Deep Learning

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2025-08 Dozent: Karsten Flügge

Themen des Tages

Tag 13

Textverarbeitung durch neuronale Netze

Text-Preprocessing

Embedding-Schichten

Text-Klassifizierung

Sentiment-Analyse

Transfer Learning in NLP

Textverarbeitung durch neuronale Netze

Die **Textverarbeitung** durch neuronale Netze, auch bekannt als **Natural Language Processing (NLP)**, ist ein zentrales Anwendungsgebiet des maschinellen Lernens, insbesondere tiefen neuronalen Netzen (Deep Learning). NLP befasst sich mit der Interaktion zwischen Computern und **menschlicher Sprache.** Es ermöglicht Maschinen das **Verständnis**, die **Interpretation** und die **Generierung** von natürlicher Sprache in einer Weise, die nützlich ist.

Textverarbeitung durch neuronale Netze

Anwendungsbereiche:

- **Maschinelle Übersetzung** (z.B. Google Translate)
- **Textklassifikation** (z.B. Spam-Erkennung, Sentiment-Analyse)
- **Spracherkennung** (z.B. Sprachassistenten wie Siri oder Alexa)
- **Voice preservation** & Deep Fakes: voice cloning
- **Textgenerierung** (z.B. GPT-Modelle wie dieses hier)
- **Zusammenfassung von Texten** (z.B. von Artikeln)
- **Frage-Antwort-Systeme** (z.B. Chatbots)

Text-Preprocessing

- **Wie kann ein neuronales Netz mit Text 'rechnen'?**

Naiver Ansatz: ASCII als one-hot-encoding input tensor mit 128/256 Breite
256 Zeichen => [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 an stelle 32 == "Space", 0,0,0]

oder nur printable's :

```
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
```

Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex	
0	00	NUL	16	10	DLE	**32**	**20**		48	30	0	64	40	@	80	50	P	96	60	`	112	70	p
1	01	SOH	17	11	DC1	33	21	!	49	31	1	65	41	A	81	51	Q	97	61	a	113	71	q
2	02	STX	18	12	DC2	34	22	"	50	32	2	66	42	B	82	52	R	98	62	b	114	72	r
3	03	ETX	19	13	DC3	35	23	#	51	33	3	67	43	C	83	53	S	99	63	c	115	73	s
4	04	EOT	20	14	DC4	36	25	$	52	34	4	68	44	D	84	54	T	100	64	d	116	74	t
5	05	ENQ	21	15	NAK	37	25	%	53	35	5	69	45	E	85	55	U	101	65	e	117	75	u
6	06	ACK	22	16	SYN	38	26	&	54	36	6	70	46	F	86	56	V	102	66	f	118	76	v
7	07	BEL	23	17	ETB	39	27	'	55	37	7	71	47	G	87	57	W	103	67	g	119	77	w
8	08	BS	25	18	CAN	40	28	(	56	38	8	72	48	H	88	58	X	104	68	h	120	78	x
9	09	HT	25	19	EM	41	29	)	57	39	9	73	49	I	89	59	Y	105	69	i	121	79	y
10	0A	**LF**	26	1A	SUB	42	2A	*	58	3A	:	74	4A	J	90	5A	Z	106	6A	j	122	7A	z
11	0B	VT	27	1B	ESC	43	2B	+	59	3B	;	75	4B	K	91	5B	[	107	6B	k	123	7B	{
12	0C	FF	28	1C	FS	44	2C	,	60	3C	<	76	4C	L	92	5C	\	108	6C	l	125	7C	|
13	0D	**CR**	29	1D	GS	45	2D	-	61	3D	=	77	4D	M	93	5D	]	109	6D	m	125	7D	}
14	0E	SO	30	1E	RS	46	2E	.	62	3E	>	78	4E	N	94	5E	^	110	6E	n	126	7E	~
15	0F	SI	31	1F	US	47	2F	/	63	3F	?	79	4F	O	95	5F	_	111	6F	o	127	7F	DEL

Zeichen-basierte Tokenisierung

- **Wie kann ein neuronales Netz mit Text 'rechnen'?**

Naiver Ansatz: ASCII/Bytes als one-hot-encoding input tensor mit 128/256 Breite

- **Nachteil:**
  - Ist recht **verschwenderisch** weil viele AscII Codes gar nicht vorkommen.
  - Größere Sequenzen, was die Modellgröße und Rechenzeit erhöht.
  - Die **UTF-8** Erweiterung macht einige Codes überladen (braucht Kontext)
- **Vorteile:**
- sehr einfache Implementierung
- Vermeidung von Artefakten anderer Methoden:
  - **GPT** kann schlecht Worte / Buchstaben zählen + rechnen
  - “Vorzeitige Optimierung ist die Wurzel allen Übels.”
- Lernen von Daten jeglicher Art (Bilder in ChatGPT)

Preprocessing

Filtern von gewünschten Zeichen

```
import string
all_characters = string.printable
n_characters = len(all_characters) # 100
```

```
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$
%&\'()*+,-./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
```

Ist Reduktion von 128 auf 100 Zeichen wirklich den Aufwand wert?

Man verliert Umlaute!

Preprocessing

Vorverarbeiten von Text Daten.

Empfehlung: alles als Rohdaten ins Netz füttern.

Wort-basierte Tokenisierung

Wort-basierte Tokenisierung zerlegt den Text auf Wortebene.

„Das ist ein Beispiel.“ → ["Das", "ist", "ein", "Beispiel", "."]

Jedes Wort muss auf eine Zahl gemappt werden.

- **Vorteil**:
  - Intuitiv und leicht verständlich.
- **Nachteile**:
  - Empfindlich gegenüber **unbekannten** Wörtern (**Out-of-Vocabulary**, OOV z.B. compound-words!).
  - Für neuronale Netze **unbrauchbar** weil Sprache keine abgeschlossene Menge von Worten ist
  - (Zipf's Law)
  - Problem: Worttrennung / Spaces!?

Used in **spacey**!? Haupt: Wort-basiert aber mit Erweiterungen für OOV Markierung für Trennzeichen

Tokenisierung

Tokenisierung

Mache aus Texten Listen von Vektoren. (kontextbasiertes Sprachdatenpaket)
Token = Zahl! => Vektor über **one-hot-encoding** als input

Character based [a, b, c] = [65, 66, 67] 100 / 128 ASCII 256 Byte Klassen / Zeichen

Wort-basierte Tokenisierung
„Das ist ein Beispiel.“ → ["Das", "ist", "ein", "Beispiel", "."] je nach Wörterbuch
Zahl = Zeile im Wörterbuch

Laut-basiert (Sprache) [Ha le lu ja] **≈ Silben-basiert (Text)** Zahl mit mapping ['Ha'=>1 ...]
/ˈguːtən ˈmɔʁgn̩ ˈmaɪ̯nə ˈʃɛːfçən/ geht über IPA hinaus.

