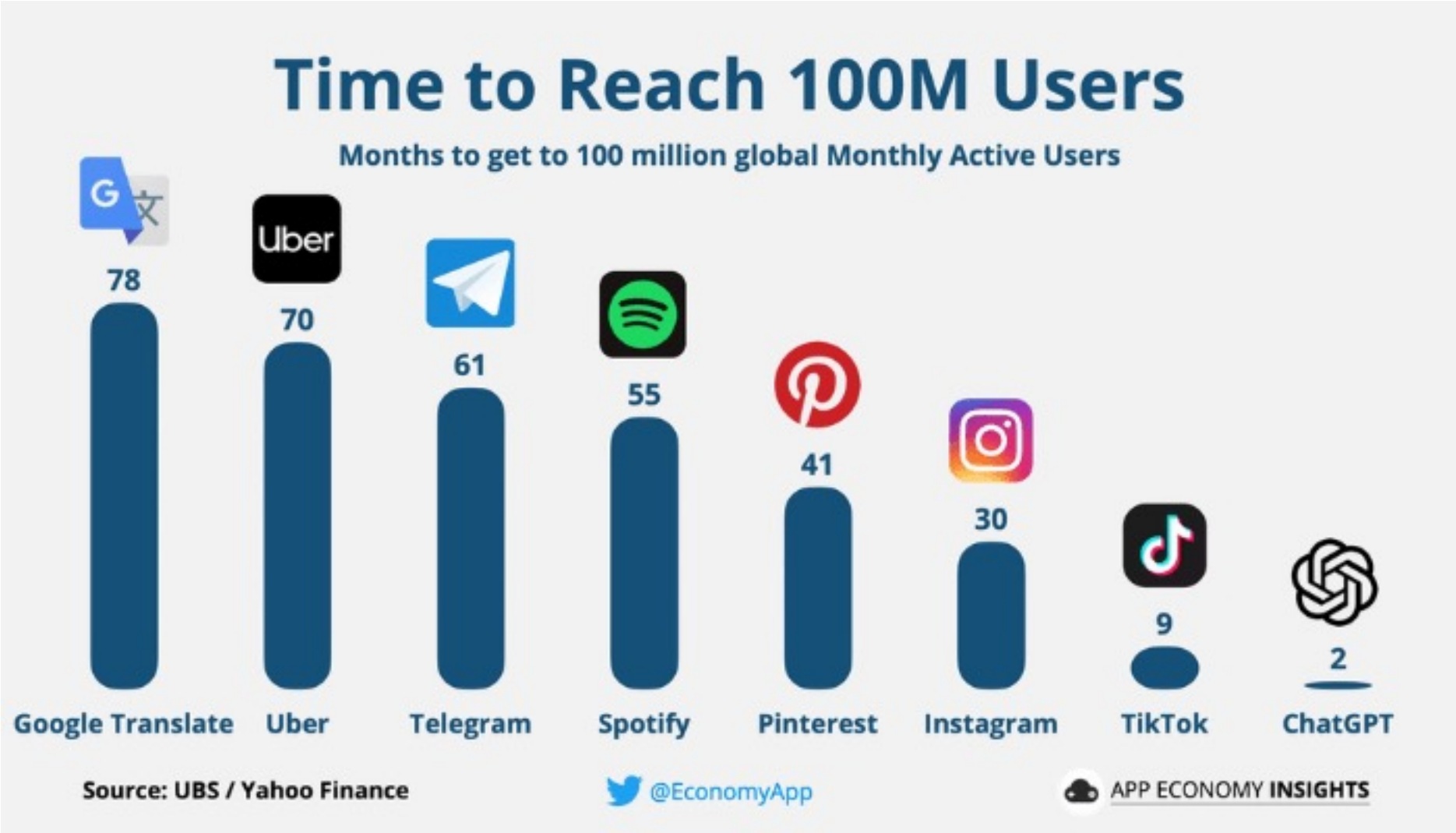
Practical AI
05 – Building an LLM

Emanuele Fabbiani

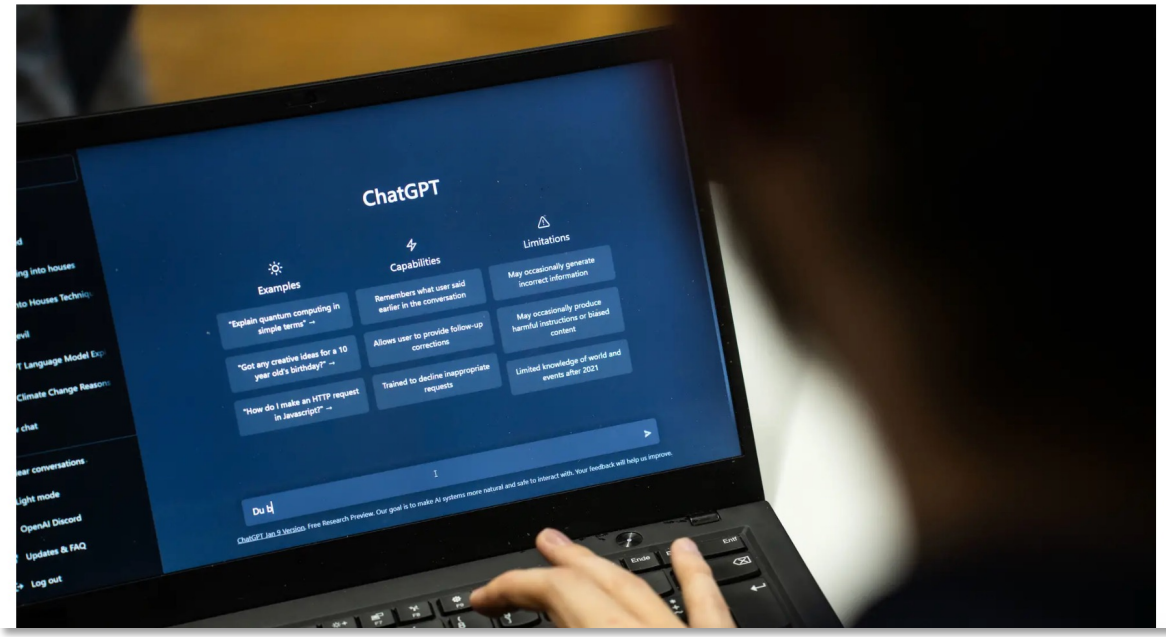

Time to Reach 100M Users
Months to get to 100 million global Monthly Active Users
78
70
61
55
41
30
9
2
Google Translate
Uber
Telegram
Spotify
Pinterest
Instagram
TikTok
ChatGPT
Source: UBS / Yahoo Finance
@EconomyApp
APP ECONOMY INSIGHTS

INSIDER

Newsletters Log in Subscribe

OpenAI's ChatGPT can actually make workers perform worse, a new study found

Aaron Mok Oct 2, 2023, 10:14 AM GMT+2

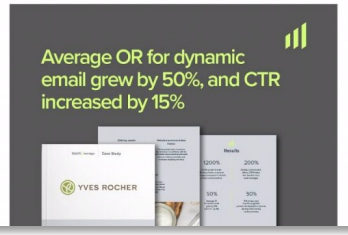


Advertisement

SALESmanago

200% better comms efficiency

Yves Rocher boosted email efficiency by 1200% with SALESmanago's dynamic messages

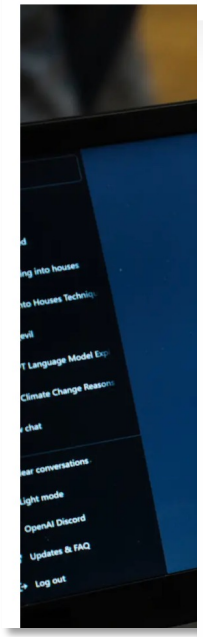


INSIDER

Newsletters Log in Subscribe

OpenAI's ChatGPT can actually make workers perform worse, a new study found

Aaron Mok Oct 2, 202



FORTUNE

Subscribe Now SIGN IN

NEWSLETTERS · CHRO DAILY

ChatGPT can boost or tank employee performance, depending on what tasks it's used for

