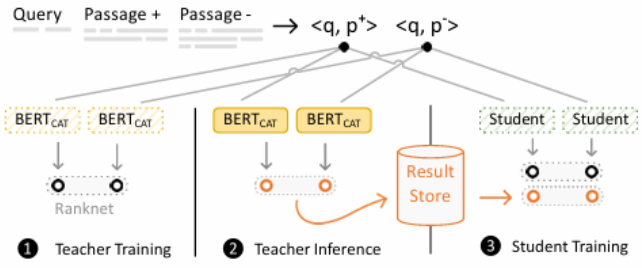
Teacher: Groß, starkes neurales Netzwerk, hohe Genauigkeit aber sehr langsam.

Student: kleineres, effizienteres Netzwerk, wird trainiert, um die Vorhersagen des Teachers anzunähern.

Wir können die finalen Output Scores als Supervisions Signal verwenden. Das macht es möglich, Architekturunabhängig zu trainieren. Es ist einfach ein Esemble von Teachern zu verwenden. Wir können auch Zwischenergebnisse in manchen oder allen Layern als Signale verwendet werden. Wir können so aber nur eine bestimmte Architektur verwenden und potenziell auch ähnliche Parameter Einstellungen. Wir haben deutlich mehr Supervisionssignale als nur den finalen Score.

DistilBERT: Eine distillierte, kleinere Version von BERT, hat dasselbe Vokabular und dieselbe allgemeine Anwendbarkeit. Hat nur 6 statt 12 Layer aber dieselbe Output Größe. Mit Knowledge Destillation trainiert, behält 97% Effektivität. DistilBERT als Basismodell für IR funktioniert sehr gut.

Das Trainingssetup bleibt das Gleiche. Wir trainieren einen Teacher auf einem binärem Loss. Wir verwenden die Scores des Teachers um ein Studentenmodell zu trainieren.



Margin-MSE

Potenzielle Verbesserung: Optimierung des Abstandes zwischen den relevanten und nicht-relevanten Passagen. Exakte Scores sind nicht relevant, nur die relative Differenz.

Loss: $L(Q, P^+, P^-) = MSE\big(Ms(Q,P^+) - Ms(Q,P^-), Mt(Q,P^+) - Mt(Q,P^-)\big)$

Der Loss macht keine Annahme über die Modelarchitektur, wir können also verschiedene neurale Ranking Methoden mischen. Wir können das Teacher Modell einmal vortrainieren und mehrfach wiederverwenden.

Wir können Margin-MSE auch für BERT_{DOT} Retrieval verwenden und liefert ziemlich gute Ergebnisse.

KL-divergence

Kullback-Leibler (KL) Divergenz wird verwendet, um Distributionen zu vergleichen. Für zwei Wahrscheinlichkeitsverteilungen P und Q, ist die KL-Divergenz:

$$D_{KL}(P||Q) = \sum_{x \in X} P(x) \log\left(\frac{P(x)}{Q(x)}\right)$$

Im Fall von Knowledge Destillation verwenden wir die Klassenwahrscheinlichkeiten des Teachers und des Studentennetzwerks als P und Q.

Wir wollen, dass Dense Retrieval Modelle **plug-and-play** sind. Dies wird als Zero-shot Transfer bezeichnet (weil wir keine Trainingsdaten von der Ziel Collection verwenden). Zero-Shot Szenarien sind schwerer als In-Domain Evaluation (weil es nur Generalisierung testet und nicht Memorisierung und Generalisierung zu mischen).

Der **BEIR**-Benchmark bringt viele IR Collections in ein Format.

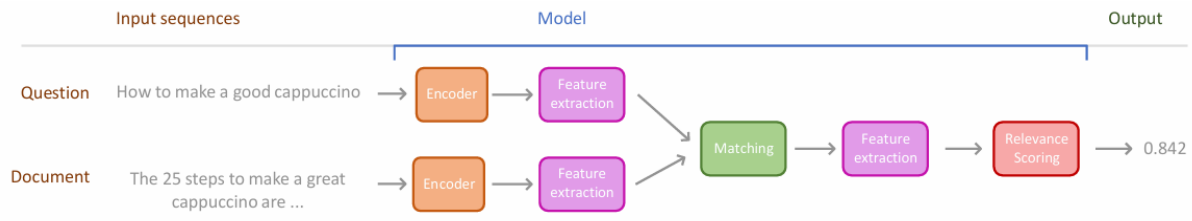
DR-Modelle haben **Schwierigkeiten mit Zero-Shot**. Es gibt hierfür verschiedene Erklärungen:

1. Generalisierung: DR Modelle generalisieren nicht zu anderen Query Verteilungen.

2. Eigenheiten: MSMARCO Trainingsdaten beinhalten zu viele Eigenheiten, spezifisch zu einer Kollektion.
3. Pool Bias: Viele ältere oder kleinere Collections sind stark biased für BM25 Ergebnisse.

Re-Ranking Methods

Wir haben die Re-Ranker bereits in eingeführt. Das Kernmodul von Re-Ranking Modellen ist ein Matching-Modul, welches auf einem Wort-Interaktions-Level arbeitet.

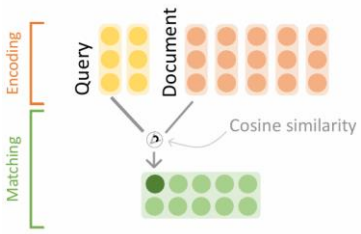


Das Training ist das gleiche wie bei Dense Retrieval.

Wir Evaluieren diese Methoden, indem wir auf einem Tupel aus einer Query und einem Dokument scoren. Die Liste der Tupel wird sortiert und evaluiert mit einer Ranking Metrik pro Query (z.B. MRR@10 Stop to look at Position 10 or first relevant). Wir können den Training Loss und die IR-Evaluationsmetrik nicht vergleichen. Wir können den Trainingsloss nur am Anfang des Trainings verwenden um zu sehen, ob das Netzwerk total unbrauchbar ist.

MatchPyramid

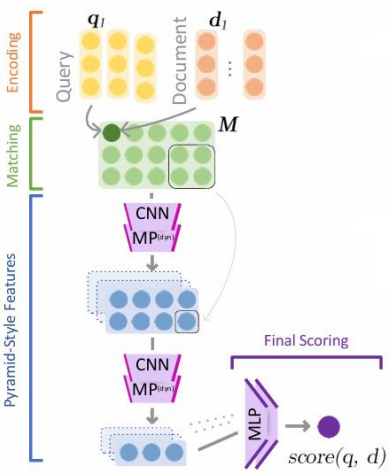
Encoding Layer: Ausgangspunkt für Textverarbeitungs Neurale Netzwerkmodelle. Enkodieren von Worttoken zu einer dichteren Repräsentation. Vor 2019 nutze man Word Embedding und ab 2019 dann BERT.



Die **Match Matrix** ist der Kern von vielen frühen neuronalen IR Modellen. Die Matrix zeigt die Ähnlichkeiten von individuellen Wortkombinationen und ist nur eine Transformation, welche selbst nicht Parametrisiert ist. Hierfür wird Cosine Similarity verwendet.

Cosine Similarity misst die Richtung von Vektoren aber nicht die Größe und ist an sich keine Distanz aber äquivalent zur euklidischen Distanz von Einheitsvektoren.

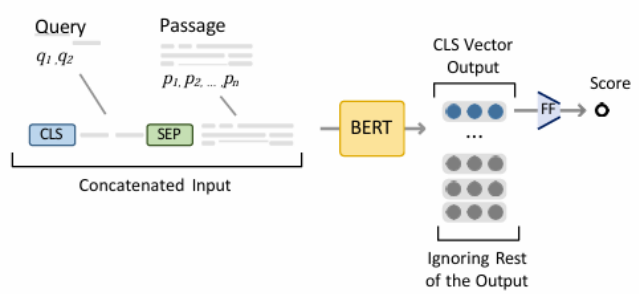
$$sim(d,q) = \cos(\theta) = \frac{d \cdot q}{|d||q|}$$



MatchPyramid wendet ein Set von 2D convolutional Layern auf die Match Matrix an. Jeder Conv Layer besteht aus 2D CNN und 2D-Dynamic Pooling. Die Architektur und Effektivität hängt stark von der Konfiguration ab (Wie viele Layer, welche CNN und Pooling Kernel Größen, Pooling Output wird langsam kleiner)

Die Convolutional Layer extrahieren lokale Interaktionsfeatures. Max Pooling behält nur starke Interaktionssignale (bessere Matches). Verschiedene Channel können verschiedene Interaktionsmuster lernen. Zum Schluss werden die extrahierten Feature Vektoren durch ein Multi-layered Feed Forward Module (MLP) gescored.

Re-Ranking with BERT



Wir betrachten nur Ad-Hoc Re-Ranking Modelle und fokussieren uns auf den Effizienz-Effektivität Tradeoff.

BERT Re-Ranking ($BERT_{CAT}$ oder monoBERT, vanilla BERT oder einfach BERT): Aneinanderhängen der beiden Sequenzen um zum BERT Workflow zu passen. ([CLS] query [SEP] passage). Wir poolen den [CLS] Token und machen eine Score Vorhersage mit einem linearen Layer. Dieser Workflow muss für jede Passage wiederholt werden.

BERT: $r = BERT([CLS]; q_{1..n}; [SEP]; p_{1..m})_{CLS}$, mit x_{CLS} dem CLS Vektorpooling

Scoring: $s = r \cdot W$, mit W dem linearen Layer

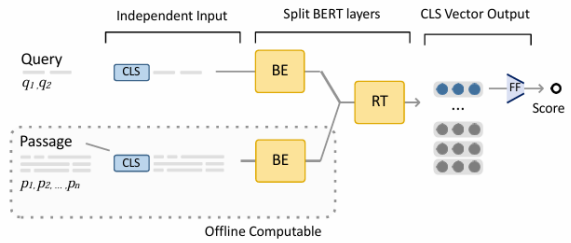
$BERT_{CAT}$ hat die aktuelle Welle von neuronalem IR gestartet und funktioniert sehr gut. Performanz siehe Foliensatz 6, Folie 19.

Mono-Duo Pattern ist ein Prozess mit mehreren Schritten, Mono ($score(q,p)$ Top-1000) und Duo ($score(q,p_1,p_2)$ Top-50). Duo macht einen Paarweisen Vergleich, welche Passage relevanter für die Query q ist. Das führt zu einer verbesserten Performance im Vergleich zum einfachen Mono Prozess.

$BERT_{CAT}$ kann maximal 512 Input Tokens verarbeiten (Query+ Dokument), man muss also längere Dokumente abschneiden oder ein Sliding Window auf dem Dokument einrichten (maximalen Window Score als Dokumenten Score verwenden). BERT ist sehr effektiv, aber sehr langsam.