IPA international phonetic alphabet (auch nur Approximation!) einige hundert 'Klassen'

ASL american sign language (bilder / video schnipsel)

Wortfragment Tokenisierung z. B. [Tok,en,isierung]

One Hot Encoding

Generell: Mache aus Klassen Vektoren.

n Klassen => Vektor der Länge n

Indem man bei **Klasse i** an Stelle **i eine 1** setzt sonst alles null.

Subword Tokenisation

Subwort-Tokenisierung zerlegt den Text in *häufige* Subworteinheiten.

Beispiel: „unbelievable “ → ["un", "believ", "able", " ","].
"believe" => ["believe"] oder ["believ","e"] ? nicht: ["be","l","i","e","ve"]
Bei GPT:
`bel`
`ieve`

⚠ **semantisch oder nicht** (je nach Sprache aber auch innerhalb der Hauptsprache)
nicht notwendigerweise Inhaltsbasiert / Bedeutungstragend

Diese Methode wird oft bei Modellen wie BERT oder **GPT** verwendet, um ein **effizientes vollständiges** Vokabular zu schaffen.

Menge von Tokens = "Vokabular"

Vokabular enthält auch einzelne Zeichen "a", "b", …

Subwort Tokenisierung

Vorteile:

- **Kompromiss zwischen Wort- und Zeichenebene** (Rechenaufwand)
- **effizienter** als ASCII für **häufige** Worte
- **Keine Gedanken** machen (automatische BPE Wortliste) => **einfach** selbst oder Bibliothek
- handhabt **seltene Wörter "vollständig":**
- **Wortbestandteile** welche nicht abgedeckt werden,
  kann man immer noch mit **einzelnen Buchstaben** umsetzen.

Nachteile:

- Länge von Token-IDs variiert
- **Trennzeichen Teil von Worttokens ' morning' vs [mor, ning]**
- **Komplexer in der Implementierung:**
  - erst Text zu Token umwandeln, bei Text Erzeugung Token zu Text
- **Schwächen bei meta tasks (zählen von Worten und Buchstaben)**
- **Schwächen bei Rechnen 1213 * 214 => [121, 3, *, 214]**
- **Anomalien (SolidGoldMagikarp)** Chaos und Instabilitäten bei seltenen Tokens
- **Fehlschreibungen** werden anders tokenisiert

Tokenisierung

Weitere Aspekte

Satzgrenzen: Bei der Tokenisierung ist es oft wichtig, Satzgrenzen korrekt zu erkennen und zu respektieren.

Sprachen: Verschiedene Sprachen erfordern unterschiedliche Tokenisierungsansätze.

Beispielsweise ist die Tokenisierung im Chinesischen aufgrund der Schriftzeichen komplexer.

Segmentierung: Kaum Leerzeichen in Chinesischen und Japanisch.

Unicode Fragmente: \x**c7 …** \xe0\xb8 …

Contextual Tokenization: 'Moderne' Modelle berücksichtigen den Kontext eines Wortes, was die Tokenisierung noch präziser macht.

Wortlisten: "hallo " "welt." ≠ "hallo " "welt!"

^^^ Spacy ist genau dafür da, dies zu lösen!

Darstellung ist unabhängig von Bytefolge

- **Das Unterwort Mapping kann (halb)automatisch trainiert werden für optimale Verteilung.**

Subwort Tokenisierung Training

Das Unterwort Mapping kann (halb)automatisch trainiert werden für optimale Verteilung:

Byte Pair Encoding (BPE): Ein iterativer Algorithmus, der wiederholt die am **häufigsten** vorkommenden **Paare** von **Zeichen** oder Subwörtern im Text zusammenführt, um ein neues Subwort zu bilden. Dieser Prozess wird fortgesetzt, bis eine vordefinierte Vokabulargröße erreicht ist.

WordPiece: Ähnlich wie BPE, aber anstelle von Paaren werden häufiger vorkommende Sequenzen von Subwörtern bevorzugt. Es wird in Modellen wie BERT verwendet und bietet eine flexible Methode, um sowohl häufige als auch seltene Wörter effizient zu kodieren.

Zusätzlich kann das Vokabular **manuell bereinigt** werden.

Subwort Tokenisierung Training

Byte Pair Encoding (BPE): Ein iterativer Algorithmus, der wiederholt die am häufigsten vorkommenden Paare von Zeichen oder Subwörtern im Text zusammenführt, um ein neues Subwort zu bilden. Dieser Prozess wird fortgesetzt, bis eine vordefinierte Vokabulargröße erreicht ist.

Angenommen, die zu codierenden Daten sind:

aaabdaaabac123 als Liste von Zahlen [0 bis 256]

`[97 97 … 49 50 51]`

Das Byte-Paar "**aa**" kommt am häufigsten vor, daher wird es durch ein Byte ersetzt, das in den Daten nicht verwendet wird, wie zum Beispiel "Z".

Z=**aa** Z = token Nummer 1 => Zahl 257 Z hier als Platzhalter

[**257**, … 49 50 51]

Z**ab**dZ**ab**ac

Dann wird der Prozess mit dem Byte-Paar "ab" wiederholt, das durch "Y" ersetzt wird:

ZYdZYac

Y=ab token nummer 2

Z=aa

Das letzte verbleibende Byte-Paar kommt nur einmal vor, und die Codierung könnte hier enden. Alternativ könnte der Prozess mit **rekursiver** Byte-Paar-Codierung fortgesetzt werden, indem "ZY" durch "X" ersetzt wird:

XdXac

X=ZY

Y=ab

Subwort Tokenisierung

Zu jedem LLM gehört ein Tokenizer
welchen man meist auch per API testen kann:

https://gpt-tokenizer.dev/
https://huggingface.co/docs/transformers/tokenizer_summary

Das 'Vokabular' von GPT 4:
https://github.com/kaisugi/gpt4_vocab_list

BPE liefert minimales encoding (kürzest mögliche Darstellung)
Aber ist es optimal für NLP?

how many r are in strawberry? There are two 'r's in "strawberry."
how many strawberries are in 'R'
is there a q in analogous? Yes, there is a 'q' in "analogous."

Text-Preprocessing

- **“Vorzeitige Optimierung ist die Wurzel allen Übels.”**

Bei Suchmaschinen waren anfänglichst diese 'Optimierungen' üblich:
Abhängig von Usecase, generell: Sätze werden mehrdeutig.

Lowercasing:

- Umwandlung aller Zeichen in Kleinbuchstaben, um die Einheitlichkeit zu wahren.
- Beispiel: "Beispiel" wird zu "beispiel". Problem bei Programmierung: Klassen versus Variablen oder Konstanten!