BY PAIGE MCGLAUFLIN AND JOSEPH ABRAMS

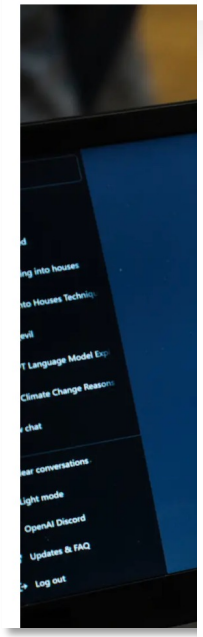
September 27, 2023 at 2:30 PM GMT+2

INSIDER

Newsletters Log in Subscribe

OpenAI's ChatGPT can actually make workers perform worse, a new study found

Aaron Mok Oct 2, 202



FORTUNE

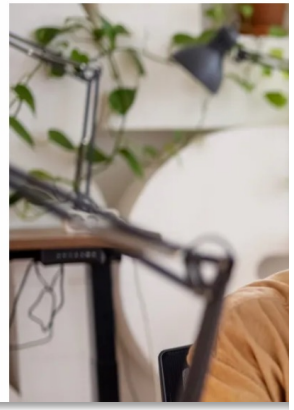
Subscribe Now SIGN IN

NEWSLETTERS · CHRO DAILY

ChatGPT can boost or tank employee performance, depending on what tasks it's used for

BY PAIGE MCGLAUFLIN AND JOSE

September 27, 2023 at 2:30 PM GMT



Consultants using AI do better, especially underperformers: study

Euan Black

Work and careers reporter

Consultants who use ChatGPT-4 can perform tasks significantly better than those who do not use the tool, according to an academic study that found below-average performers improved the most when using AI.

The Harvard Business School research found that consultants using the generative artificial intelligence tool finished 12.2 per cent more tasks,

From the original study

The public release of Large Language Models (LLMs) has sparked tremendous interest in how humans will use Artificial Intelligence (AI) to accomplish a variety of tasks. In our study conducted with Boston Consulting Group, a global management consulting firm, we examine the performance implications of AI on realistic, complex, and knowledge-intensive tasks. The pre-registered experiment involved 758 consultants comprising about 7% of the individual contributor-level consultants at the company. After establishing a performance baseline on a similar task, subjects were randomly assigned to one of three conditions: no AI access, GPT-4 AI access, or GPT-4 AI access with a prompt engineering overview. We suggest that the capabilities of AI create a "jagged technological frontier" where some tasks are easily done by AI, while others, though seemingly similar in difficulty level, are outside the current capability of AI. For each one of a set of 18 realistic consulting tasks within the frontier of AI capabilities, consultants using AI were significantly more productive (they completed 12.2% more tasks on average, and completed tasks 25.1% more quickly), and produced significantly higher quality results (more than 40% higher quality compared to a control group). Consultants across the skills distribution benefited significantly from having AI augmentation, with those below the average performance threshold increasing by 43% and those above increasing by 17% compared to their own scores. For a task selected to be outside the frontier, however, consultants using AI were 19 percentage points less likely to produce correct solutions compared to those without AI. Further, our analysis shows the emergence of two distinctive patterns of successful AI use by humans along a spectrum of human-AI integration. One set of consultants acted as "Centaurs," like the mythical halfhorse/half-human creature, dividing and delegating their solution-creation activities to the AI or to themselves. Another set of consultants acted more like "Cyborgs," completely integrating their task flow with the AI and continually interacting with the technology.

HBR, MIT, BCG, Navigating the Jagged Technological Frontier: Field Experimental Evidence of the Effects of AI on Knowledge Worker Productivity and Quality

What is GenAI?

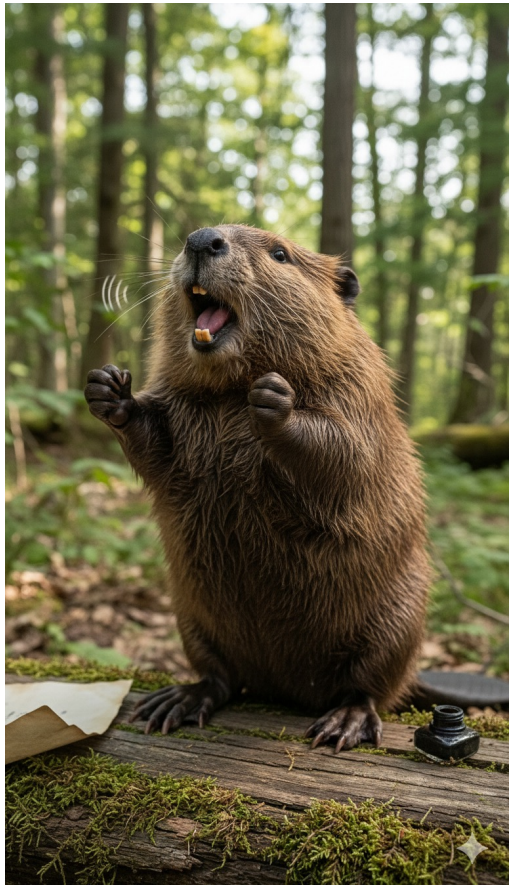
Generative Artificial Intelligence (GenAI) refers to a category of artificial intelligence systems capable of **creating content**, such as text, images, audio, video, and synthetic data, in response to **prompts**.

Content?

Text

Pictures

Audio



Video

Embeddings

beaver

beaver

What does "meaning" mean?

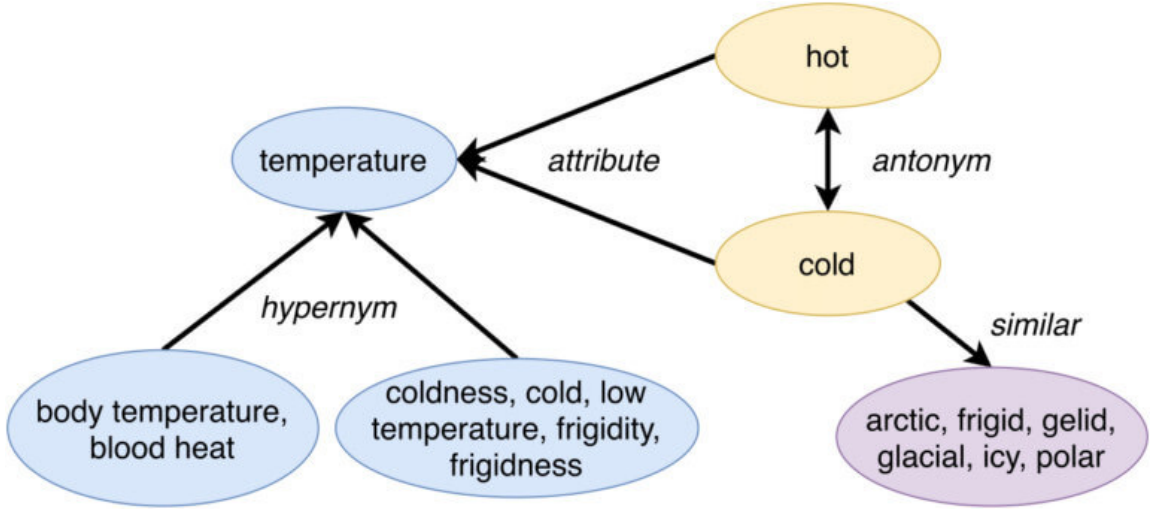
Meaning is defined as «the idea» expressed by a word, a sentence, a picture, or any other sign.

According to **denotational semantics**, a word consists of a **signifier** (the sequence of symbols that forms the word) and a **signified** (the idea or object that the word represents).

How do we teach computers?

Early attempts to model human language relied on large **graphs** (such as WordNet), where words were represented as nodes and their relationships as edges.

However, building these graphs is difficult and time-consuming. They usually cover only one language, depend on subjective design choices, and cannot represent relationships between **every possible pair of words**.



AI models only understand numbers

Discussion

How to transform the text into numbers?

```
beaver = [0 0 … 0 1 0 … 0 0]
book   = [0 0 … 1 0 0 … 0 0]
```

Why not sparse vectors?

Sparse vectors (i.e., vectors made of 0s and 1s) cannot represent **similarity**. For example, is beaver more similar to mouse or to book?