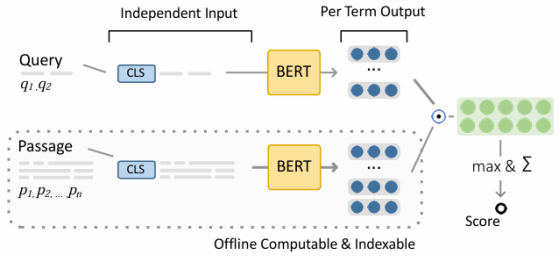
Um BERT effizienter zu machen können wir die Modellgröße reduzieren und die Passage Representation vorberechnen. BERT benötigt x (re-ranking Tiefe) volle Evaluationen währen der Query Time. Die meiste Berechnungszeit wird beim Passage Encoding benötigt. Wir betrachten hierzu zwei Ansätze.

PreTTR:



Wir teilen die BERT Layer auf, die ersten n Layer werden geteilt und alle nachfolgenden Layer wieder zusammengeführt. n ist ein Hyperparameter. So können wir die ersten n Layer der Passagen vorberechnen. Dieser Ansatz bietet ungefähr die gleiche Qualität wie BERT aber hat immernoch niedrige Query Latenz und hohe Speicheranforderungen.

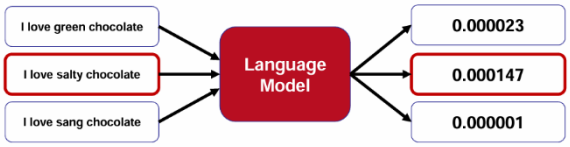
ColBERT:



ColBERT erstellt eine Match Matrix von BERT Term-Repräsentationen und verwendet dann einfaches Max Pooling für die Dokumenten Dimension und die Summe für die Query Dimension. Dieser Ansatz bietet sehr schnelle Query Latenz aber wir müssen alle Passagen-Term-Vektoren speichern (sehr große Speicheranforderung).

Encoding: $\hat{q}_{1..n} = BERT([CLS]; q_{1..n})$, $\hat{p}_{1..n} = BERT([CLS]; p_{1..n})$

Aggregation: $s = \sum_{i=i..n} \max_{t=1..m} \hat{q}_i \cdot \hat{p}_t$



Large Language Models

Langauge Modelle sind Modelle, die Sequenzen von Wörtern Wahrscheinlichkeiten zuweist. Und wird für Spracherkennung, Autokorrektur, und Sprachgenerierung verwendet.

N-Gram Modelle

Das einfachste LM basieren auf N-Grams (Zur Wiederholung: Sequenz von n Tokens (Wörter oder Satzzeichen)).

Dieses Modell basiert auf abhängigen Wahrscheinlichkeiten, wie wahrscheinlich folgt Wort w auf Sequenz s $P(w|s)$. Wir können diese Wahrscheinlichkeit annähern, in dem wir die Vorkommen zählen $\frac{C(sw)}{C(s)}$.

Das Problem ist, dass es die ganze Zeit neue Sätze gibt und wir nicht ganze Sätze zählen können. Die Joint Probability für lange Sequenzen ist schwer zu berechnen. N-Gram Modelle nähern diese Wahrscheinlichkeit an, indem die „Historie“ gekürzt wird (Markov Assumption).

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

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

Beispiel siehe Foliensatz 7, Folie 13-26

- Unigram Model: $P(w_{1:n}) \sim \prod_{k=1}^{n} P(w_k)$
- Trigram Model: $P(w_{1:n}) \sim \prod_{k=1}^{n} P(w_k|w_{k-2:k-1})$
- 4-gram Model: $P(w_{1:n}) \sim \prod_{k=1}^{n} P(w_k|w_{k-3:k-1})$
- 5-gram Model: $P(w_{1:n}) \sim \prod_{k=1}^{n} P(w_k|w_{k-4:k-1})$

Problem: Die Wahrscheinlichkeiten sind per Design <= 1. Das Multiplizieren dieser Wahrscheinlichkeiten führt zu sehr kleinen Zahlen und dies kann zu numerischem Underflow führen. Daher speichert man die log Wahrscheinlichkeiten anstelle der rohen Wahrscheinlichkeiten. Das Summieren ist im log Raum äquivalent zur Multiplikation im linearen Raum und die relative Reihenfolge bleibt die Gleiche. Wir machen diese Konvertierung nur rückgängig, wenn dies notwendig ist.

Extrinsic Evaluation: Modell in einer tatsächlichen Anwendung verwenden und die Verbesserung der Endaufgabe messen. Diese Evaluationen sind sehr teuer.

Intrinsic Evaluation: Auf einem Standard Testset evaluieren, wichtig ist, dass die Test Sätze nicht in im Trainingsdatensatz sind. Um verscheiden Ansätze und Parameter fine-Tuning zu machen, verwenden wir ein dritten Datensatz das Development oder Validations Set. Gängig ist ein 60/20/20 Split.

Wir evaluieren auf dem Test Datensatz, indem wir die Wahrscheinlichkeiten berechnen, die das Modell den Sätzen zuweist. Das Modell mit den höchsten Wahrscheinlichkeiten gewinnt. Anstelle der rohen Wahrscheinlichkeiten verwenden wir Perplexity (PPL). Für $W = w_1, w_2 \dots w_n$:

$$Perplexity(W) = \sqrt[n]{\frac{1}{P(W)}}$$

Minimieren wir Perplexity, maximieren wir die Wahrscheinlichkeit.

```
from transformers import AutoModelForCausalLM, AutoTokenizer
model = AutoModelForCausalLM.from_pretrained("gpt2")
tokenizer = AutoTokenizer.from_pretrained("gpt2")

inputs = tokenizer("ABC is a startup based in New York City and Paris", return_tensors = "pt")
loss = model(input_ids = inputs["input_ids"], labels = inputs["input_ids"]).loss
ppl = torch.exp(loss)
print(ppl)

Output: 29.48 # Model was trained on this sentence = lower PPL

inputs_wiki_text = tokenizer("Generative Pretrained Transformer is an opensource artificial
intelligence created by OpenAI in February 2019", return_tensors = "pt")
loss = model(input_ids = inputs_wiki_text["input_ids"], labels = inputs_wiki_text["input_ids"]).loss
ppl = torch.exp(loss)
print(ppl)

Output: 211.81 # Model did not see such sentenced before = higher PPL
```

Generation mit N-Gram Modellen

Beispiel siehe Foliensatz 7, Folie 35-43

Out-of-Vocabulary Problem

Ein großes Problem sind 0-Wahrscheinlichkeit N-Gams. In der Evaluation, kann das Testset N-Grams enthalten, die niemals im Training auftauchen (Perplexity is undefined). Führt dazu, dass das LM die Wörter im Prompt noch nie gesehen hat.

Unknown Words <UNK>: Entweder am Anfang ein Vokabular wählen und jedes out-of-vocabulary Wort in <UNK> ändern oder Wörter in den Trainingsdaten mit <UNK> basierend auf der Frequenz ersetzen.

Smoothing: Ein Teil der Wahrscheinlichkeiten auf ungesehen Events verteilen. Bei Lapplace/add-one Smoothing, bilden wir uns ein, dass wir jedes Wort einmal öfter gesehen haben, als wir es tatsächlich haben. Beispiel Unigrams: $P(w_i) = \frac{C(w_i)}{N}$ und $P_{Laplace}(w_i) = \frac{C(w_i)+1}{N+V}$ mit N Anzahl der Token und V Größe des Vokabulars.

Add-k Smoothing: Anstatt 1 zu jedem Vorkommen hinzuzufügen, fügen wir einen Bruchteil k hinzu (0.5, 0.05, 0.01). Dieser Parameter kann auf einem Development Set optimiert werden.

Backoff: Wir verwenden kleinere N-Grams, wenn wir keine Evidenz haben.

Interpolation: Immer eine gewichtete Kombination von verschiedenen N-Grams verwenden. Beispiel für trigrams: $P^*(w_n|w_{n-2}w_{n-1}) = \alpha_1 P(w_n) + \alpha_2 P(w_n|w_{n-1}) + \alpha_3 P(w_n|w_{n-2}w_{n-1})$ mit $\alpha_1 + \alpha_2 + \alpha_3 = 1$. Diese Parameter können auch auf dem Development Datensatz optimiert werden.

World-level vs Sub-word Tokenization

Word-level Tokenization hat Schwierigkeiten mit out-of-vocabulary Wörtern und hat Schwierigkeiten gut auf ungesehene oder seltene Terme zu generalisieren. Außerdem benötigt man ein größeres Vokabular, um alle Wortvariationen zu betrachten.

Deswegen verwendet man Sub-Word Tokenization, welches Wörter in kleinere Einheiten unterteilt, dies erlaubt ein fast komplette Abdeckung auch von ungesehenem Text. Ein Prinzip hierfür ist Byte-Pair Encoding.

WordPiece: Wurde von Google eingeführt und in BERT und DistilBERT verwendet. WordPiece ist ähnlich zu BPE aber unterscheidet sich, indem Symbolpaare gewählt werden, die die Likelihood-

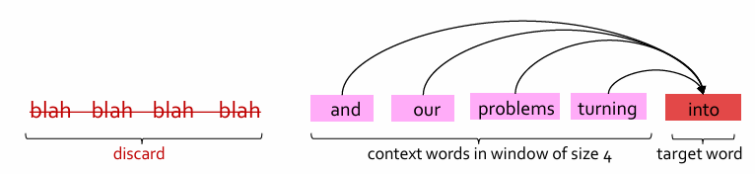
Wahrscheinlichkeit der Trainingsdaten maximiert und nicht das häufigste paar sein muss. Der Einfluss des zusammenführen wird gemessen, um zu vermeiden, dass sich die generelle Datenwahrscheinlichkeit reduziert. Beispiel: u und g nur zusammenführen, wenn $\frac{P(ug)}{P(u)\cdot P(g)}$ größer als für jedes andere Paar ist.

SentencePiece: BPE und WordPiece gehen davon aus, dass Wörter durch Leerzeichen getrennt sind, dies ist aber nicht für alle Sprachen der Fall. SentencePiece behandelt ein Leerzeichen wie jedes andere Zeichen, man benötigt keine vorherige Tokenization, das Leerzeichen ist bereits im Basisvokabular und dann kann man BPE oder andere Merge-Algorithmen verwenden. Somit ist diese Methode Sprachen unabhängig.

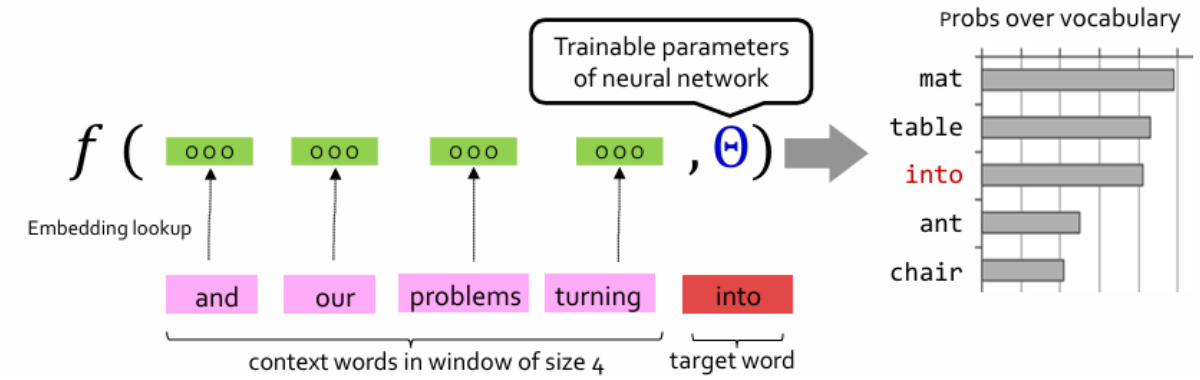
Neural LLM

Generell sind Zähl-basierte LMs ungenügend für Sprache, da Sprache Abhängigkeiten hat, die weit auseinander liegen können.