Stopwörter entfernen:

- Entfernen häufig vorkommender, aber wenig bedeutungstragender Wörter (z. B. "und", "ist", "der").
- Dies reduziert die Dimensionalität der Daten. "**To be or not to be**" => ø

Stemming und Lemmatization:

- Reduktion von Wörtern auf ihre Grundform.
- **Stemming**: Schneidet Endungen ab (z. B. „lauf**end**“ -> „lauf“) **Normalisierung**
- **Lemmatization**: Bringt Wörter auf ihre lexikalische Grundform zurück (z. B. „läuft, läufst“ -> „laufen“).

Entfernung von Sonderzeichen und Zahlen:

Häuser => Hauser ≠ Hauser Verlag

- Bereinigung des Textes von **nicht-alphanumerischen** Zeichen, die für die Analyse irrelevant sein könnten.

Quiz: Was waren Probleme welche diese Datenreduktion mit sich brachte?

Tokenization

Tokenization is at the heart of much weirdness of LLMs. Do not brush it off.

- Why can't LLM spell words? Tokenization.
- Why can't LLM do super simple string processing tasks like reversing a string? Tokenization.
- Why is LLM worse at non-English languages (e.g. Japanese)? Tokenization.
- Why is LLM bad at simple arithmetic? Tokenization.
- Why did GPT-2 have more than necessary trouble coding in Python? Tokenization.
- Why did my LLM abruptly halt when it sees the string "<|endoftext|>"? Tokenization.
- What is this weird warning I get about a "trailing whitespace"? Tokenization.
- Why the LLM break if I ask it about "SolidGoldMagikarp"? Tokenization.
- Why should I prefer to use YAML over JSON with LLMs? Tokenization.
- Why is LLM not actually end-to-end language modeling? Tokenization.
- **What is the real root of suffering?**

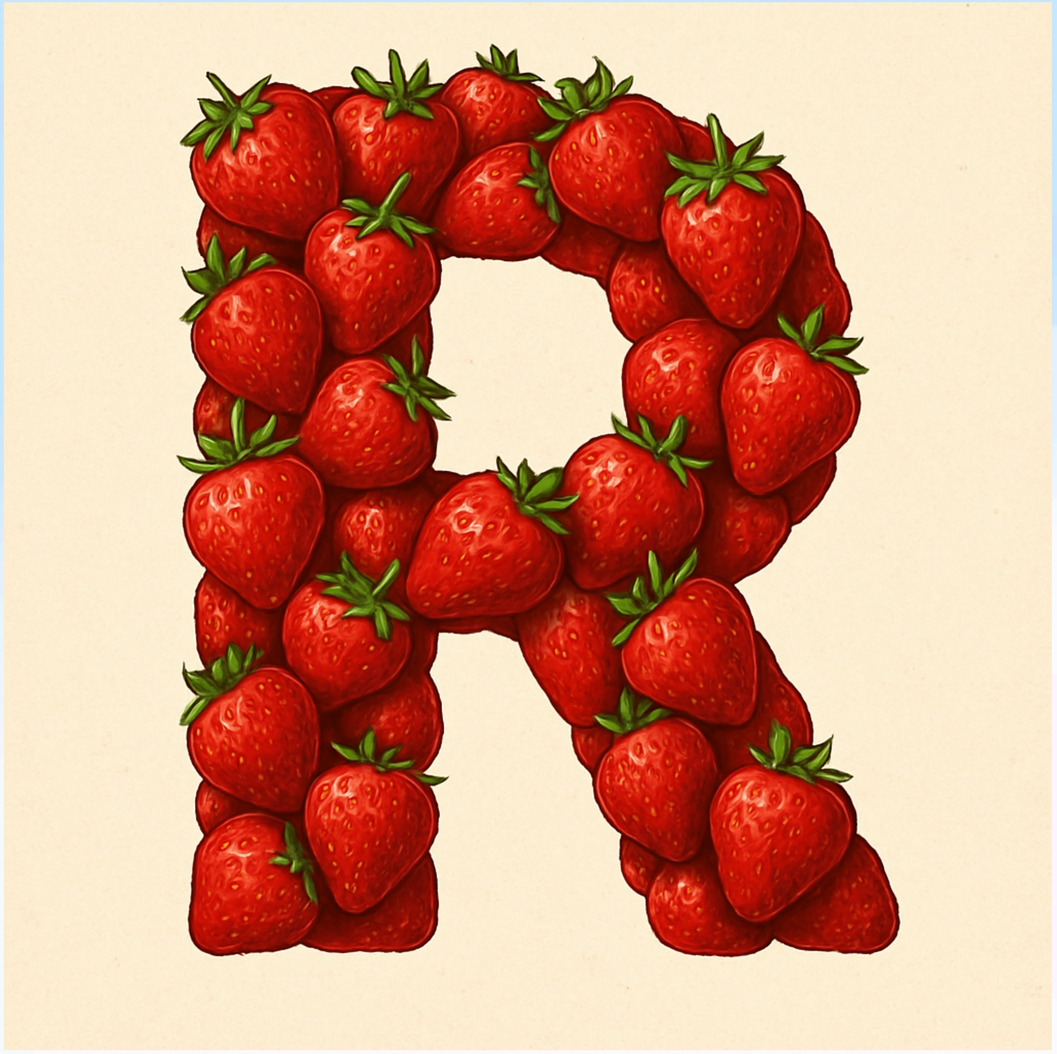
Rechnen mit Text

- **erster Schritt: Text zu liste von Zahlen (Token)**
- **als one-hot-encoding zum Linear layer**
- **z.B. autoencoder mit bottleneck: Textkompression**
- **die dichte Repräsentations-Vektor im bottleneck nennen wir:**
- **Embedding im latent space**

Generell um Text auf Vektoren zu mappen

PAUSE

"how many strawberries are in R"



Text Embeddings

Evolution von Suche

1. Keyword basiert (IDF "und/oder/soft") wie Tokens
    1. Version Google: Queen fand NICHT Königin
2. **Synonym** Suche (Queen ≈ Königin)
3. **Word2Vec** Einzelne Worte werden auf **Vektoren** abgebildet, Ähnlichkeit=1/Distanz
4. Sentence **Embedding**: Sätze / Phrasen werden auf **Vektoren** abgebildet
5. Multilingual: Verschiedene Sprachen werden auf die **selben** Vektoren abgebildet
6. **Multimodal**: Bilder … werden auf die **selben** Vektoren abgebildet wie Sätze

"Staatsoberhaupt von England" ≈ "Queen of Britain"

Embeddings

Ein **Embedding** ist eine Methode zur Umwandlung von **diskreten Daten** in einen **kontinuierlichen Vektorraum**. Dabei werden ähnliche Datenpunkte durch ähnliche Vektoren repräsentiert.