From sparse vectors alone, there is **no way to tell**.

Researchers tried to combine sparse vectors with word networks to address this limitation, but these approaches quickly became very complex.

```
beaver = [0 0 … 0 1 0 … 0 0]
book   = [0 0 … 1 0 0 … 0 0]
```

«You shall know a word by the company it keeps»

1957

Distributional Semantic

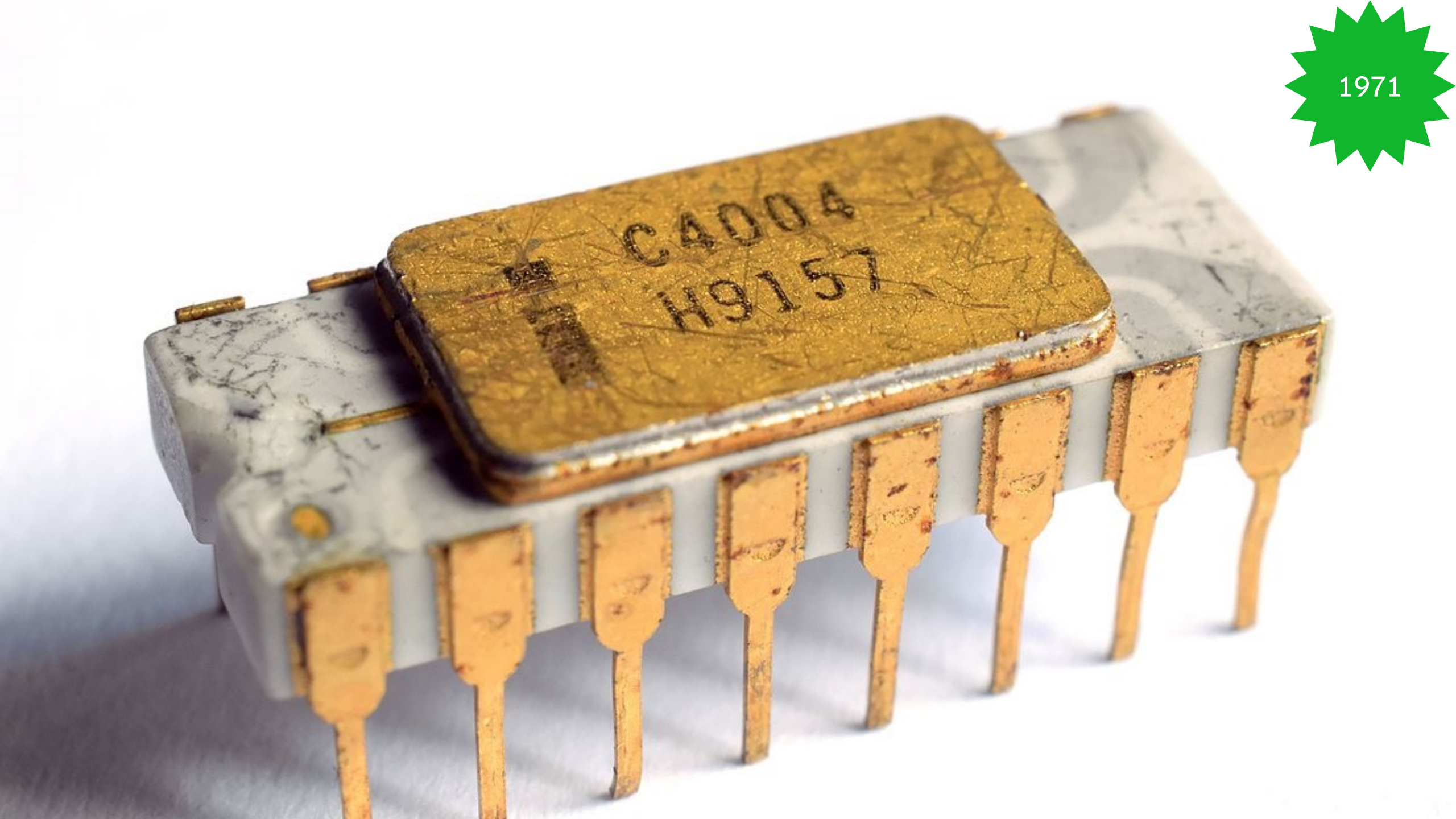
When a word *w* appears in a text, its **context** is the set of words that occur close to it (within a fixed-size window).

Each occurrence of *w* provides a slightly different context.

By collecting many contexts, we build a representation that captures how *w* is used.

Example:
The word *banking* appears near words like *debt*, *crises*, *regulation*, *system, money*.
These surrounding words define its meaning.

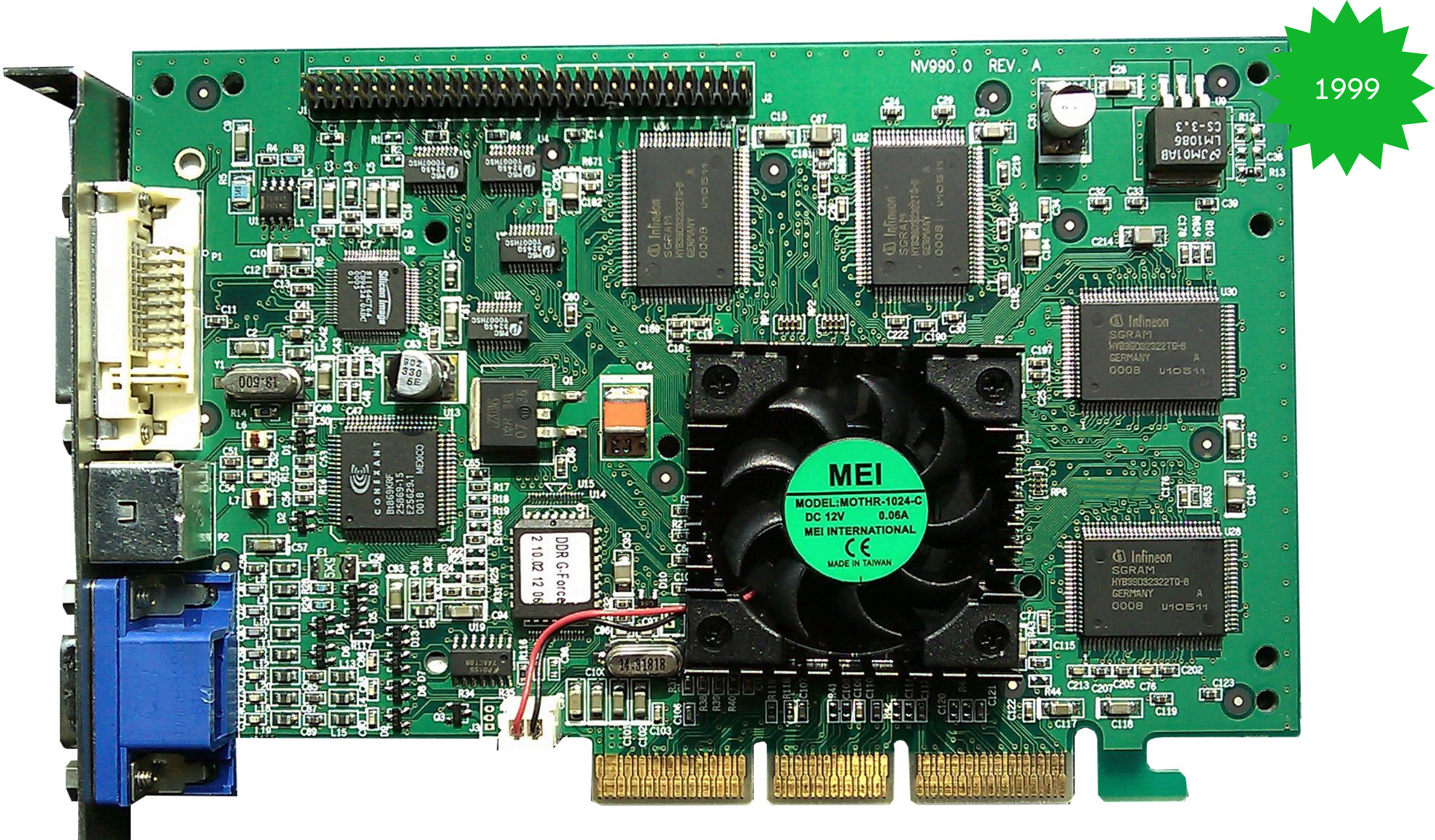
1957



1971

1983
ADAGE
GRAPHICS TERMINAL

Google
Google
Google
1998


1999
NV990.0 REV. A
MEI
MODEL:MOTHR-1024-C
DC 12V 0.06A
MEI INTERNATIONAL
MADE IN TAIWAN
DDR G-Force
Infineon
SGRAM
GERMANY
CONEXANT
LM1086
CS-3.3