Aufgabe: Wir wollen das nächste Wort vorhersagen mit den vorgegeben Embeddings des Kontextes, alles außerhalb des Kontextfensters wird nicht benötigt.

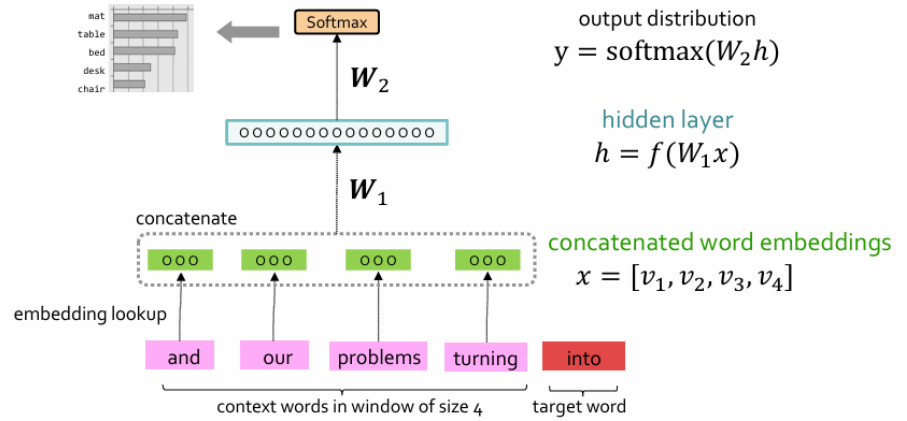


Wir trainieren das Modell, sodass wir die Paramater Θ optimieren, sodass wir die höchste Wahrscheinlichkeit für das Zielwort erhalten. Dies legt auch den Grundstein für die anderen Modelle, die wir betrachten (RNN, Transformer).



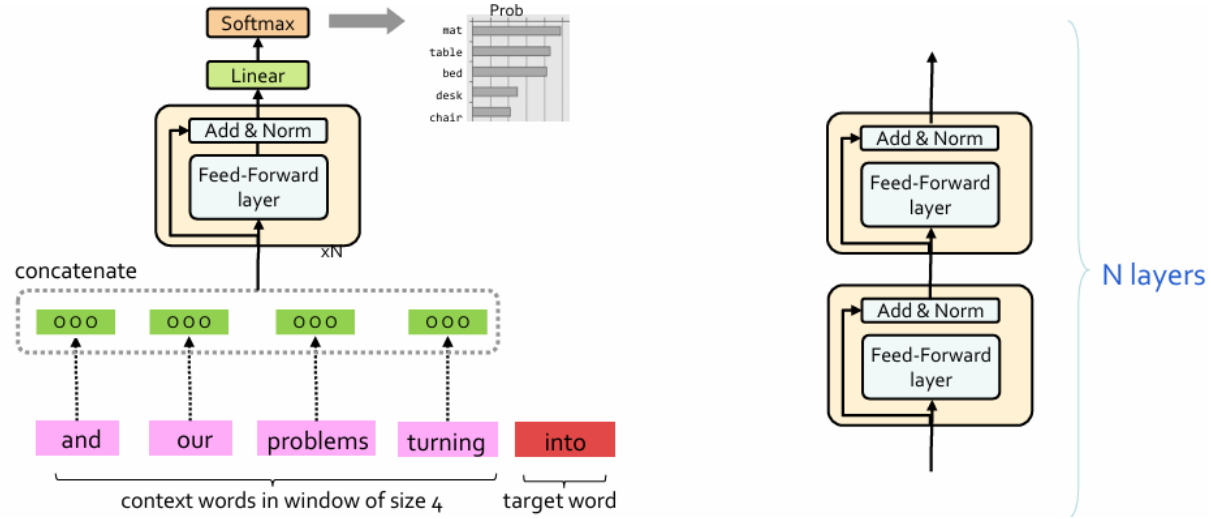
Für das Training müssen wir die folgenden 3 Schritte machen:

1. Jedes Wort in einen einzigartigen Index mappen
2. Jeden Index in einen One-Hot Vektor mappen
3. Nachschlagen des entsprechenden Word Embeddings mittels Matrixmultiplikation



Verbesserungen gegenüber n-gram LM: Bewältigt das Sparsity Problem. Die Modellgröße ist O(n) und nicht O(exp(n)), wobei n die Fenstergröße ist.

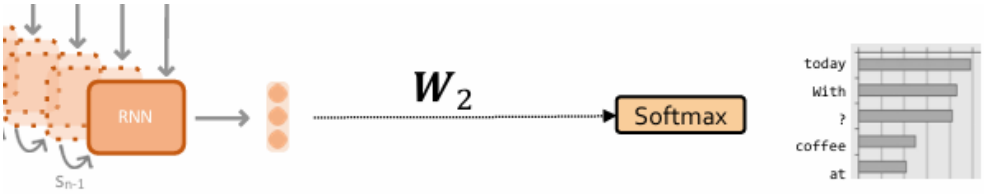
Probleme, die bleiben: Festgelegtes Fenster zu klein, Fenster vergrößern vergrößert auch W, das Fenster kann also nie groß genug sein und ist nicht tief genug, um kleine kontextuale Bedeutungen einzufangen.



In n-grams betrachten wir alle Präfixe unabhängig voneinander (auch die semantisch ähnlichen). Neurale LMs hingegen sind in der Lage Informationen über diese semantisch-ähnlichen Präfixe zu teilen und können so das Sparsity Problem bewältigen.

RNN und Transformer LM

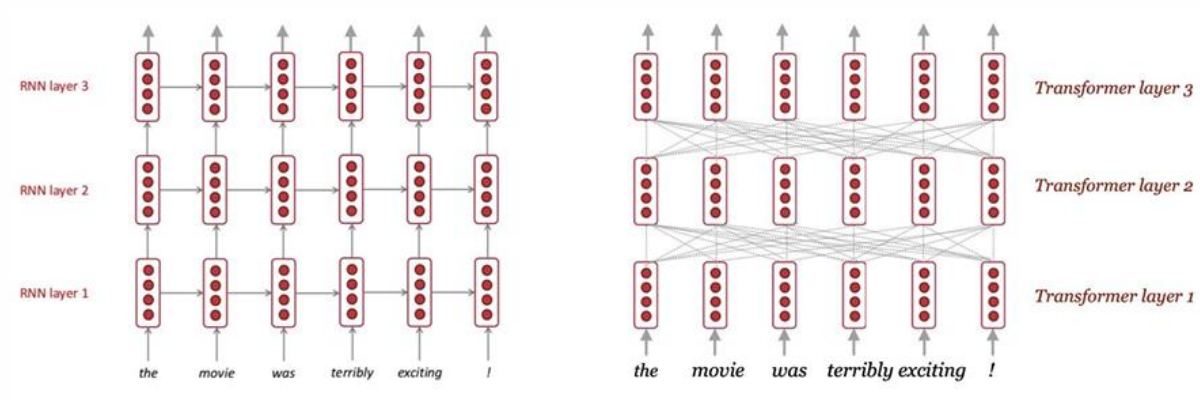
Wir erinnern uns an die Recurrent NN Struktur mit der Sequenz als Input und einem einzelnen Vektor als Output.



Nachteile:

- RNNs können in der Theorie lange Sequenzen repräsentieren aber vergessen schnell Teile ihres Inputs.

- Verschwindende oder Explodierende Gradienten
- Schwierigkeiten zu Parallelisieren

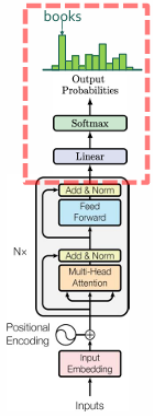


Self-Attention

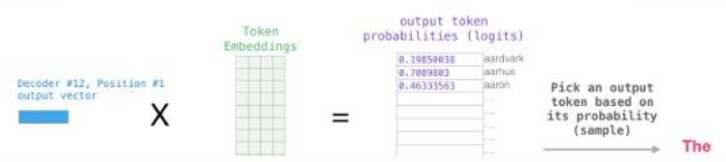
Attention ist ein starker Mechanismus, um Kontextbewusste Repräsentationen zu erstellen und auf bestimmte Teile des Inputs zu fokussieren. Sind besser um weit verteilte Abhängigkeiten im Kontext zu erhalten.

Layertyp	Komplexität pro Layer	Sequenzielle Operationen
Self-Attention	$O(n^2 \cdot d)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$

Wobei n die Sequenzlänge und d die versteckten Dimensionen sind.



Wir erhalten ein Tiefes Netz indem wir mehr Layer stapeln.