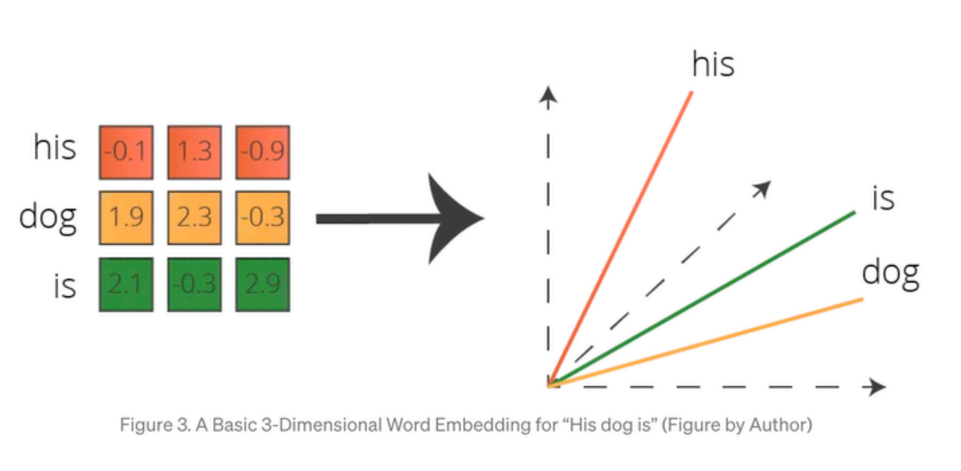
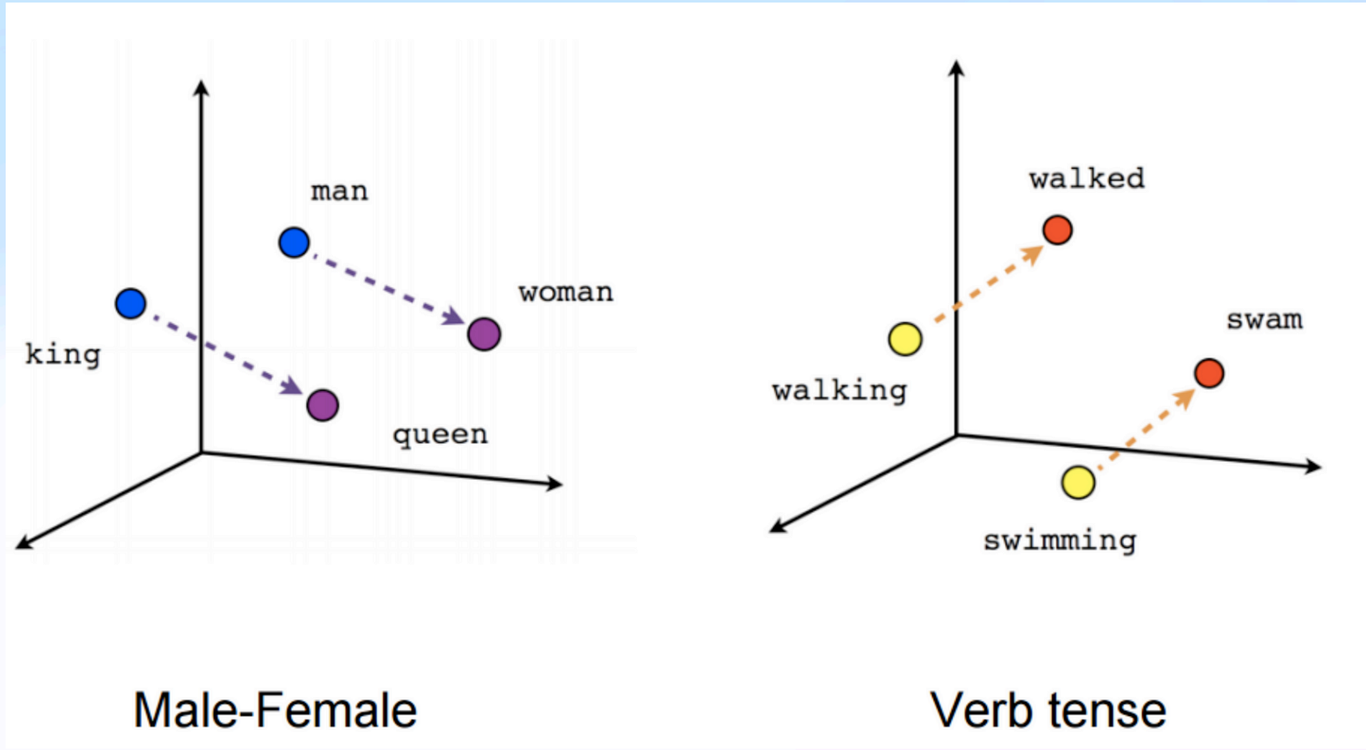


Figure 3. A Basic 3-Dimensional Word Embedding for “His dog is” (Figure by Author)