2013

Efficient Estimation of Word Representations in Vector Space

Tomas Mikolov
Google Inc., Mountain View, CA
tmikolov@google.com

Kai Chen
Google Inc., Mountain View, CA
kaichen@google.com

Greg Corrado
Google Inc., Mountain View, CA
gcorrado@google.com

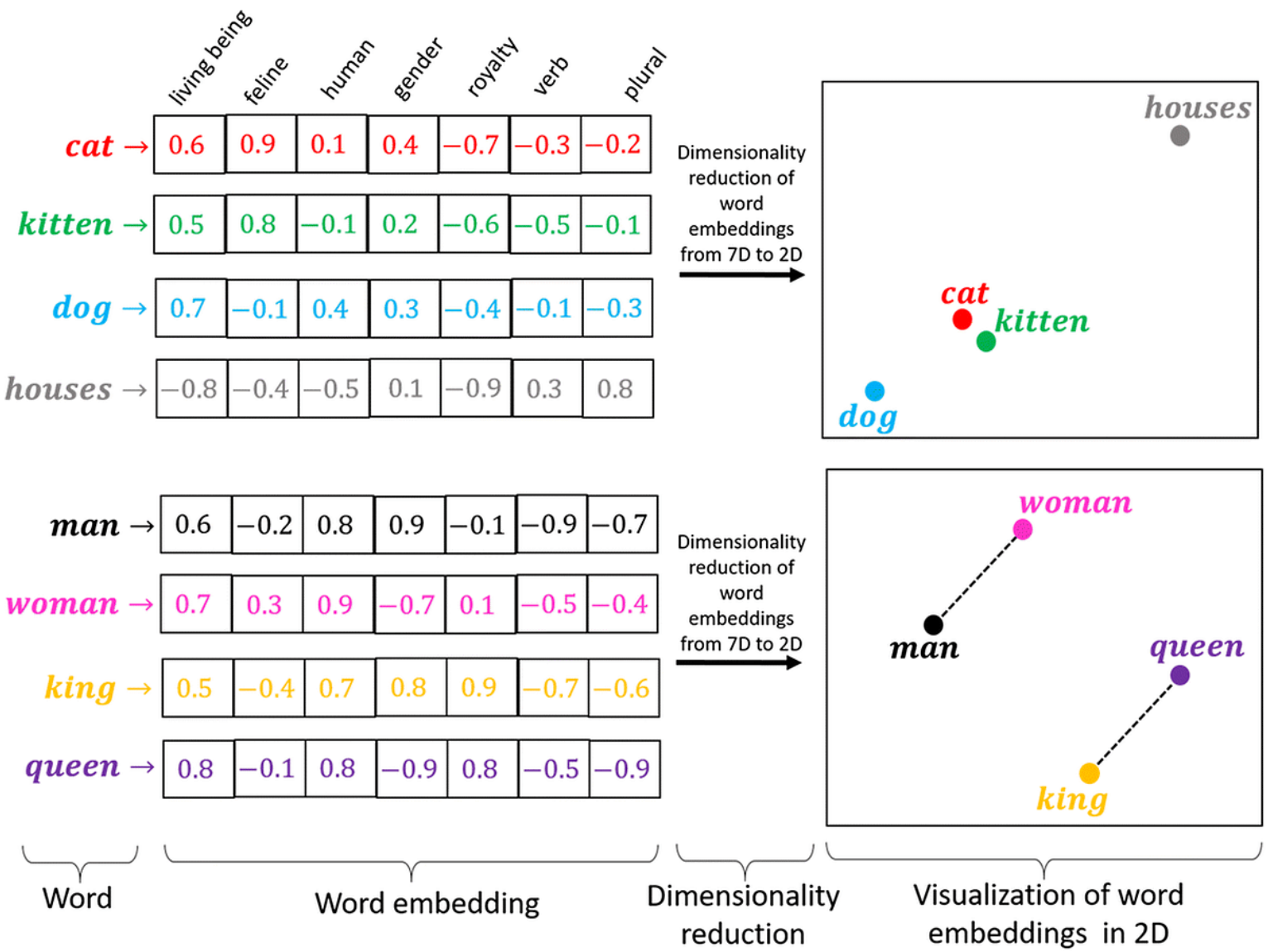
Jeffrey Dean
Google Inc., Mountain View, CA
jeff@google.com

781v3 [cs.CL] 7 Sep 2013

Abstract

We propose two novel model architectures for computing continuous vector representations of words from very large data sets. The quality of these representations is measured in a word similarity task, and the results are compared to the previously best performing techniques based on different types of neural networks. We observe large improvements in accuracy at much lower computational cost, i.e. it takes less than a day to learn high quality word vectors from a 1.6 billion words data set. Furthermore, we show that these vectors provide state-of-the-art performance on our test set for measuring syntactic and semantic word similarities.

https://arxiv.org/pdf/1301.3781.pdf



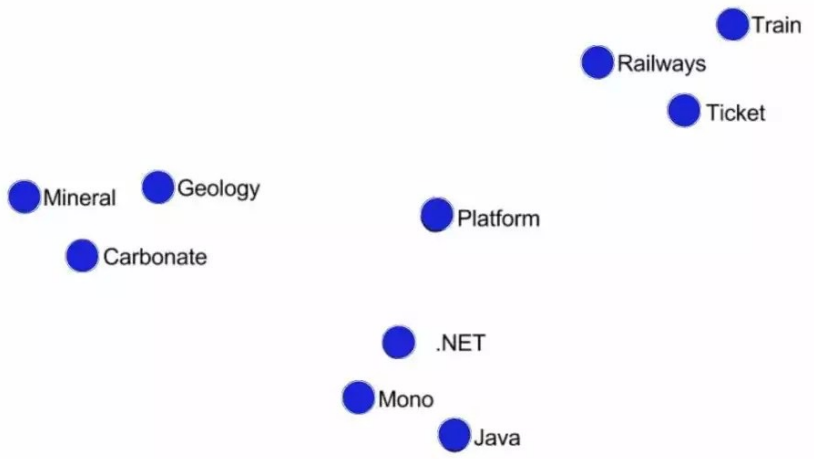
https://medium.com/@hari4om/word-embedding-d816f643140