Um die Prediction zu machen, müssen wir einen Klassifikationskopf auf den finalen Layer des Transformers hinzufügen. Das kann entweder pro Token (Sprachmodellierung) oder für die gesamte Sequenz gemacht werden (nur ein Token).

$$out \in \mathbb{R}^{S \times d} \text{(S: Sequenzlänge)}$$

$$logits = Linear_{(d,V)}(out) = f(out.W_V) \in \mathbb{R}^{S \times V}$$

$$probabilities = softmax(logits) \in \mathbb{R}^{S \times V}$$

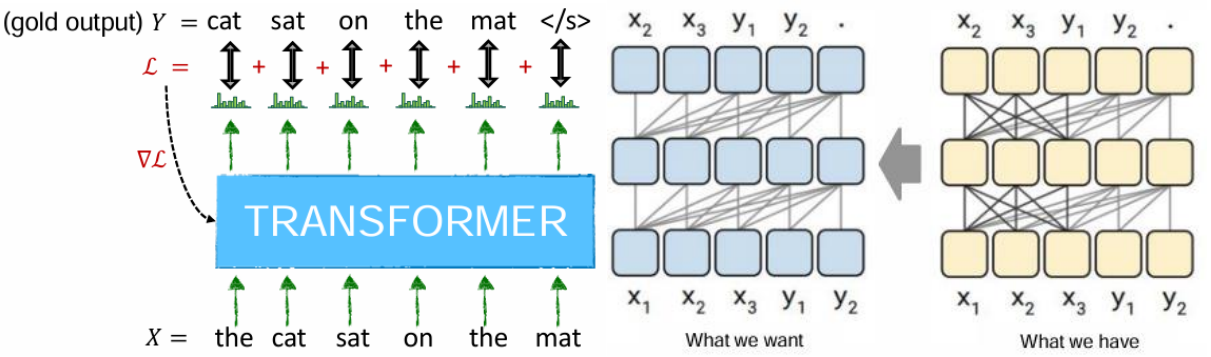
Training a Transformer LM

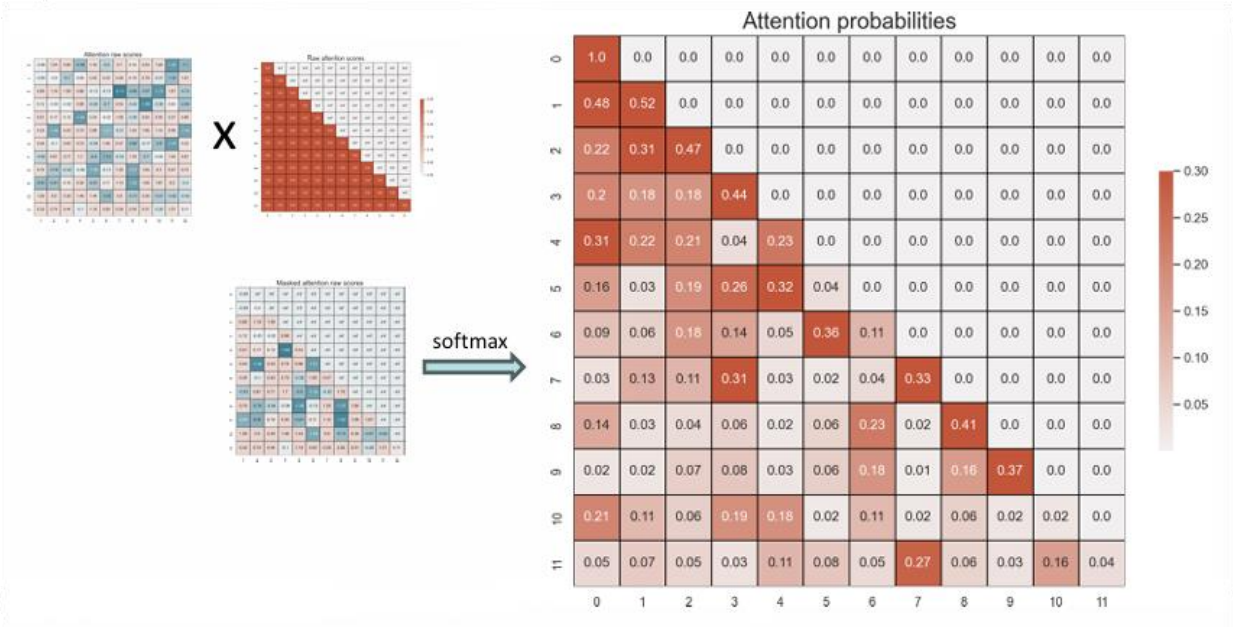
Ziel: Transformer trainieren für Sprachmodellierung.

Ansatz: Die Transformer werden so trainiert, dass jede Position der Vorhersager für den nächsten Token ist. Wir shiften den Input immer um eins weiter und nutzen das als Label.

- Für jede Position berechnen wir die zugehörige Verteilung über das gesamte Vokabular.
- Für jede Position berechnen wir den Loss zwischen den Wahrscheinlichkeiten und dem goldenen Output label.
- Damm summieren wir die positionsweisen Losswerte um den globalen Los zu erhalten

- Mit diesem Loss können wir Backpropagation machen und die Transformerparameter updaten.
- Das Model löst die Aufgabe, in dem der nächste Token zum Output kopiert wird (data leakage)
- Wir müssen Informations-Leakage von zukünftigen Tokens verhindern, also verwenden wir Masking mit der Attention Probability Matrix im Transformer.



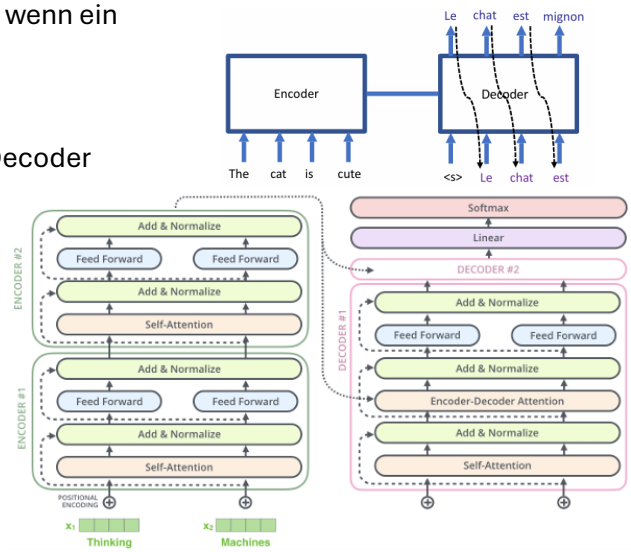


- Der Output des vorherigen Schrittes wird als Input für den nächsten Schritt verwendet.
- Die Wahrscheinlichkeiten werden geupdaten wenn ein neuer Token zum Input hinzugefügt wird.

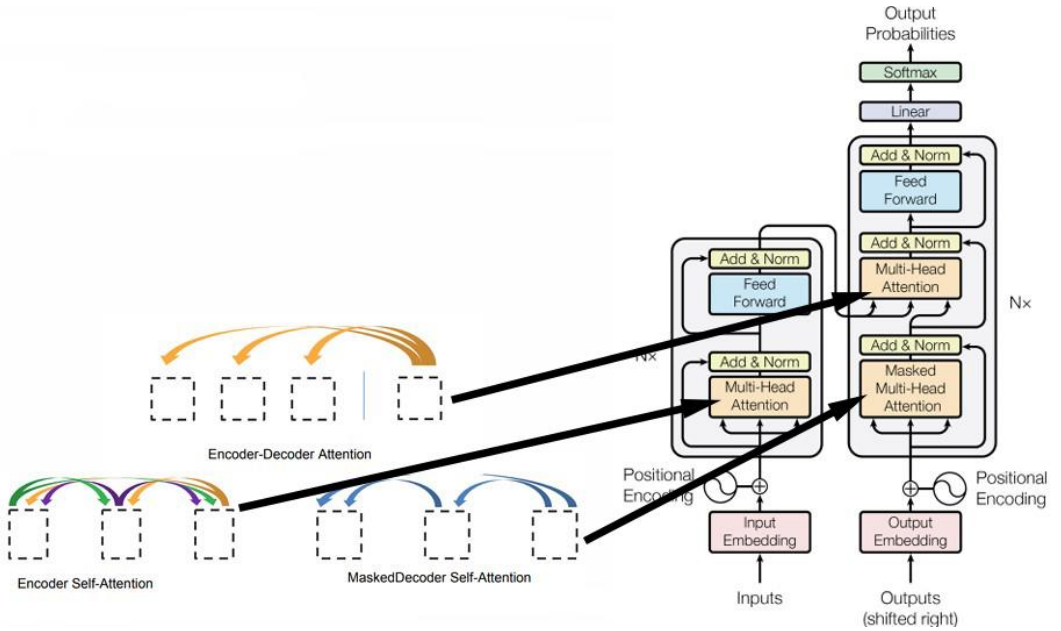
Encoder-Decoder in Transformator Models

Der Encoder ließt oder enkodiert den Input und der Decoder generiert oder dekodiert den Output.

Encoder Decoder Transformerarchitektur mit Attentionmodulen:



Dabei wirkt die Attention der Encoder von beiden Seiten. Bei jedem Decoder schritt wirkt sich die Attention von dem vorherigen Encoder aus. Bei jedem Decoder wirkt sich auch die Attention der vorherigen Decoder aus.



Adaption

LLMs sind auf massiven Datenmengen vortrainert. Diese LLMs machen nicht unbedingt nützliche Dinge. Wir passen ein Model auf unseren Use-Case an, indem wir...

- **Tuning:** ... die Modelparameter anpassen.
- **Prompting:** ... die Modellinputs anpassen (Sprach Statements)

Wir können die Modelle für bestimmte Aufgaben tunen, z.B. für Klassifikation, QA, Übersetzung und Zusammenfassung.

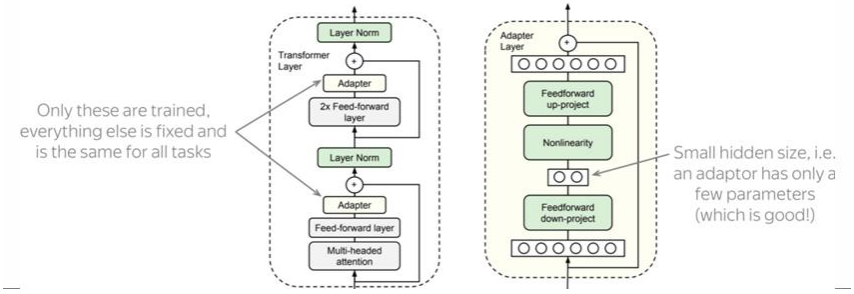
Tuning des gesamten Modells: Eine Optimierung ausführen, die alle Modelparameter updatet, **die auf den Aufgabedaten definiert sind.**

Head-Tuning: Eine Optimierung ausführen, die die Parameter des Model Heads updatet, die auf den Aufgabedaten definiert sind

Erweiterung des bestehenden vortrainierten Modells um zusätzliche Parameter oder Schichten und Training nur der neuen Parameter.

Adapter

Idee: Kleine Sub-Netze trainieren und nur diese tunen. Der Adapter layer projiziert in einen niedrig-dimensionalen Raum, um Parameter zu reduzieren. Wir müssen also nicht das volle Modell für jede Aufgabe speichern sondern nur die Adapterparameter.



Parameter-effizientes Tuning ist nicht schneller, da der gesamte Forward- und Backwardpass trotzdem nötig ist. Es ist jedoch speichereffizienter, weil nur die feingetunten Parameter gespeichert werden müssen.

Selektive Methoden fine-tunen ein Subset der bestehenden Modellparameter. Diese Auswahl kann Layer-Tiefen-Basiert sein, Layer-Typen-Basiert oder sogar Individuelle Parameter.