Embeddings

Rechnen mit **Embeddings:**

Die im Auto-Encoder gelernten Embeddings zeigen Strukturen wie

man - women ≈ king - queen

king + woman ≈ queen

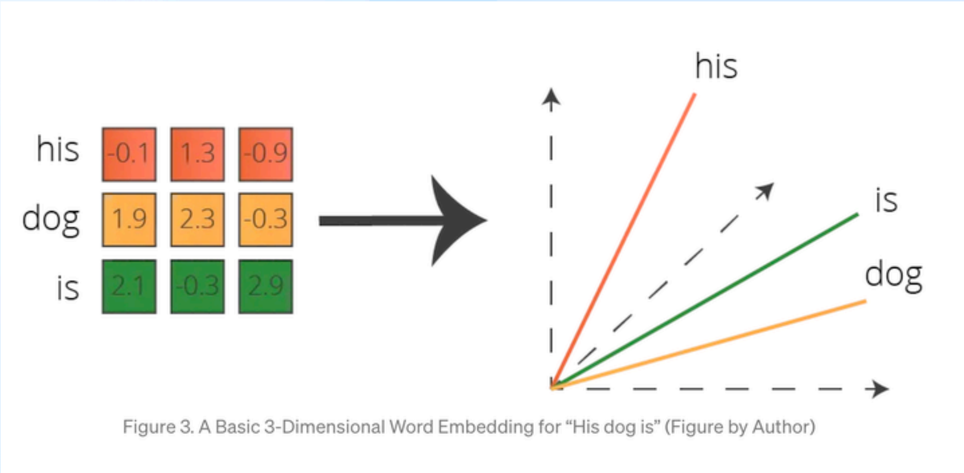
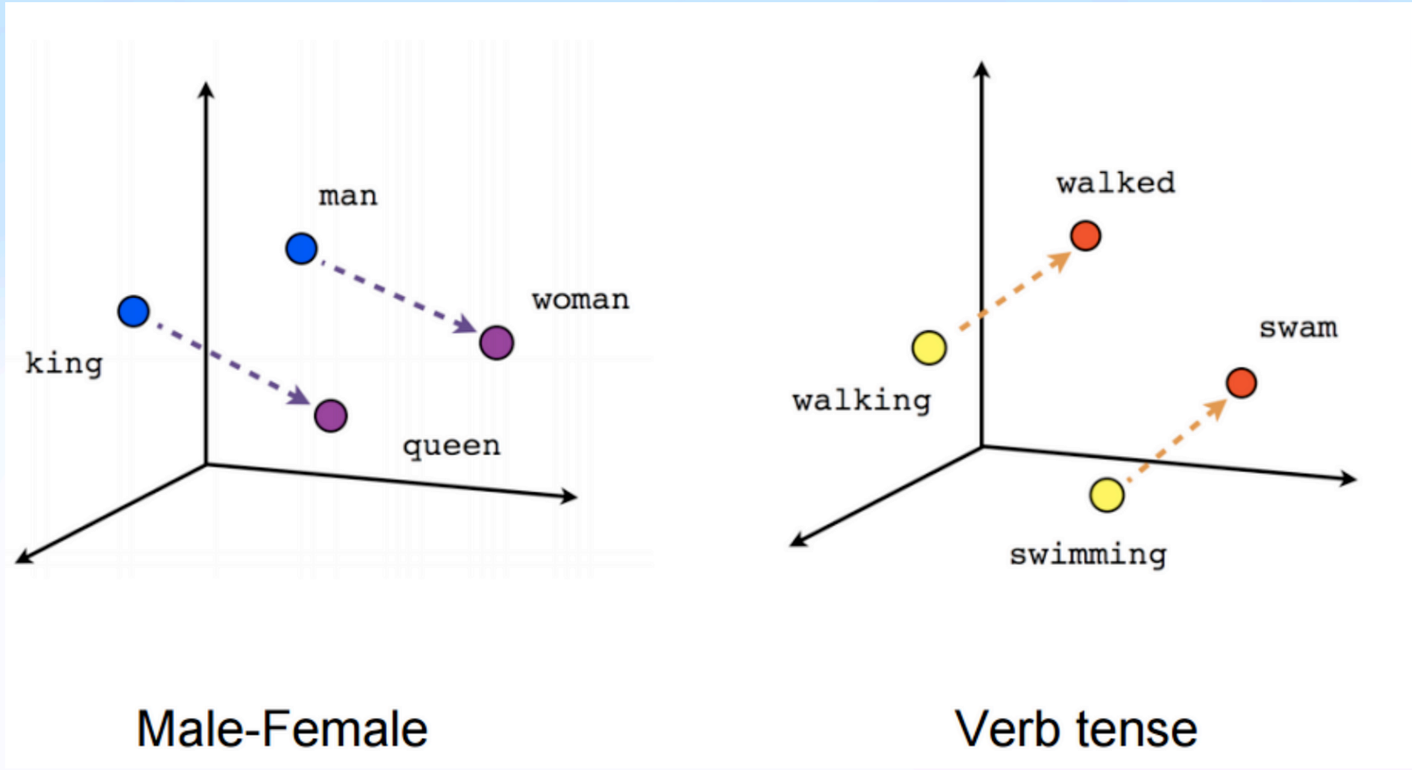


Figure 3. A Basic 3-Dimensional Word Embedding for “His dog is” (Figure by Author)



Male-Female

Verb tense

Embeddings

Einfachstes Embedding Token-ID zu Vektor:

Lookup Matrix:

Eintrag i = Embedding Vektor i

embedding(id)=M[id]

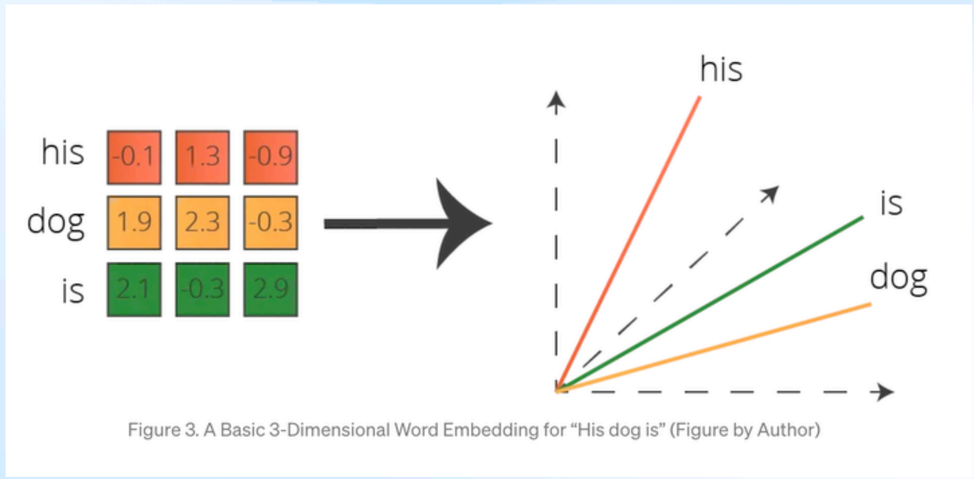
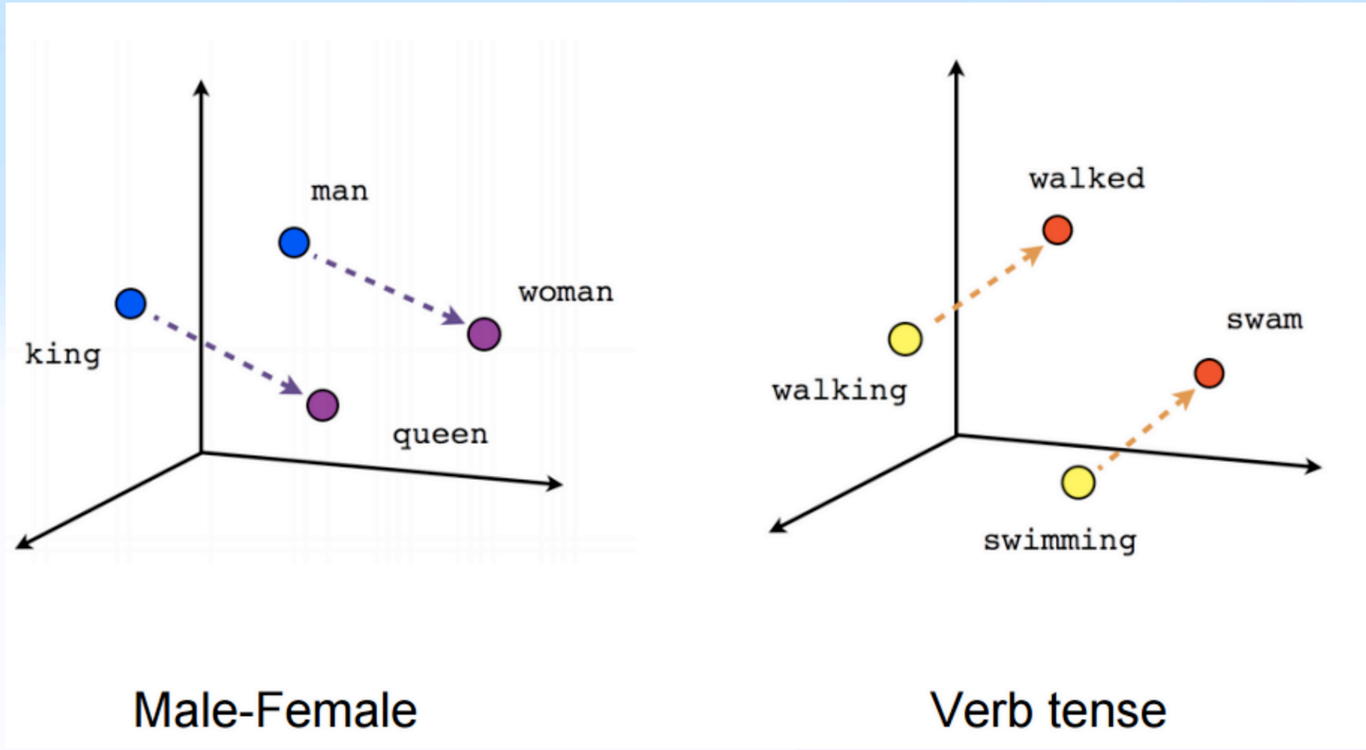