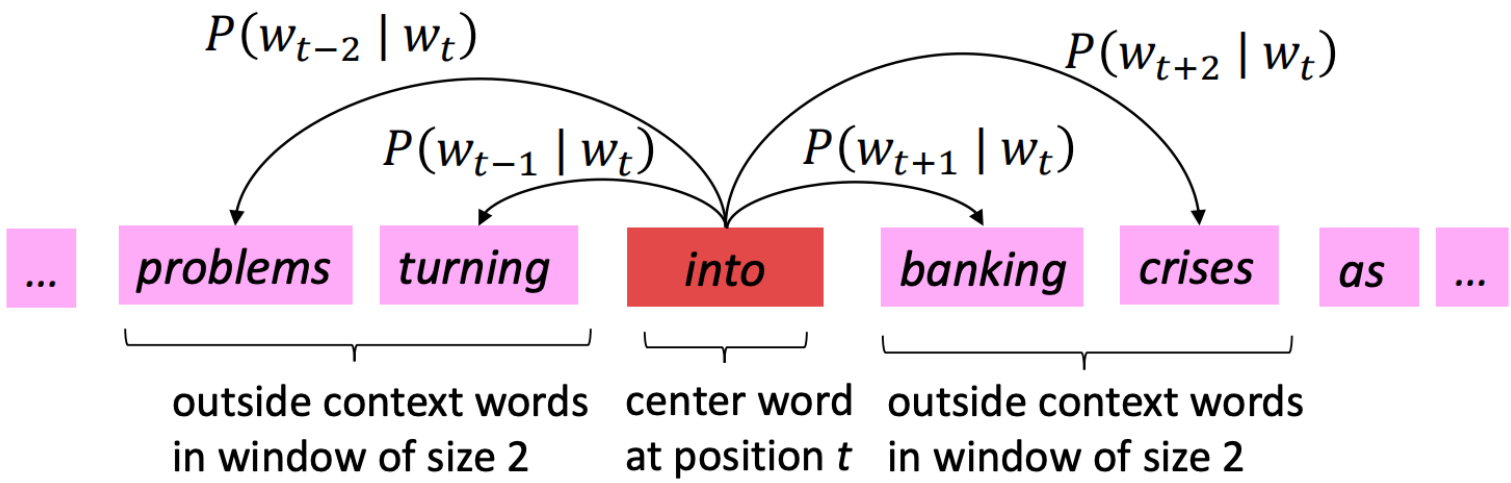
From Words to Vectors

Each word is represented as a **dense vector**, a vector of real numbers, not just 1s and 0s.

Words that appear in **similar contexts** are mapped to **similar vectors**. In this way, the semantic similarity becomes geometric proximity.

Semantic similarity is measured using the **dot product** (or cosine similarity).

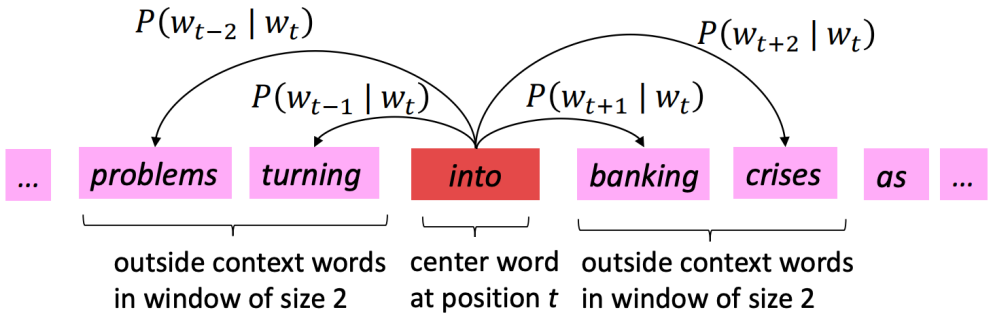




https://web.stanford.edu/class/cs224n/

How to Build Word Embeddings?

- Start with a large text **corpus** and a fixed vocabulary
- Assign each word a vector, initialized **randomly**
- Slide a context window over the text
- For each position, choose a **center word** and its **context words**
- Use the current word vectors to compute the **probability** (e.g., dot product + softmax) that a context word happens given the center word (or vice versa)
- Update the word vectors to **maximise the probability** of the occurrence of words that actually appear in the context (minibatch gradient descent).
- **Repeat** over the entire corpus until vectors stabilize



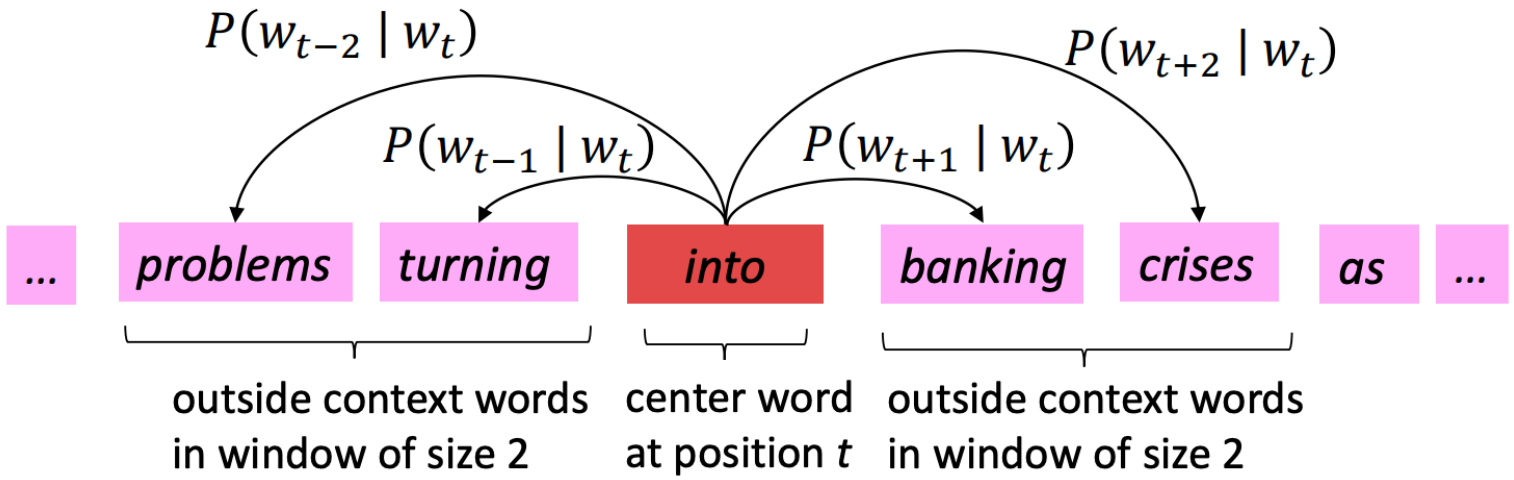
How to Compute the Probability?

② Exponentiation makes anything positive

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

① Dot product compares similarity of *o* and *c*.
$u^T v = u.v = \sum_{i=1}^{n} u_i v_i$
Larger dot product = larger probability

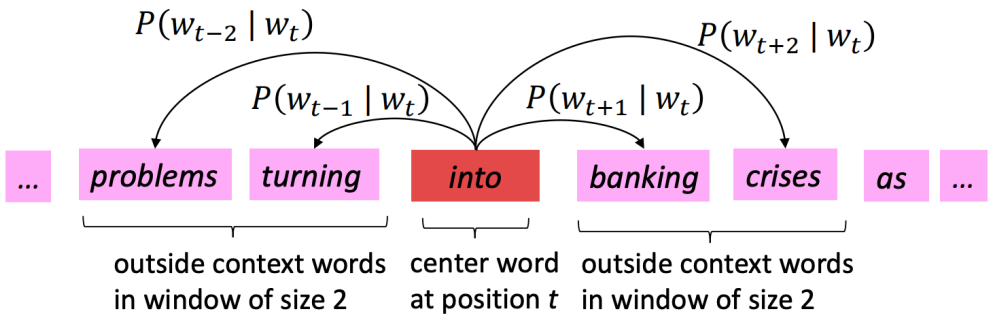
③ Normalize over entire vocabulary to give probability distribution



https://web.stanford.edu/class/cs224n/

About Word2Vec

- There are two variants of the training algorithm: one predicts the center word from a context word, and the other predicts the context word from the center word. In practice, the best results are obtained by **using both**.
- To make training feasible, optimizations such as negative sampling were introduced to avoid computing the **probability denominator**, which would otherwise require summing over billions of terms.
- More advanced ideas were later proposed, such as encoding meaning in **vector differences** (GloVe, 2014).
- However, all these approaches produce a **single vector for each word (bag of words)**. This limitation leads to several issues…



② Exponentiation makes anything positive

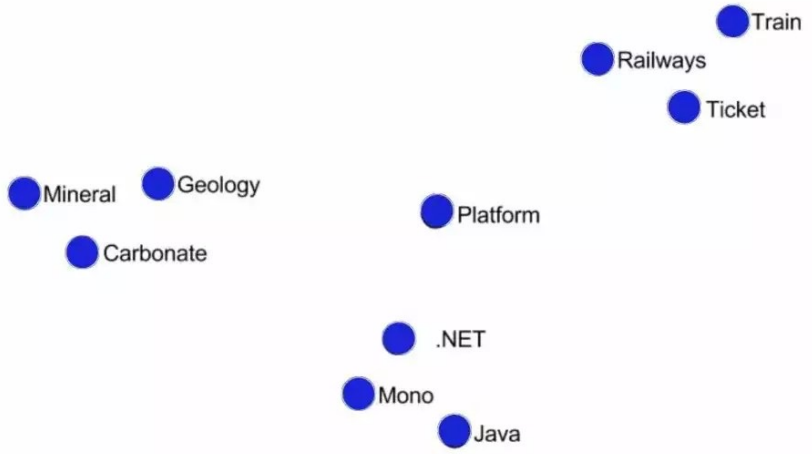
$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

① Dot product compares similarity of *o* and *c*.
$u^T v = u.v = \sum_{i=1}^{n} u_i v_i$
Larger dot product = larger probability

③ Normalize over entire vocabulary to give probability distribution

Interactive Visualization

Visualize embeddings



The seal put her seal of approval.