BitFit: tuned die Bias Terme in Self-Attention und MLP Layern und updatet nur ca 0.05% der Modellparameter.

$$\mathbf{Q}^{m,\ell}(\mathbf{x}) = \mathbf{W}_q^{m,\ell}\mathbf{x} + \mathbf{b}_q^{m,\ell}$$
$$\mathbf{K}^{m,\ell}(\mathbf{x}) = \mathbf{W}_k^{m,\ell}\mathbf{x} + \mathbf{b}_k^{m,\ell}$$
$$\mathbf{V}^{m,\ell}(\mathbf{x}) = \mathbf{W}_v^{m,\ell}\mathbf{x} + \mathbf{b}_v^{m,\ell}$$

$$\mathbf{h}_2^\ell = \text{Dropout}\left(\mathbf{W}_{m_1}^\ell \cdot \mathbf{h}_1^\ell + \mathbf{b}_{m_1}^\ell\right) \quad (1)$$
$$\mathbf{h}_3^\ell = \mathbf{g}_{LN_1}^\ell \odot \frac{(\mathbf{h}_2^\ell + \mathbf{x}) - \mu}{\sigma} + \mathbf{b}_{LN_1}^\ell \quad (2)$$
$$\mathbf{h}_4^\ell = \text{GELU}\left(\mathbf{W}_{m_2}^\ell \cdot \mathbf{h}_3^\ell + \mathbf{b}_{m_2}^\ell\right) \quad (3)$$
$$\mathbf{h}_5^\ell = \text{Dropout}\left(\mathbf{W}_{m_3}^\ell \cdot \mathbf{h}_4^\ell + \mathbf{b}_{m_3}^\ell\right) \quad (4)$$
$$\text{out}^\ell = \mathbf{g}_{LN_2}^\ell \odot \frac{(\mathbf{h}_5^\ell + \mathbf{h}_3^\ell) - \mu}{\sigma} + \mathbf{b}_{LN_2}^\ell \quad (5)$$

Häufig braucht man einen großen gelabelten Datensatz, wobei diese Anforderung häufig durch Pre-Training reduziert werden kann.

In-Context Learning

Lernen einer Downstream Aufgabe durch Konditionen von Input-Output Beispielen.

Keine Gewicht-Updates: Das Modell wird nicht vor-trainiert, um von Beispielen zu lernen. Die grundliegenden Modelle sind relativ generell.

Pre-Trained LMs imitieren Beispiele gegeben ihren Kontext.

Große Varianz in Performanz in Abhängigkeit von der Enkodierung. Also die Wahl von Demonstrationen, die Reihenfolge, Wording etc. Die Enkodierungen sollten nicht auf den Testdaten basieren.

Generell wollen wir eine Enkodierung, die unsere Aufgabe ähnlich zu LM machen.

Praktisch nützlich

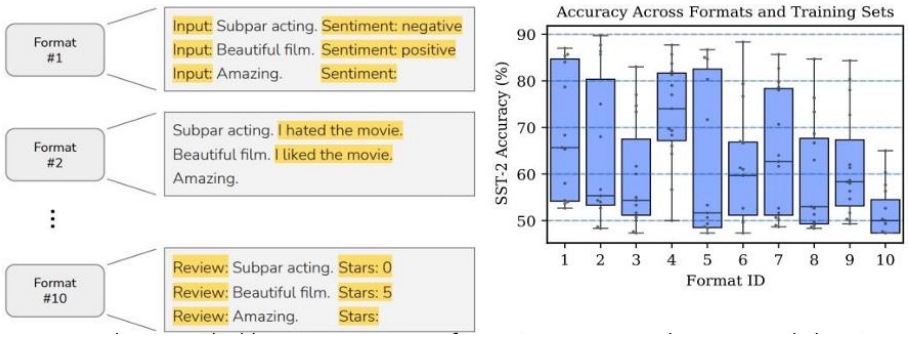
Daten zu labeln ist teuer, man benötigt Domain Wissen (Medizinisch, Legal, Finanzen), und wollen nicht mehr Daten aber schnell auf aufkommende, zeitkritische Szenarien reagieren.

Finetuning kann schwierig sein. Das Training ist sensitiv auf die Hyperparameter abgestimmt. Häufig haben wir nicht genug Validierungsdaten und es ist nicht ganz klar, wie Finetuning funktioniert. Außerdem sind Modelle teuer zu trainieren (Zeit und Speicher).

Intellektuell interessant

Potenzialtest für Intelligentes Verhalten: Generalisieren von wenigen Beispiel ist fundamentales Teil von Intelligenz und wir häufig in der Psychologie verwendet und kann sich schnell an die Umgebung anpassen.

Einblicke in Sprachmodellierung, was weiß die LLM und was sind die Biases und Grenzen von LLMs.



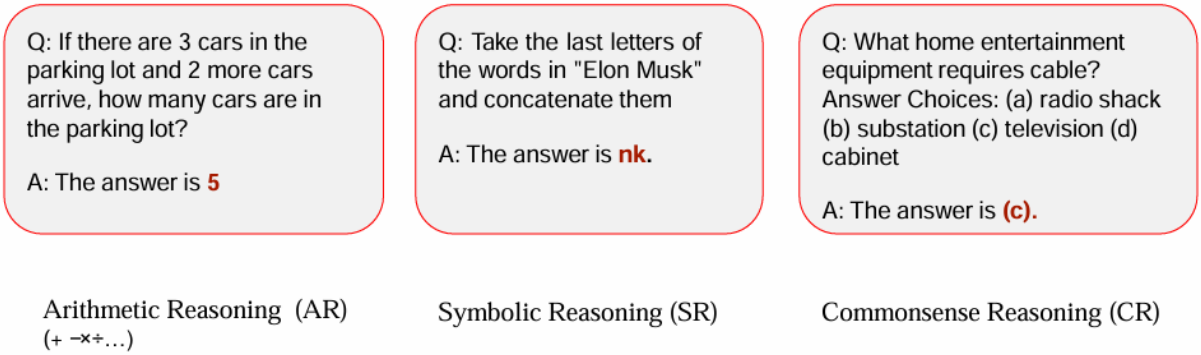
In-Kontext Lernen ist hoch empfindlich gegenüber dem Format des Prompts (Trainingsdatensätze und Muster/Verbalizers).

Majority Label Bias: Häufige Trainingsantworten dominieren die Vorhersage.

Recency Bias: Beispiele in der Nähe des Prompt-Endes dominieren die Vorhersgae, das erklärt die Varianz über Beispielpermutationen.

Reasoning

Mache Probleme erfordern Resoning. Man unterscheidet 3 Typen von Reasoning.



3 Fine-Tuning Ansätze:

- **Vanilla ICL (In-Context Learning) für Reasoning-Probleme**: Hier gibt man dem Modell einige Beispiele (Prompts) mit Eingaben und richtigen Antworten. Das Modell lernt aus diesen Beispielen, ohne dass die Gewichte aktualisiert werden. Es versucht, das Muster zu erkennen und auf neue Eingaben anzuwenden.
- **CoT (Chain-of-Thought)**: Anstatt direkt eine Antwort zu geben, wird das Modell dazu gebracht, den Denkprozess schrittweise aufzuschreiben. Das hilft besonders bei komplexen Reasoning-Aufgaben, da das Modell sich selbst Zwischenschritte herleitet.
- **Zero-Shot CoT**: Das Modell wird ohne Beispiele (zero-shot) verwendet, aber mit einer speziellen Aufforderung wie „Denke Schritt für Schritt nach“. Dadurch wird es angeregt, eine schrittweise Argumentation zu liefern, obwohl es keine direkten Beispiele erhalten hat.

Die Self-Consistency Methode hat 3 Schritte:

1. LM prompt mit CoT
2. Sampeln vom Decoder des LM um ein diverses Set an Reasoning Pfaden zu generieren
3. Wähle die am meisten übereinstimmende Antwort mittels Majority/Plurality Vote.

Multi-Step Prompting

LMs darum zu bitten ihre Antwort zu erklären verbessert ihre Performanz, aber die Schritte sind nicht immer korrekt.

Alignment

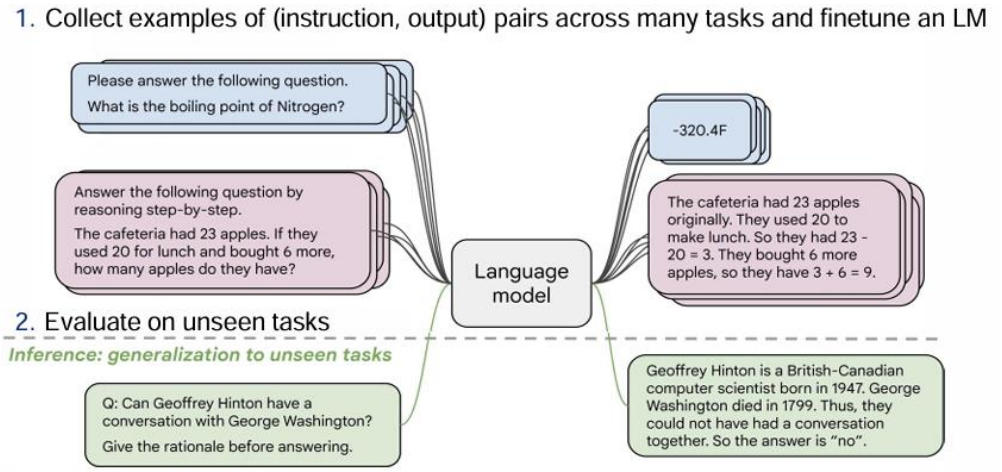
Es besteht eine Diskrepanz zwischen dem LLM-Pre-Training und den Benutzerabsichten und Menschlichen Werten.

Ocford Dictionary: „The result of arranging in or along a line, or into appropriate relative positions; the layout or orientation of a thing or things disposed in this way“

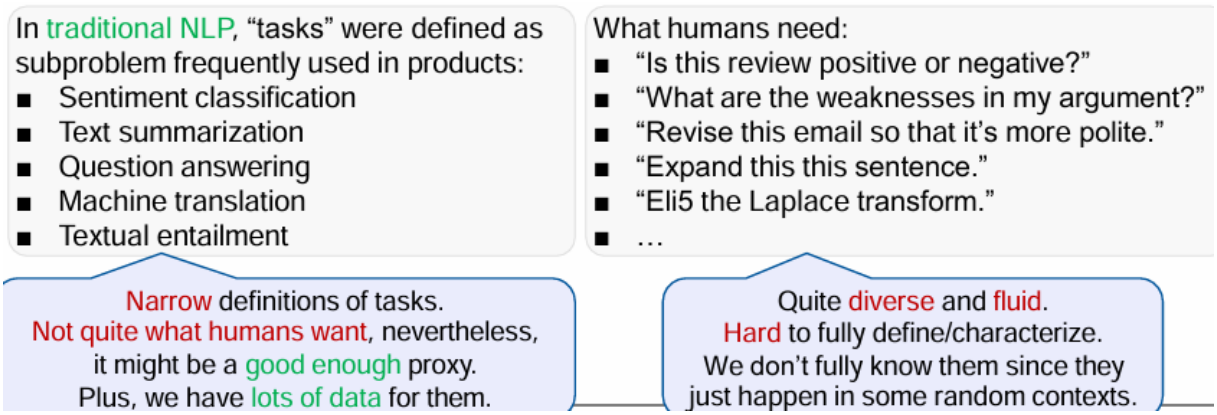
Es reicht nicht aus, wenn die KI das tut, was wir sie fragen.

Instruction Tuning

Finetuning von Sprachmodellen anhand einer Sammlung von Datensätzen, die eine Zuordnung von Sprachinstruktionen auf die entsprechenden wünschenswerten Generationen.