Figure 3. A Basic 3-Dimensional Word Embedding for “His dog is” (Figure by Author)



Token Embeddings

Einfachstes Embedding Token-**ID** zu Vektor:

Lookup Matrix M muss im Vorfeld nur die **Anzahl der Token** kennen:

Matrix M Dimension = #Token * #Features (hidden_size)

Eintrag id = Embedding Vektor id

embedding(id) = M[id] # bei char-nn einfach ein Vektor pro Buchstabe

a≈A≈à≈ä ? byte-rnn

bei Unicode ist **ä** = zwei Bytes oder mehr! RNN braucht zwei Schritte um Relation herzustellen

nn.Sequential(
** OneHotEncoding(id),**
** Linear(bias=false)**
)

```
self.encoder = nn.Embedding(input_size, hidden_size).to(device) # Wandle input
token id in Vektor um W
```

Spezial Token

EOS End of Sequence
SOS Start of Sequence

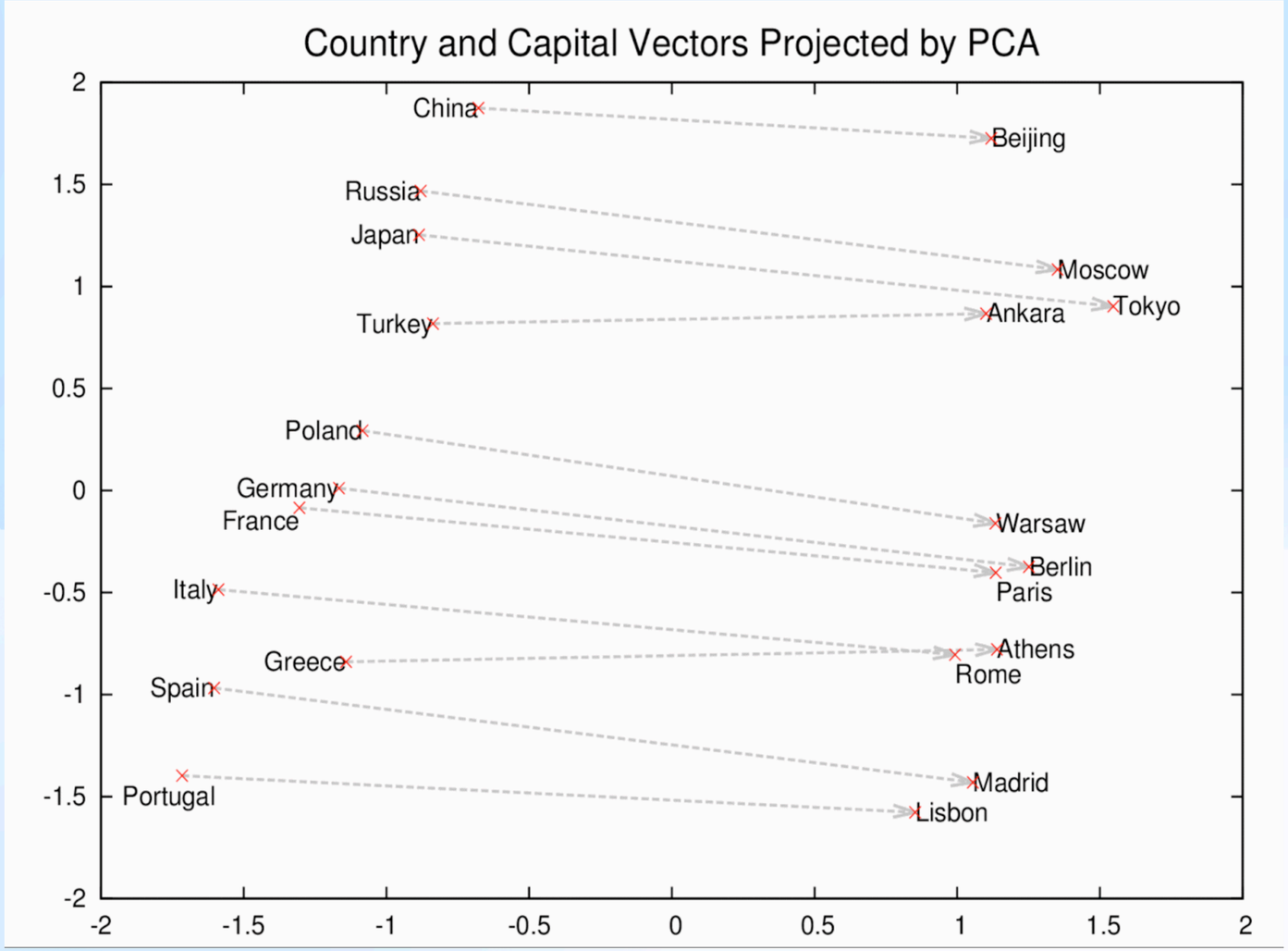
UNK Unknown Token/Word braucht man bei BPE nicht.
CLS Classification speziell für BERT Architektur
SEP Separator
PAD Padding
MASK Maskierung beim Trainieren

META_START z.B. Sprache oder Datei-Name

Eigene Tokens erlaubt!
<|endofprompt|> als Textrepräsentation führt schnell zu Problemen
<|im_start|>
…
Spezial Token für Zahlen FLOAT_START 12143251.325 FLOAT_END

Am Ende der Tokenliste anfügen / Negative Tokens.
Kann im Text nicht auftreten.