Embeddings Recap

Artificial intelligence models are designed to process numerical values, which requires human language to be converted into a mathematical representation through a two-step process.

In denotational semantics, a word consists of a signifier (symbols) and a signified (the idea it represents), but early attempts to model this using large graphs like WordNet were difficult to build and could not represent relationships between every pair of words.

While sparse vectors consisting of 1s and 0s were used initially, they were unable to represent similarity between words. Distributional semantics addresses this by suggesting that you shall know a word by the company it keeps, meaning context defines representation.

Modern embeddings are dense vectors of real numbers generated by models trained on massive text collections where words appearing in similar contexts are mapped to similar vectors. This shift ensures that semantic similarity becomes geometric proximity, typically measured using dot products or cosine similarity.

High-quality word vectors can be learned efficiently from datasets containing billions of words, as demonstrated by the Word2Vec architectures introduced in 2013.

Language Models

2018

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Jacob Devlin **Ming-Wei Chang** **Kenton Lee** **Kristina Toutanova**
Google AI Language
{jacobdevlin,mingweichang,kentonl,kristout}@google.com

04805v2 [cs.CL] 24 May 2019

Abstract

We introduce a new language representation model called **BERT**, which stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers. Unlike recent language representation models (Peters et al., 2018a; Radford et al., 2018), BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and language inference, without substantial task-specific architecture modifications.

BERT is conceptually simple and empirically powerful. It obtains new state-of-the-art results on eleven natural language processing tasks, including pushing the GLUE score to 80.5% (7.7% point absolute improvement), MultiNLI accuracy to 86.7% (4.6% absolute improvement), SQuAD v1.1 question answer-

There are two existing strategies for applying pre-trained language representations to downstream tasks: *feature-based* and *fine-tuning*. The feature-based approach, such as ELMo (Peters et al., 2018a), uses task-specific architectures that include the pre-trained representations as additional features. The fine-tuning approach, such as the Generative Pre-trained Transformer (OpenAI GPT) (Radford et al., 2018), introduces minimal task-specific parameters, and is trained on the downstream tasks by simply fine-tuning *all* pre-trained parameters. The two approaches share the same objective function during pre-training, where they use unidirectional language models to learn general language representations.

We argue that current techniques restrict the power of the pre-trained representations, especially for the fine-tuning approaches. The major limitation is that standard language models are unidirectional, and this limits the choice of architectures that can be used during pre-training. For

https://arxiv.org/pdf/1810.04805.pdf

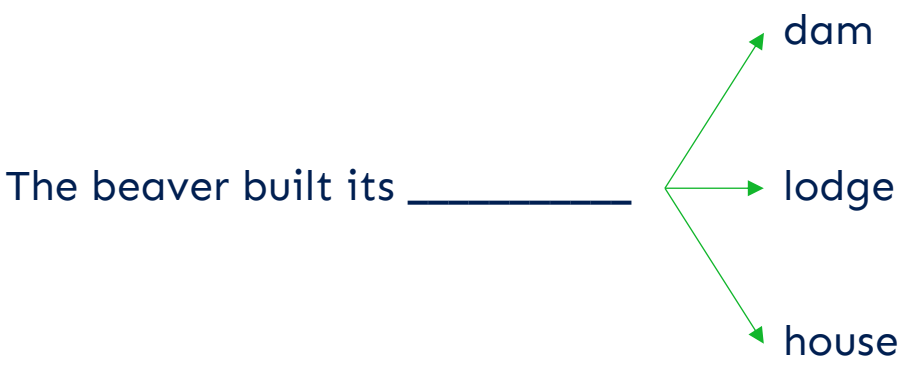
Language Modelling

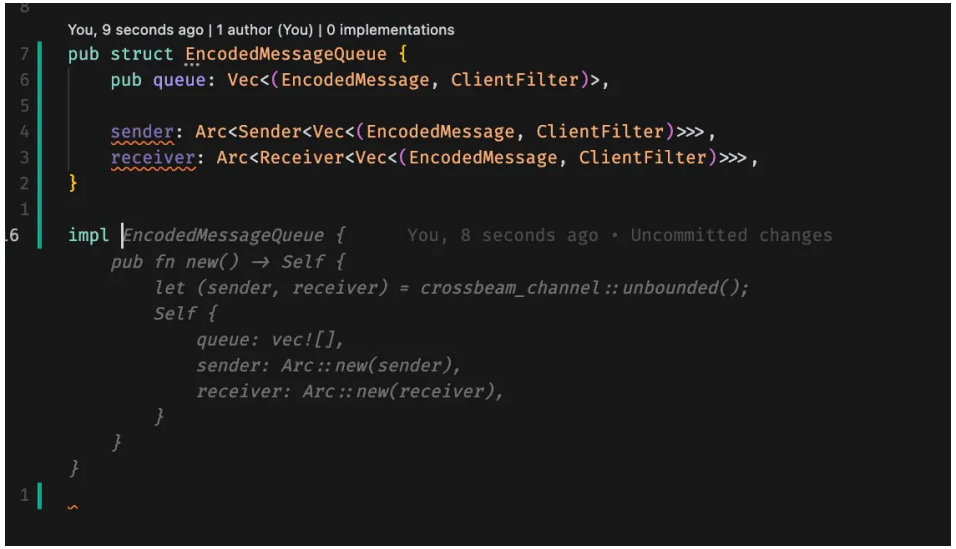
Language modelling is the task of **predicting the next word** in a piece of text.

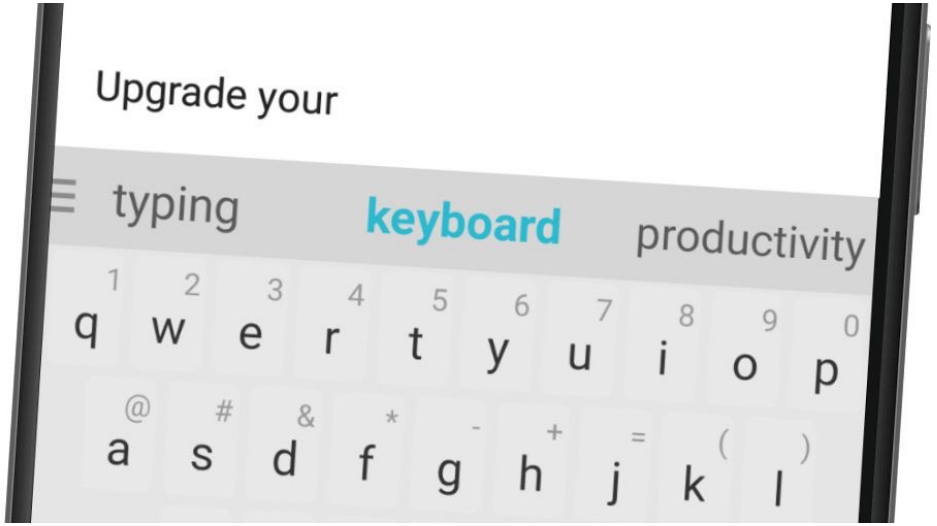
More formally, given a sequence of words, it is about **computing the probability distribution** of the next word over a fixed set of possible words, called the vocabulary.