Gelabelte Daten sind hier der Schlüssel. Gute Daten müssen viele verschiedene Aufgaben repräsentieren.



Instruction Tuning bringt also die folgenden Vorteile mit sich:

- Verbesserte Performanz im Folgen von Anleitungen in Zero-Shot Szenarien
- Skalierung der Datengröße für die Befehlsabstimmung verbessert die Performanz
- Die Vielfalt der Prompts ist entscheidend.

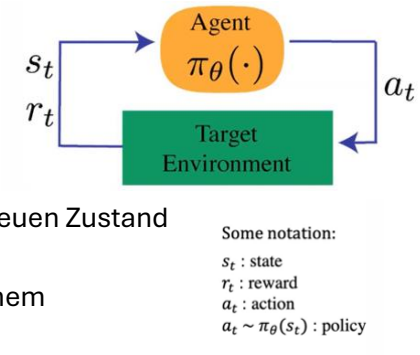
- Verglichen mit Pre-Training ist Instruction Tuning mit geringen Kosten verbunden (in der Regel verbraucht es <1% des gesamten Trainingsbudgets)

Aber es gibt auch Nachteile:

- Es ist teuer die Ground-Thruth-Data für die Aufgaben zu sammeln
- Dies ist besonders schwierig bei offener kreativer Generierung, bei der es keine richtige Antwort gibt.
- Anfällig für Halluzinationen

Reinforcement Learning

Intuition: Wir haben eine Aktion (Antwort Generieren) und eine Belohnung (Ob dem Mensch diese Antwort gefällt oder nicht)



- Ein Agent interagiert mit einer Umgebung, indem er Aktionen ausführt.
- Die Umgebung liefert eine Belohnung für die Aktion und einen neuen Zustand (Darstellung der Welt zu diesem Zeitpunkt).
- Der Agent verwendet eine Policy-Funktion, um eine Aktion in einem bestimmten Zustand zu wählen.
- Wir haben eine Reward-Funktion, die diese Entscheidung bewertet.

Human Feedback

Wir haben eine Reward-Funktion $R(s; prompt) \in \mathbb{R}$ für jeden Output s zu einem Prompt. Der Reward ist größer, wenn der Mensch die Antwort bevorzugt. Gute Generation ist äquivalent zum Finden des maximalen Rewards.

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

Mit $\mathbb{E}$ einer empirischen Erwartung, $\sim$ bedeutet Sampeln aus einer gegeben Verteilung, $\mathbb{E}_{\hat{s}\sim p_\theta}$ der erwartete Reward im Laufe des Samplings aus der Policy (Generatives Modell) und $p_\theta(s)$ einem Pre-Trained Modell mit Parametern θ, welche wir optimieren möchten (Policy Funktion)

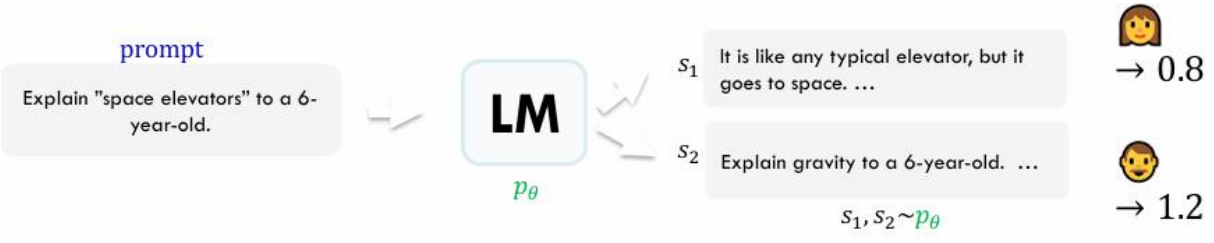
Wir wollen also die Reward-Function R schätzen und das beste generative Modell finden, dass den erwarteten Reward maximiert:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}}\, \mathbb{E}_{\hat{s}\sim p_\theta}[R(\hat{s}; prompt)]$$

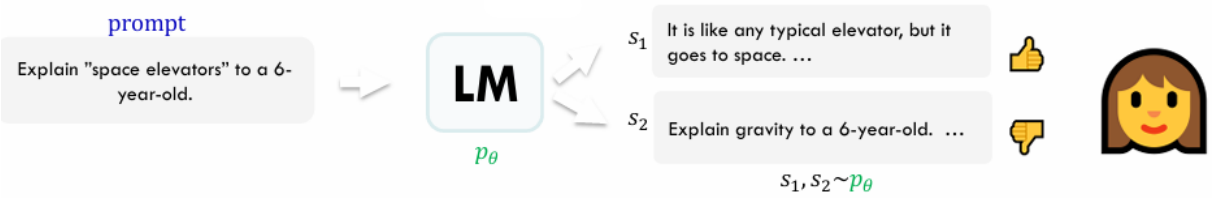
Reward R schätzen

Offensichtlich können wir kein direktes Menschliches Feedback nehmen, da dies sehr teuer ist, daher nehmen wir ein Model, welches die Menschlichen Vorlieben abbildet.

Ansatz 1: Menschen stellen absolute Scores für jeden Output zur Verfügung, aber Menschliches Urteil kann auf verschiedenen Instanzen und bei verschiedenen Menschen verrauscht oder schlecht kalibriert sein.

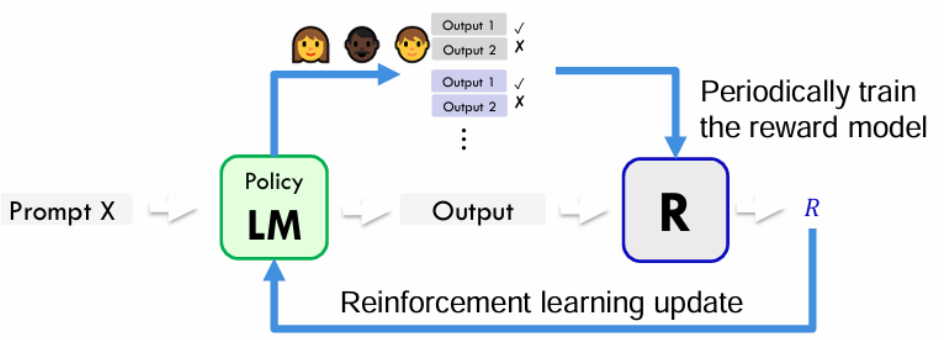


Ansatz 2: Der Mensch vergleicht paarweise Ergebnisse.



Training mit Human Feedback

- Zunächst sammeln wir ein Datensatz an Menschlichen Vorlieben, indem wir mehrere Outputs dem Menschen bereitstellen und diese anhand seiner Präferenz beurteilt.
- Mit diesen Daten können wir ein Reward-Modell trainieren. Das Reward Modell produziert einen skalaren Reward, welcher numerisch die Vorliebe des Menschen darstellen soll.
- Wir lernen eine Policy (ein Sprachmodell), die den Reward des Reward Modells optimiert.
- Wir trainieren das Reward-Model periodisch mit mehr Beispielen und Menschlichen Feedback.



Dieser Ansatz hat aber das Problem, dass die Policy lernt zu cheaten, sie lernt Output zu produzieren, der einen hohen Reward bekommt aber unrelevant oder Unsinn ist. Da R auf natürlichem Input trainiert wird, kann er nicht gut auf unnatürlichem Input generalisieren.

Lösung: Eine Bestrafung für zu starke Abweichung von der Verteilung des vortrainierten Modells.

$$\hat{R}(s;p) := R(s;p) - \beta \log\left(\frac{p^{RL}(s)}{p^{PT}(s)}\right)$$

Das Reward Modell kann dafür genutzt werden, das gewünschte Verhalten zu beeinflussen. zB. Biases verhindern, Antworten außerhalb des Scopes verhindern, Toxizität verhindern, etc.

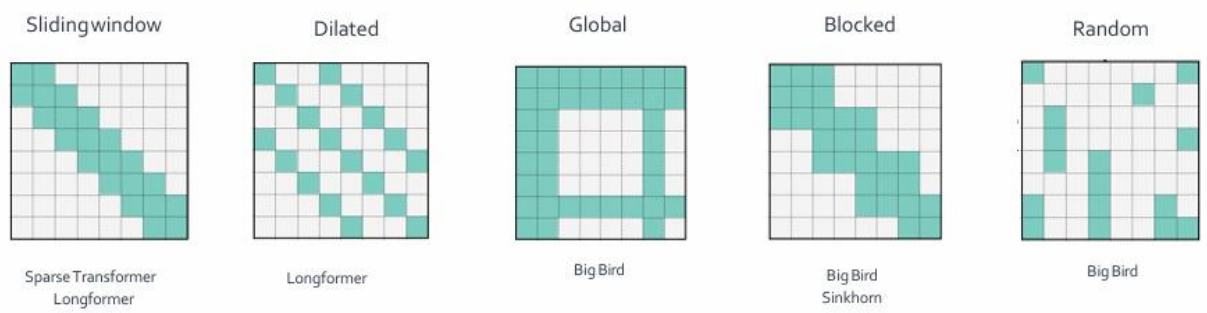
Fazit: RL ist nicht einfach und ein gutes Reward-System benötigt viel Menschliche Annotation.

Long Contexts

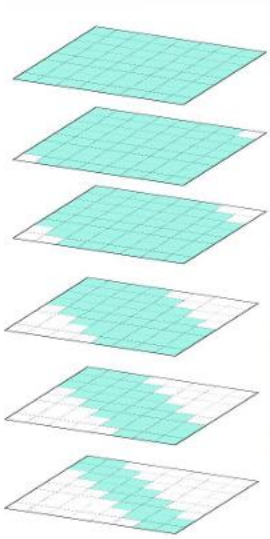
Viele Dokumente haben deutlich mehr als 2000 Tokens. Dieses Limit kann man umgehen, indem man größere Modelle mit größerer Kontext Window Größe trainiert aber die Speichernutzung und die Anzahl der Rechenoperationen steigt quadratisch bei der Self-Attention.

Sparsity Patterns

Ein Ansatz zur Verbesserung der Effizienz sind (Fixed) Sparse Attention Patterns.



Verschiedene Layer und Attention Heads können verschiedenen Mustern folgen. Ein häufiges Setup ist frühere Layer mit sparseren Attention Pattern zu haben.



Retrieval Augmented Generation

Retrieval-based Language Models sind Modelle, die Daten von externe Datenquellen beziehen (zumindest während der Inference Zeit). LLMs können nicht alles (long-tail) Wissen in ihren Parametern merken. Das Wissen von LLMs ist schnell veraltet und schwer zu updaten. Der Output ist schwer zu interpretieren und verifizieren.LLMs sind groß und teuer zu trainieren und auszuführen.