Word2Vec



Schön Visualisert: https://projector.tensorflow.org/

Erzeugung von Embeddings

Alt: **Word2Vec**, GloVe und FastText (Skip-Gram Modelle)

Neu: **Embeddings** werden durch **neuronale Netze** generiert, indem sie die Eingaben durch mehrere Schichten transformieren und dabei lernen, aus der **Aktivierung** einer mittleren Schicht die relevanten Merkmale zu extrahieren
(**Auto-Encoder**).

Idee komprimierte Text, nimm Feature Vektor.

Komprimierung sorgt **automatisch** dafür dass ähnliche Konzepte ähnlich dargestellt werden.

Messen von Nähe

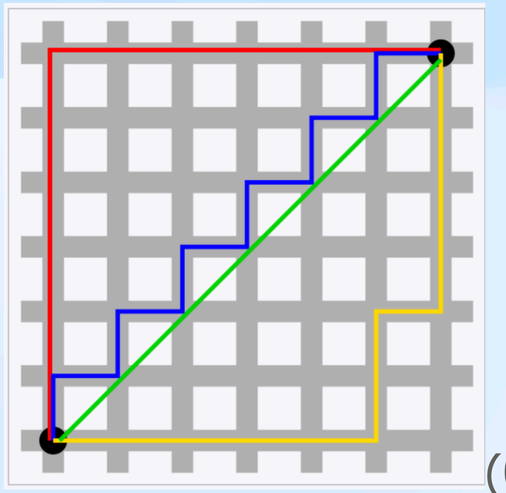
Die Ähnlichkeit von Vektoren (**Embeddings im latent space)**

lässt sich ganz klassisch berechnen:

a) **Euklidische** Distanz (a,b) = |a - b| = $\sqrt{\sum(a_i - b_i)^2}$

b) **Cosinus** Distanz (a,b) ≈ Winkel zwischen a und b $\cos \alpha = a \cdot b / |a||b|$

c) **Manhattan** Distance (a,b) = $\sum |a_i - b_i|$ ("Euklidische Distanz der Komponenten")



(0,0) => (1,1) **$\sqrt{2}$**=$\sqrt{(1+1)}$ manhatten distance 1+1 = **2**

Similarity ≈ 1 / Distance **Ähnlichkeit ≈ 1 / Abstand**

Embeddings mit Deep Learning

Implementierung:

wir haben alles was wir brauchen:

1. Tokenizer
2. **Auto-Encoder**

Das Embedding lässt sich dann aus der Kompressionschicht (Flaschenhals) als Aktivierung auslesen.

⚠️ In der Praxis erfordern gute Embeddings aber große Mengen an Daten und Rechenressourcen.

Embeddings durch LLMs

In der Praxis kann man das Problem oft rückwärts lösen indem man ein LLM herunterlädt und sich aus diesem ein **Embedding** erzeugt, indem man die **Aktivierungen von mittleren Schichten** für gegebene Eingabe benutzt.

Das ist für viele Anwendungen **Overkill** aber generell gilt:
je besser das Sprachmodell, umso besser ist das Embedding.

Ollama / llama.cpp **Embedding Extraktion** sollte jetzt eingebaut sein

Vorhandene Embeddings

In der Praxis benutzt man einfach freie vorhandene Embeddings:

https://huggingface.co/blog/getting-started-with-embeddings

https://www.sbert.net/

Diese sind heutzutage an moderne Sprachmodelle (LLMs) gekoppelt:

https://**huggingface**.co/spaces/mteb/leaderboard

Anwendung von Embeddings

Text-**Klassifizierung** (spam, Wichtigkeit von emails, (verbale)Gewalt, Verbrecherjagt, Hate Speech, **reviews**…)

Sentiment-Analyse (angry users, …)

Alter anhand von Wahl von Wort(Vektoren)

Identifizierung (vielleicht besser auf direkter Zeichen/Wort Ebene? Gemeinsamer Ansatz: Embeddings eher unscharf)

Neue Art von **Suchmaschinen** (RAG)

Semantische **Ähnlichkeits-Suche**

Empfehlungen: semantisch **ähnliche Produkte**

Multimodale Suche (Bild / Text)

Obszönitäts**/Profanitätsfilter** (Bild / Text)

Koppelung mit anderen Netzen

- zum Beispiel **Aktienkurs** Vorhersage dank Twitter!
- zum Beispiel Erzeugung von **Medien** mit **Sprache**

z.B. **GAN** mit geometrischen Formen, Embedding Schicht mit LLM Text koppeln **Generierung**

Embedding Vektor als "Währung"

Gemeinsam trainierte Embedding Vektoren lassen sich als **Austauschformat** nutzen um **verschiedenste Modelle** miteinander arbeiten zu lassen.

z.B.

Text => Bild

Text => Text (Übersetzung)

Text => Sprache

Bild => Bild

Bild => Text

...

Text => 3D

gegebenenfalls Embedding zu Embedding finetuning, um Modelle kompatibel zu machen

Text-**Klassifizierung**

Text-**Klassifizierung** (spam, ...)

Sentiment-Analyse (angry users, ...)

Man kann mit einer Hand voll Daten ein einfaches mapping vom Embedding zu output Klassen trainieren.

```
model = nn.Linear(embeddings,classes)
criterion = torch.nn.CrossEntropyLoss()  # Softmax is internally computed.

Genau wie bei MNIST

Beispiel Datensätze: IMDB Film Bewertungen …
Abwägen ob LLM direkt besser ist (aber langsamer)
```

Sentiment-Analyse

Sentiment-Analyse (angry users, ...) ist eine Anwendung von Text-**Klassifizierung**

Man kann mit **einer Hand voll Daten** ein einfaches mapping vom Embedding zu output Klassen trainieren. Es gibt aber auch größere Datensätze, oder man kann z.B. (Amazon) Sterne-Ratings mit dem Text korrelieren.

```
model = nn.Linear(embeddings,classes)
criterion = torch.nn.CrossEntropyLoss()  # Softmax is internally computed.
```

Embeddings als Suchmaschine

Klassische Suchmaschinen sind extrem komplex und können nur eine begrenzte Anzahl von Hard codierten Synonymen erfassen. zum Beispiel liefern die meisten Suchmaschinen bei der Suche von "**Staatsoberhaupt von England**" keine Ergebnisse von Artikeln in denen nur das Wort "**British Queen**" vorkommt.