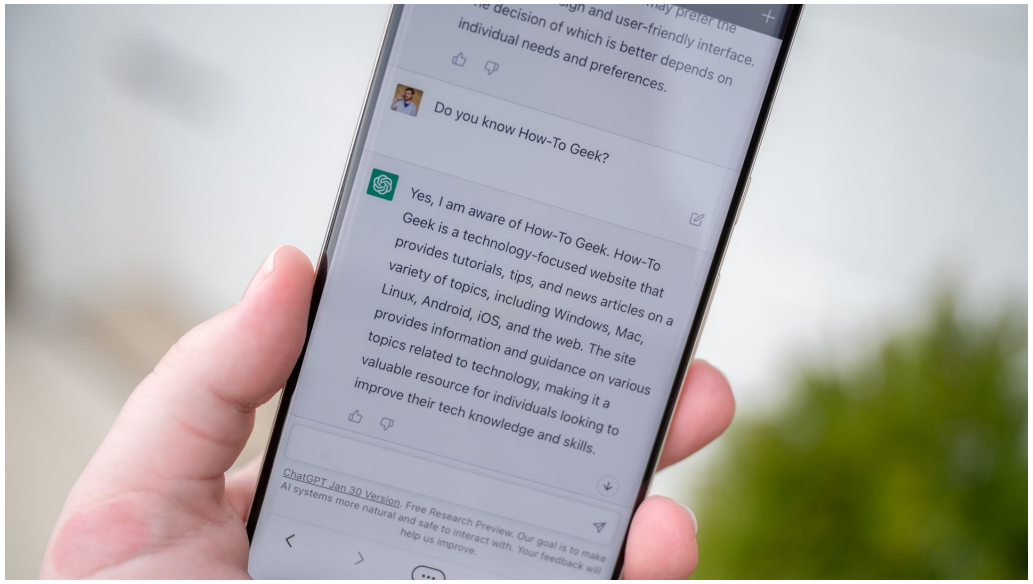
A system that performs this task is called a language model.

GPT, Gemini, Claude, and similar systems are all **language models**—though today they are capable of much more, as we will see.




You, 9 seconds ago | 1 author (You) | 0 implementations
pub struct EncodedMessageQueue {
pub queue: Vec<(EncodedMessage, ClientFilter)>,
sender: Arc<Sender<Vec<(EncodedMessage, ClientFilter)>>>,
receiver: Arc<Receiver<Vec<(EncodedMessage, ClientFilter)>>>,
}
impl EncodedMessageQueue {
You, 8 seconds ago • Uncommitted changes
pub fn new() -> Self {
let (sender, receiver) = crossbeam_channel::unbounded();
Self {
queue: vec![],
sender: Arc::new(sender),
receiver: Arc::new(receiver),
}
}
}


Upgrade your
typing
keyboard
productivity


Do you know How-To Geek?
Yes, I am aware of How-To Geek. How-To Geek is a technology-focused website that provides tutorials, tips, and news articles on a variety of topics, including Windows, Mac, Linux, Android, iOS, and the web. The site provides information and guidance on various topics related to technology, making it a valuable resource for individuals looking to improve their tech knowledge and skills.

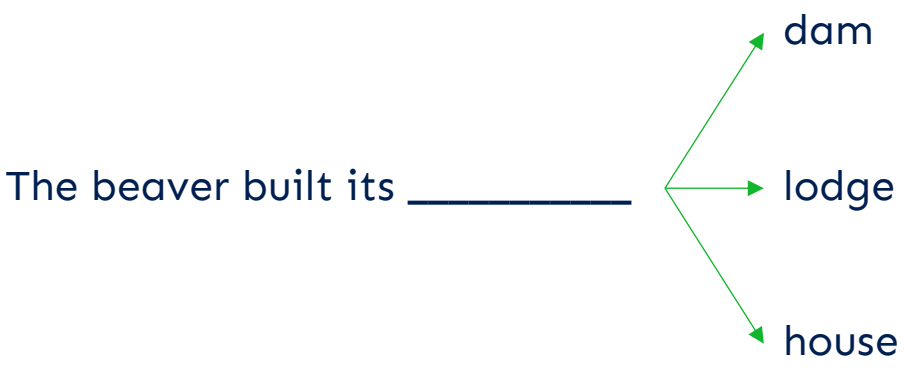
Google
how to teach
how to teach english
how to teach english in italy
how to teach english to kids
how to teach cockatiel to talk
how to teach italian to beginners
how to teach english online
how to teach italian online
how to teach ielts
how to teach kids to read
how to teach

LM as Foundation

Language modelling is a **foundational** task.

When it is solved with sufficiently strong performance, it enables **many other AI problems** to be solved, including:

- Named entity recognition
- Summarization
- Text generation
- Question answering
- Sentiment analysis
- Parts of data mining
- Machine translation



N-Grams

How to build a Language Model?

The first attempt was with **n-grams**.

An n-gram is a **sequence** of n **consecutive** words.

Unigram: “the”, “beaver”, “built”, ...

Bigram: “the beaver”, “beaver built”, “built its”, ...

Trigram: “the beaver built”, “beaver built its”, ...

The beaver built its _____ → dam, lodge, house

N-Grams

We assume that the word n **only depends** on the previous n-1 words.

Then,

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

And we can estimate the probabilities by simply **counting** the occurrences over large enough corpus of text.

$$P(dam|the\ bever\ built\ its) = \frac{count(the\ beaver\ built\ its\ dam)}{count(the\ beaver\ built\ its)}$$

To generate text, we **sample** from the resulting probability distribution.

The beaver built its _____

- dam
- lodge
- house

Text Generation with N-Grams

today the price _____

condition on this

get probability distribution

of	0.308
for	0.050
it	0.046
to	0.046
is	0.031
…	

sample

today the price of _____

condition on this

get probability distribution

the	0.072
18	0.043
oil	0.043
its	0.036
gold	0.018
…	

sample

Discussion

What are the problems of n-grams?

Lots of Problems with N-Grams!

- What is a specific n-gram **never** occurred in the training corpus?
- The storage needed to store all the n-grams **scales** with n and with the corpus.
- How to account for **different languages**?
- We learnt that embeddings work. How can we **link n-grams with embeddings**?

$$P(dam|the\ bever\ built\ its) =$$

$$\frac{count(the\ beaver\ built\ its\ dam)}{count(the\ beaver\ built\ its)}$$

Language Modelling with DL

output distribution

$$\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{U}\boldsymbol{h} + \boldsymbol{b}_2) \in \mathbb{R}^{|V|}$$

hidden layer

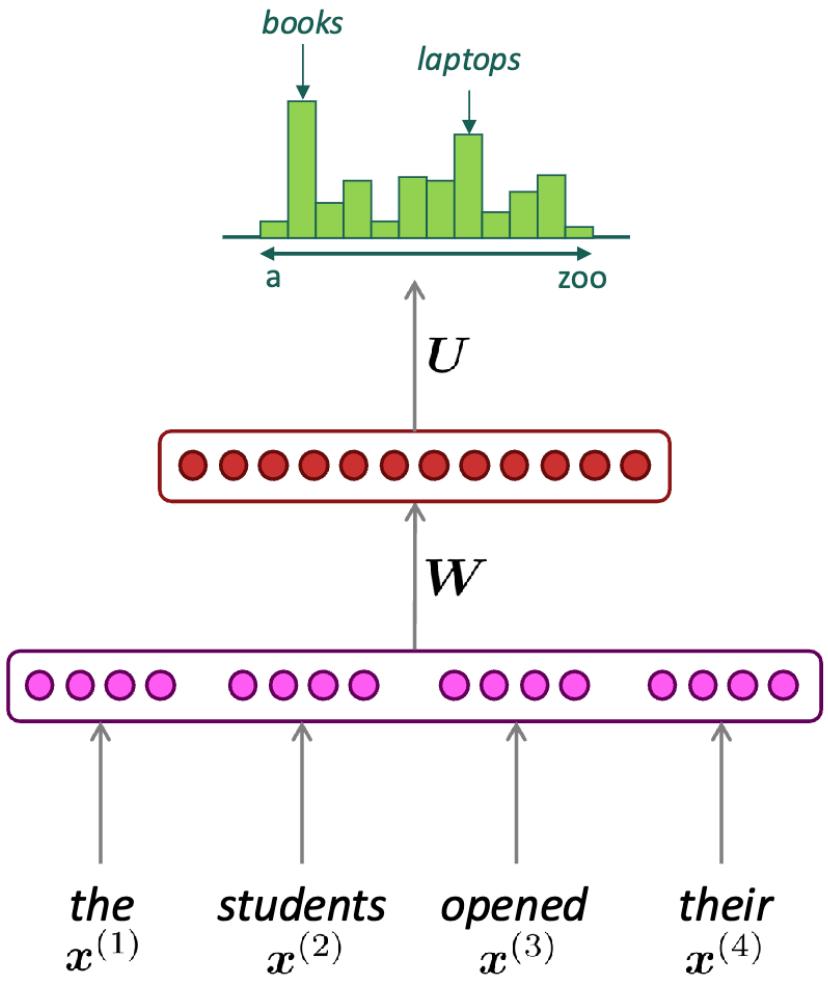
$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b}_1)$$