Ein Retriever ist eine Funktion $f(input, memory) \rightarrow score$

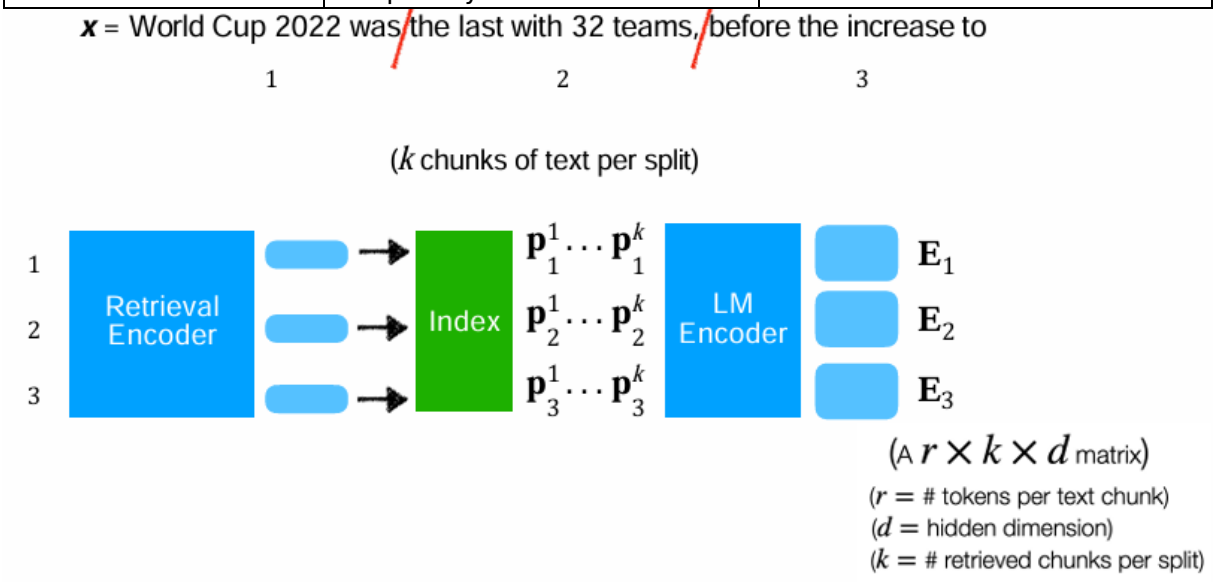
RAG wird verwendet, damit LM externes Wissen verwenden kann.

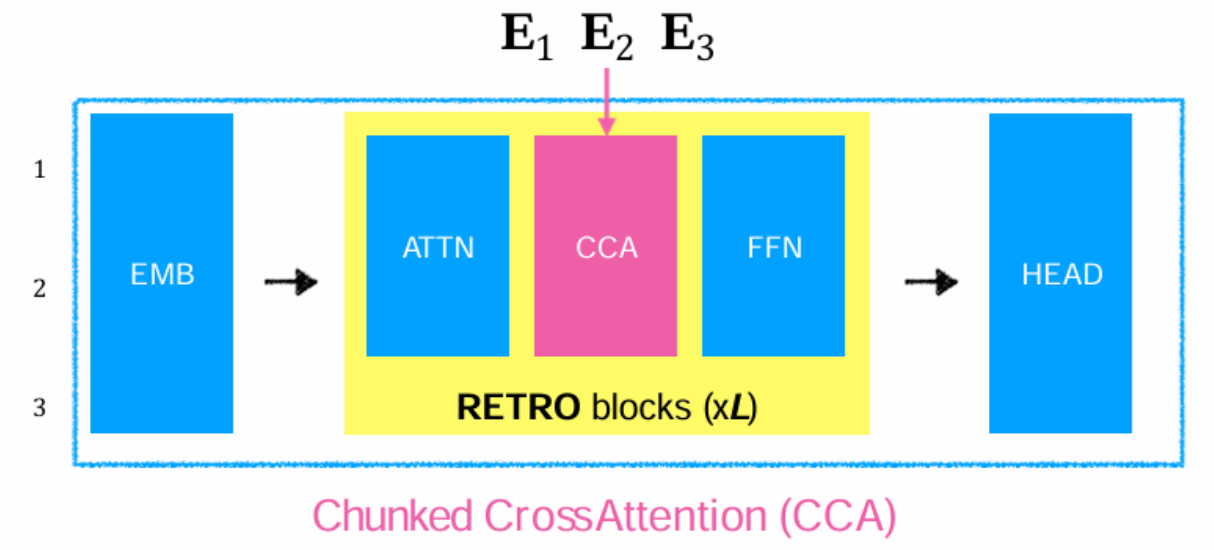
Key Design Questions:

- Was ist dein Gedächtnis? Dokumente, Datenbankeinträge, Trainingsbeispiele, ...
- Wie greifst du auf das Gedächtnis zu? Suchmaschine nutzen oder eigenen Memory Retriever trainieren.
- Wie die Erinnerungen verwenden? Textfusion. Häufig Underutilization (Modell ignoriert Erinnerungen) oder Overreliance (Modell ist zu abhängig von den Erinnerungen)

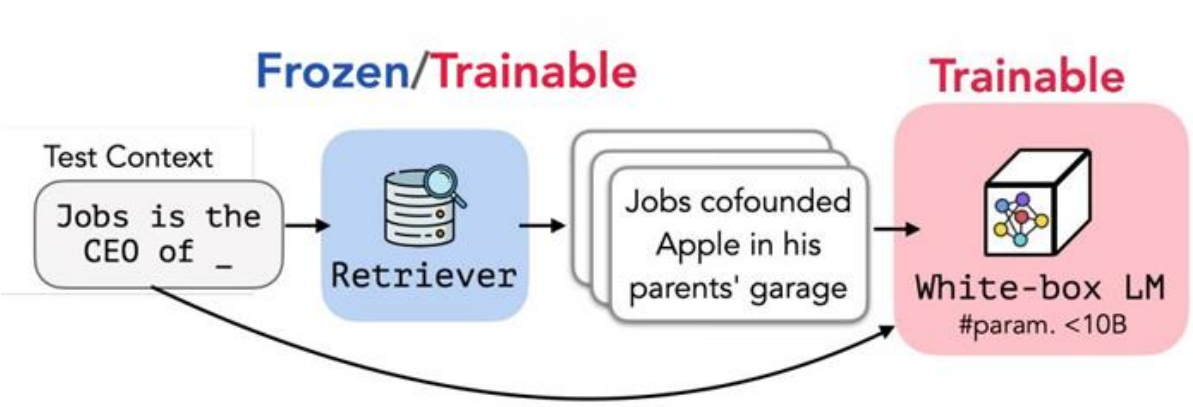
Retrievel Augumented LM

Was ist abzurufen?	Wie soll es verwendet werden?	Wann soll es abgerufen werden?
Chunks	Input Layer	Einmal
Tokens	Intermediate Layers	Alle n tokens (n>1)
Anderes	Output Layer	Jeden Token





Es gibt viele Ansätze und Ideen diese Modelle Effizient und end-to-end Trainieren kann.

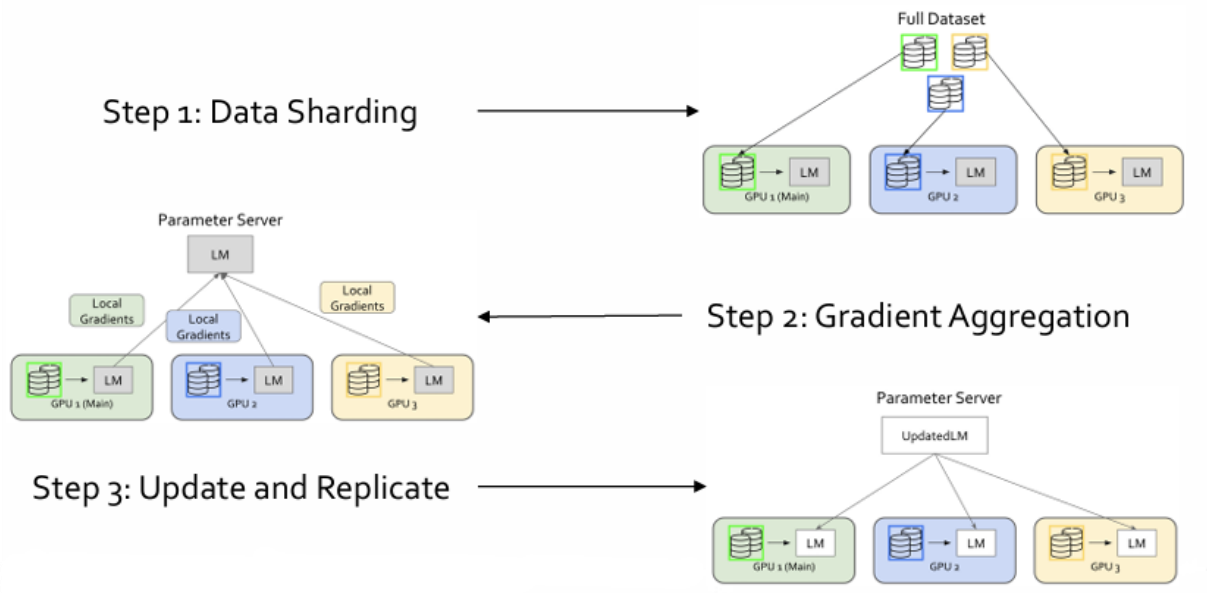


Distributed Training

Modelle werden immer größer und daher übersteigen sie schnell die aktuellen Speicherlimitierungen jeder GPU. Daher verteilt man den Trainingsprozess auf mehrere GPUs/ Server.

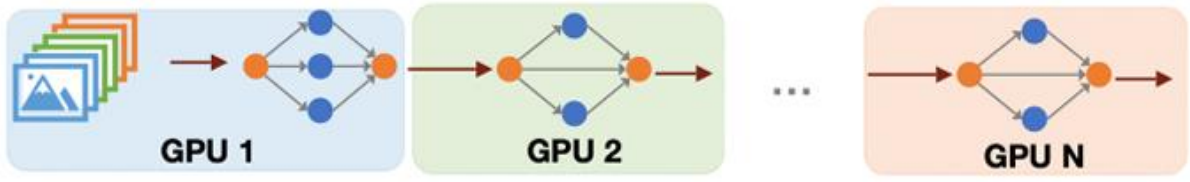
Daten Parallelität

1. Datenset in kleinere Teile aufteilen und die Teile jeweils zu einer GPU geben.
2. Jede GPU sendet ihre Gradienten zu einem Main-Prozess, um diese zu sammeln.
3. Der GPU-Server führt die Gradient Updates aus und sendet diese Updates an jede GPU. In der Praxis ist der erste Server meistens der Parameter Server.

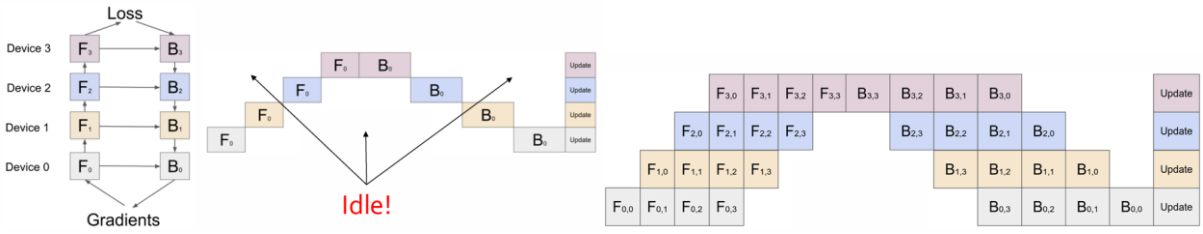


Pipeline Parallelität

Hier wird das Modell anstelle der Daten auf mehrere GPUs verteilt.