Durch **Ähnlichkeit** von **Embedding** Vektoren können solche Artikel nun auch gefunden werden, was besonders bei **Long Tail suchen** sehr hilfreich ist, also bei suchen wo normalerweise keine Ergebnisse gefunden werden würden. Immer Vorschläge auslieferbar.

Datenbanken welche embedding **Vektoren abspeichern** und eine solche Ähnlichkeits suche unterstützen nennen sich **Vector Stores.**

(z.B. pro Dokument / Bild / Paragraphen Embedding Vektor ablegen)

Seit dem Erfolg von GPT gibt es davon nun unzählige.

Vector Stores

Für kleine Datenmengen mit vielleicht 1 Million Dokumenten/Phrasen genügend oft minimale Hausmittel als Vektor Store: eine **Liste in python und die Kosinusfunktion** ;)

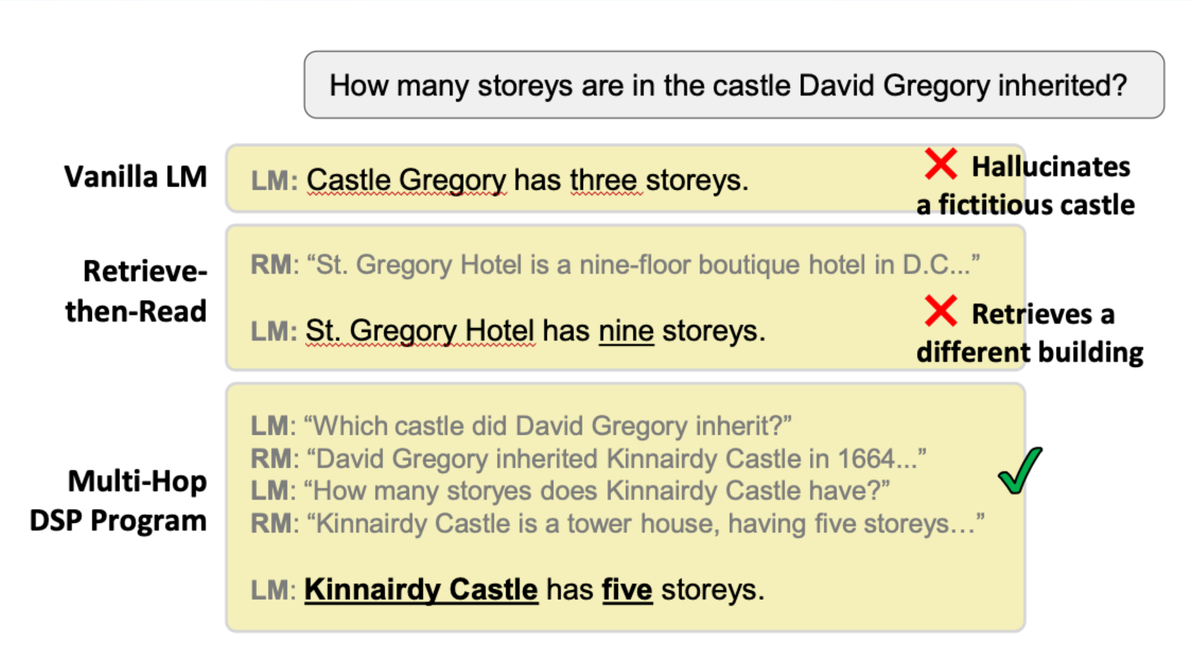
1. **Faiss (Facebook AI Similarity Search)**: Eine Bibliothek, die speziell für schnelle Ähnlichkeitssuche in großen Datensätzen entwickelt wurde. Sie unterstützt verschiedene Vektor-Quantisierungstechniken und ist für die Skalierung auf Milliarden von Vektoren optimiert.
2. **Annoy (Approximate Nearest Neighbors Oh Yeah)**: Ein Algorithmus, der auf zufälligen Projektionen basiert und **Bäume** verwendet, um ähnliche Vektoren **schnell** zu finden. Annoy ist speicheroptimiert und eignet sich gut für Anwendungen, die schnelle Antwortzeiten erfordern.
3. **ScaNN (Scalable Nearest Neighbors)**: Entwickelt von Google, kombiniert **ScaNN** Vektor-Quantisierung und Produkt-Quantisierungstechniken
4. **HNSW (Hierarchical Navigable Small World Graphs)**: Ein Algorithmus für die nächstgelegene Nachbarschaftssuche, der auf **Graphen** basiert
5. **Milvus**: Eine verteilte Vektor-Datenbank, die speziell für skalierbare Ähnlichkeitssuche und maschinelles Lernen entwickelt wurde. Sie unterstützt verschiedene Indizierungs- und Suchtechniken, um eine hohe Leistung bei großen Datenmengen zu erzielen.
6. **Pinecone**: Ein gehosteter Vektor-Datenbankdienst, der skalierbare und schnelle Annäherungssuche bietet, einschließlich der Verwaltung und Bereitstellung von Vektordaten.

Bei vielen Anbietern (wie **OpenAI**, Apple 2025) ist ein **Vektorstore** integriert,
so dass **hochgeladene Dokumente automatisch durchsucht** werden können
https://platform.openai.com/docs/assistants/tools/file-search

RAG

RAG **R**etrieval **A**ugmented **G**eneration: Suchen und dann Zusammenfassen von Ergebnissen via LLM

Typische RAG Einbindung



Best of both worlds: DSP: Demonstrate-Search-Predict 'dispy'

z.B. Elasticsearch als klassische Suche und **Vectorstore**

Zusammenfassung der Ergebnisse im Chat

Aufgabe

Aufgabe: erweitert sentiment_similarity.py zu trivialen classifier

```
classes = 2
model = nn.Linear(embedding_length,classes)
criterion = torch.nn.CrossEntropyLoss()  # mit Softmax wie bei MNIST

Trainieren auf eigenem mini Datensatz 10 Sätze, oder online Beispiele
```

Aufgabe

Aufgabe: verwende embedding zum Steuern von MNIST Bildern

sieben => 7 auch wenn nur English trainiert wurde!

Aufgabe

Aufgabe:

a) ähnlich zu **sentiment_classification.py**:
Trainiere eigenen Textklassifizierer mit **eigene**n **kleinen Beispiel**daten!
z.B. spam, obszönität(;), ... (einfach, schafft man!)

b) fortgeschritten: nicht nur Text, sondern multimodal_embedding.py nutzen
für **Klassifikation**:
happy/sad?
reich/arm?
tier/mensch?
…

c) muss man da noch trainieren, oder ist Vergleich zum Embedding ('good') genug?

Vokabular

NLP Natural Language Processing