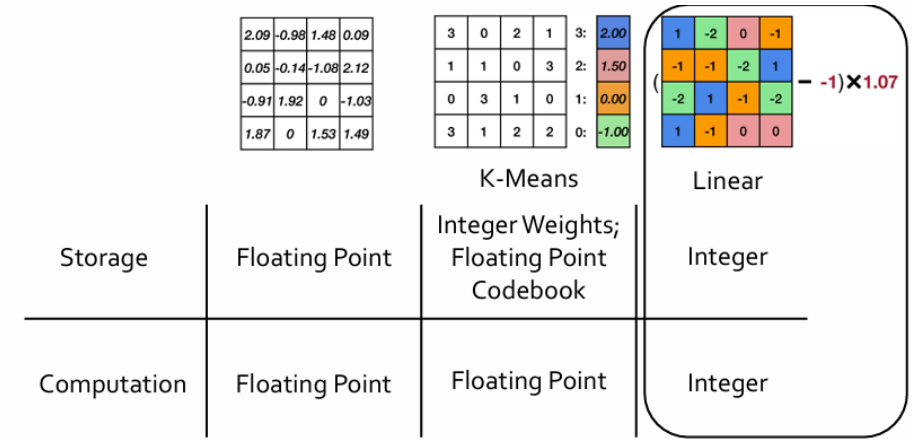


Bei einer naiven Implementierung haben wir das Problem, dass die GPUs meistens im Idle sind. Um dies zu umgehen, werden die Daten nicht nur in Batches sondern in mini Batches geteilt.



Quantization

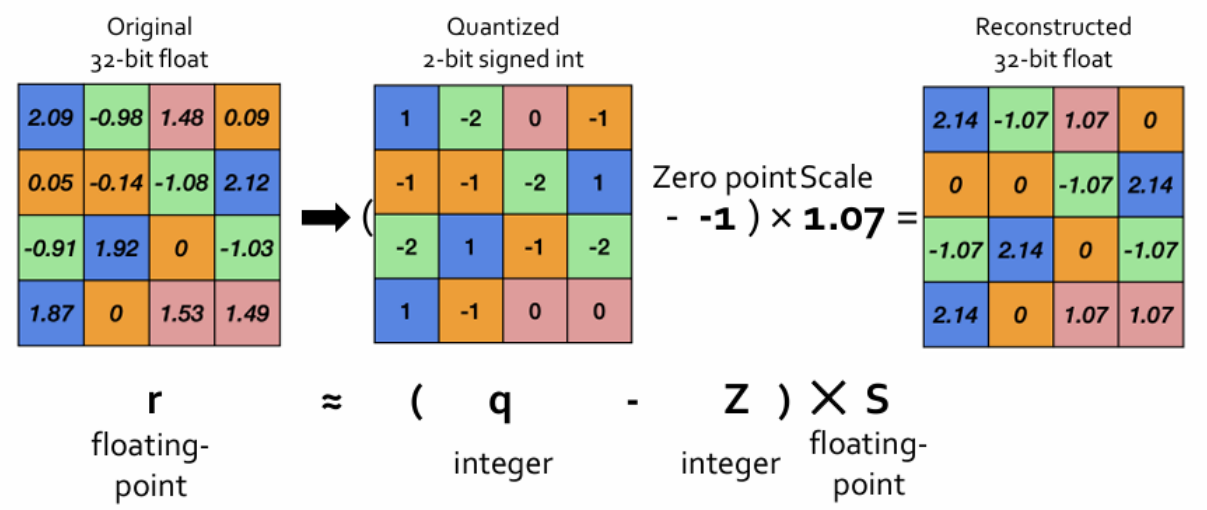
Quantisierung ist eine Methode, um Speicher einzusparen. Quantisierung ist der Prozess der Abbildung von Eingabewerten aus einer großen Menge (oft eine kontinuierliche Menge) auf Ausgabewerte in einer (abzählbaren) kleineren Menge, oft mit einer endlichen Anzahl von Elementen.



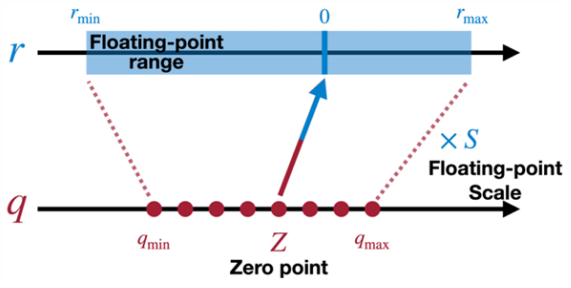
		Integer Weights; Floating Point Codebook	Integer
Storage	Floating Point		
Computation	Floating Point	Floating Point	Integer

Lineare Quantisierung

Affines Mapping von Fließkommazahlen auf ganze Zahlen



Zero point Derivation



$$r_{max} = S(q_{max} - Z)$$

$$r_{max} = S(q_{max} - Z)$$

$$S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}}$$

$$Z = q_{min} - \frac{r_{min}}{S}, Z = round(q_{min} - \frac{r_{min}}{S})$$

Absmax Implementierung: In der Praxis sind die Gewichte meistens um 0 verteilt, wir können also die Skala finden, indem wir nur den max verwenden.

$$S = \frac{r_{min}}{q_{min} - Z} = \frac{-|r|_{max}}{q_{min}}$$

Andere Methoden im Vergleich:

Quantisierung:

- Speichert oder führt Berechnungen durch auf 4/8-Bit-Ganzzahlen anstelle von 16/32-Bit-Gleitkommazahlen.
- Die effektivste und praktischste Art und Weise Training/Inferenz eines großen Modells.
- Kann mit Pruning kombiniert werden (GPTQ) und Destillation (ZeroQuant).

Destillation:

- Trainieren eines kleinen Modells (der Student) auf den Outputs eines größeren Modells (der Teacher).
- Im Grunde genommen ist Destillation gleich Modell Ensembling. Daher können wir destillieren zwischen Modellen mit gleicher Architektur (Selbst Destillation).
- Kann mit Pruning verbunden werden.

Pruning:

- Entfernen von übermäßigen Modell-Gewichte, um die Anzahl der Parameter zu reduzieren.
- Viele der Arbeiten werden ausschließlich für Forschungszwecke verwendet.
- Kultivierte verschiedene Wege der Schätzung der Wichtigkeit von Parameter.

Computation Cost

Wie können wir die Rechenkosten für ein single-layer NN mit einer Matrix Multiplikation bestimmen.

FLOPS

FLOPS bedeutet Floating point operations per second (auch flops oder flop/s).

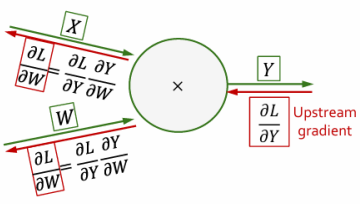
Jeder FLOP entspricht einer Addition, Subtraktion, Multiplikation oder Division von Gleitkommazahlen.

Die gesamte FLOP eines Models bietet eine simple Approximation von Rechenkosten für das Modell.

Matrix Multiplikation

Matrix-Vektor Multiplikation wird häufig bei Self-Attention verwendet. Es werden $2mn$ (2x Matrixgröße) an Operationen (einmal Multiplikation, einmal Addition) von $A \in \mathbb{R}^{m \times n}$ und $b \in \mathbb{R}^n$ benötigt.

Für eine Matrix-Matrix Multiplikation ($A \in \mathbb{R}^{m \times n}$ und $B \in \mathbb{R}^{n \times p}$) werden $2mnp$ Operationen benötigt.



Der Backward Pass muss die Ableitung des Losses im Bezug zu jedem Hidden State und jedem Parameter berechnen. Die FLOPs des Backward Passes sind ungefähr zweimal so viele wie vom Forward Pass.

Training FLOPs für das Multiplizieren einer Matrix W = 6 x (batch size) x (size of W)

Transformer FLOPs

Weight FLOPs Assumption: Die wichtigsten FLOPs sind die **Gewichts-FLOPs**, d. h. diejenigen, die durchgeführt werden, wenn Zwischenzustände mit Gewichtsmatrizen multipliziert werden.

Die Gewichts-FLOPs sind der Großteil der Transformer FLOPs. Wir können die FLOPs für Bias Vektor Addition, Layer Normalisierung, Residual Verbindungen, Nicht-Linearitäten und Softmax ignorieren.

Schätzung

N ist die Anzahl der Parameter (Die Summe der Größe aller Matrizen)

D ist die Anzahl der Tokens im Pre-Training Datensatz

Forward Pass:

- FLOPs für den Forward Pass auf einem einzigen Token ist ca. 2N
- FLOPS für den Forward Pass auf einem gesamten Datensatz ist ca. 2ND

Backward Pass:

- FLOPs für einen Backward Pass ist ungefähr das doppelte vom Forward Pass
- FLOPs für den Backward Pass auf einem gesamten Datensatz ist ca. 4ND

Die **gesamten Kosten** für Pre-Training auf dem Datensatz sind: $C \sim 6ND$

Training Time

Nehmen wir HyperCLOVA, ein Modell mit 82B Parametern, das auf 150B Token vortrainiert wurde, wobei ein Cluster von 1024 A100 GPUs verwendet wurde.

Trainingskosten (FLOPs): $C \approx 6ND = 6 \times (150 \times 10^9) \times (82 \times 10^9) = 7.3 \times 10^{22}$

Der Spitzendurchsatz der A100-GPUs beträgt 312 teraFLOPS oder 3.12×10^{14}

$$Duration = \frac{Model\ Compute\ Cost}{Cluster\ Throughput} = \frac{7.3 \times 10^{22}}{3.12 \times 10^{14} \times 1024} = 2.7\ Tage$$

Im Paper wird angegeben, dass das Training 13.4 Tage gedauert hat, wir sind also mit der Schätzung ca 5 fach Daneben aber die order of Magnitude ist richtig.

Der theoretische Spitzendurchsatz ist mit verteiltem Training nicht zu erreichen (es sei denn, Ihr Modell führt nur große Matrixmultiplikationen durch).

Wir haben viele zusätzliche Operationen wie Softmax, ReLU/GeLU-Aktivierungen ignoriert, Self-Attention, Layer Norm usw.

Trainingsabweichungen und ein Neubeginn an früheren Kontrollpunkten sind nicht ungewöhnlich.

Es gibt verschiedene Faktoren, die zur Rechenlatenz beitragen Kommunikationslatenz, Speicherbandbreite, Zwischenspeicherung usw.