concatenated word embeddings

$$\boldsymbol{e} = [\boldsymbol{e}^{(1)}; \boldsymbol{e}^{(2)}; \boldsymbol{e}^{(3)}; \boldsymbol{e}^{(4)}]$$

words / one-hot vectors

$$\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(3)}, \boldsymbol{x}^{(4)}$$



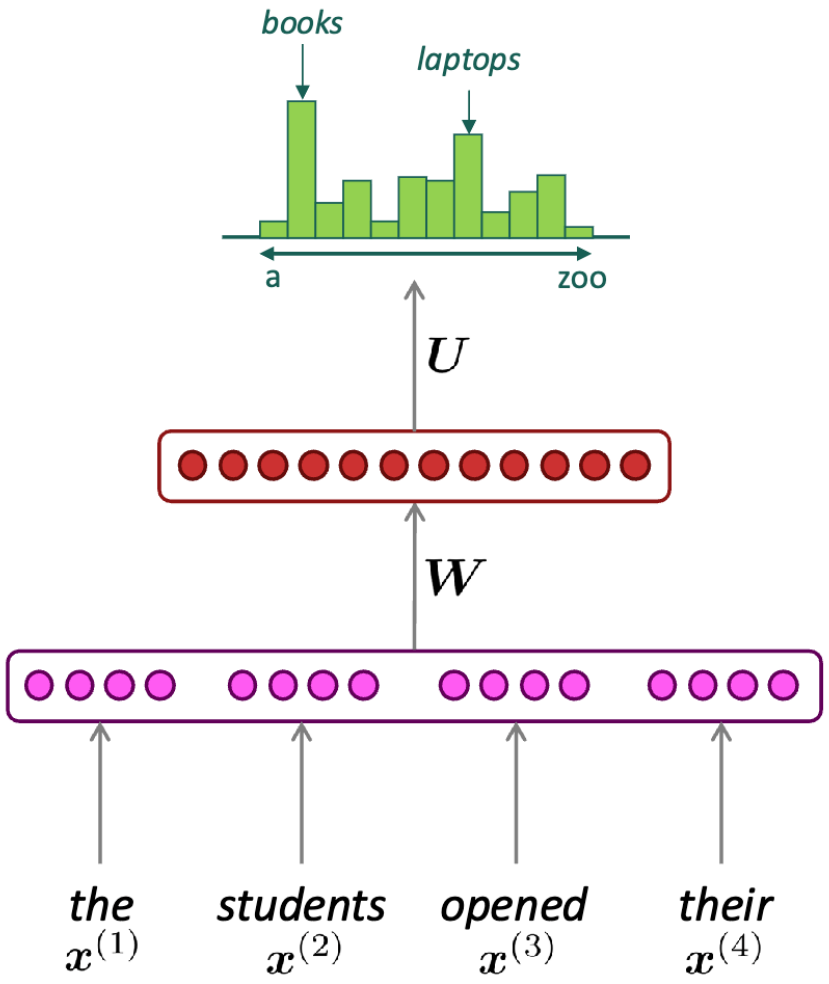
Lots of Problems still!

The deep learning model solved the problem of missing n-grams.

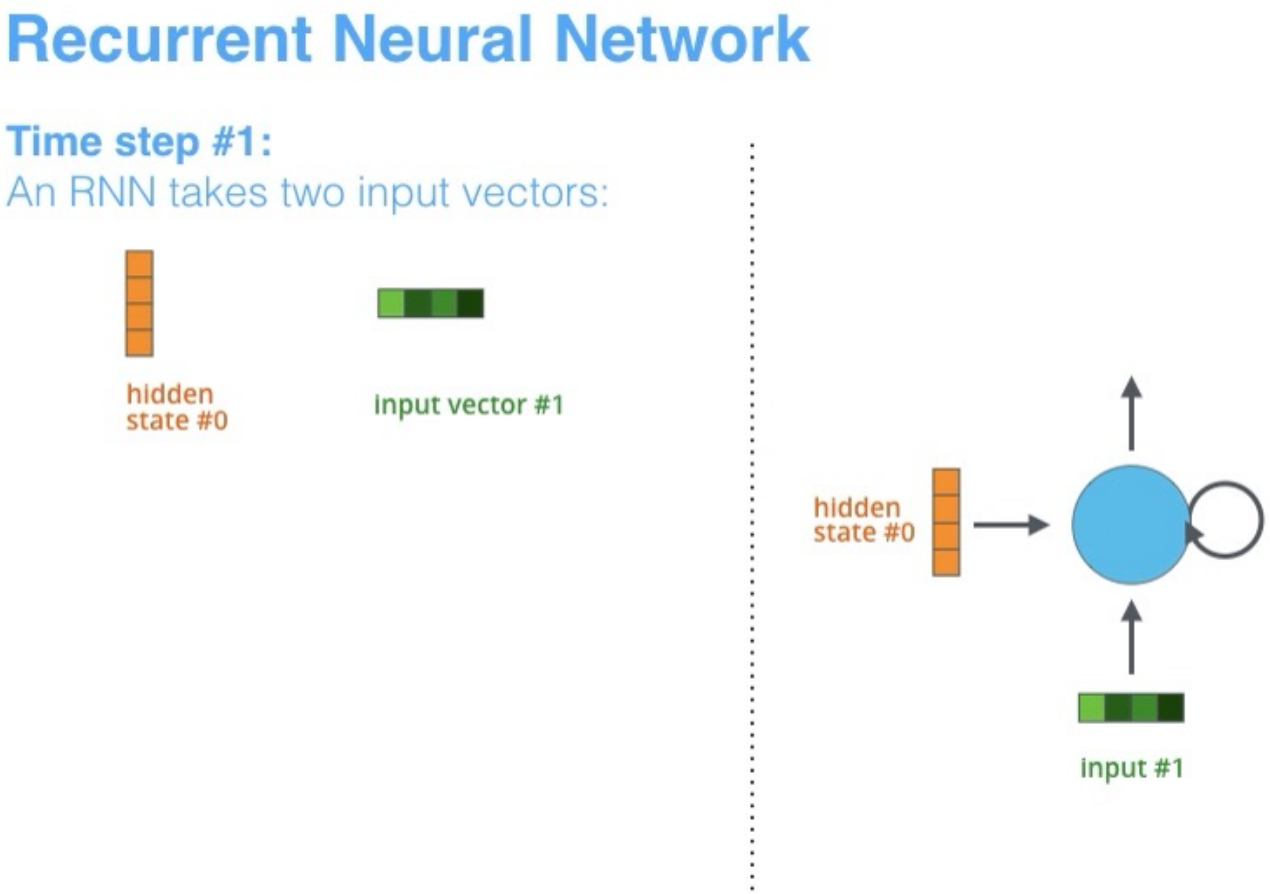
In addition, the model size no longer scales with the number of n-grams observed.

However, the model size still **scales with n**, the size of the context window. To effectively solve language modelling, we need **very large n values** (for example, Gemini currently uses around 1 million tokens, or about 800,000 words).

This means we need a **different architecture**—one that can handle much larger n values, or equivalently, much longer **context windows**.



Recurrent Neural Networks



https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

Recurrent Neural Networks

output distribution

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$$

hidden states

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

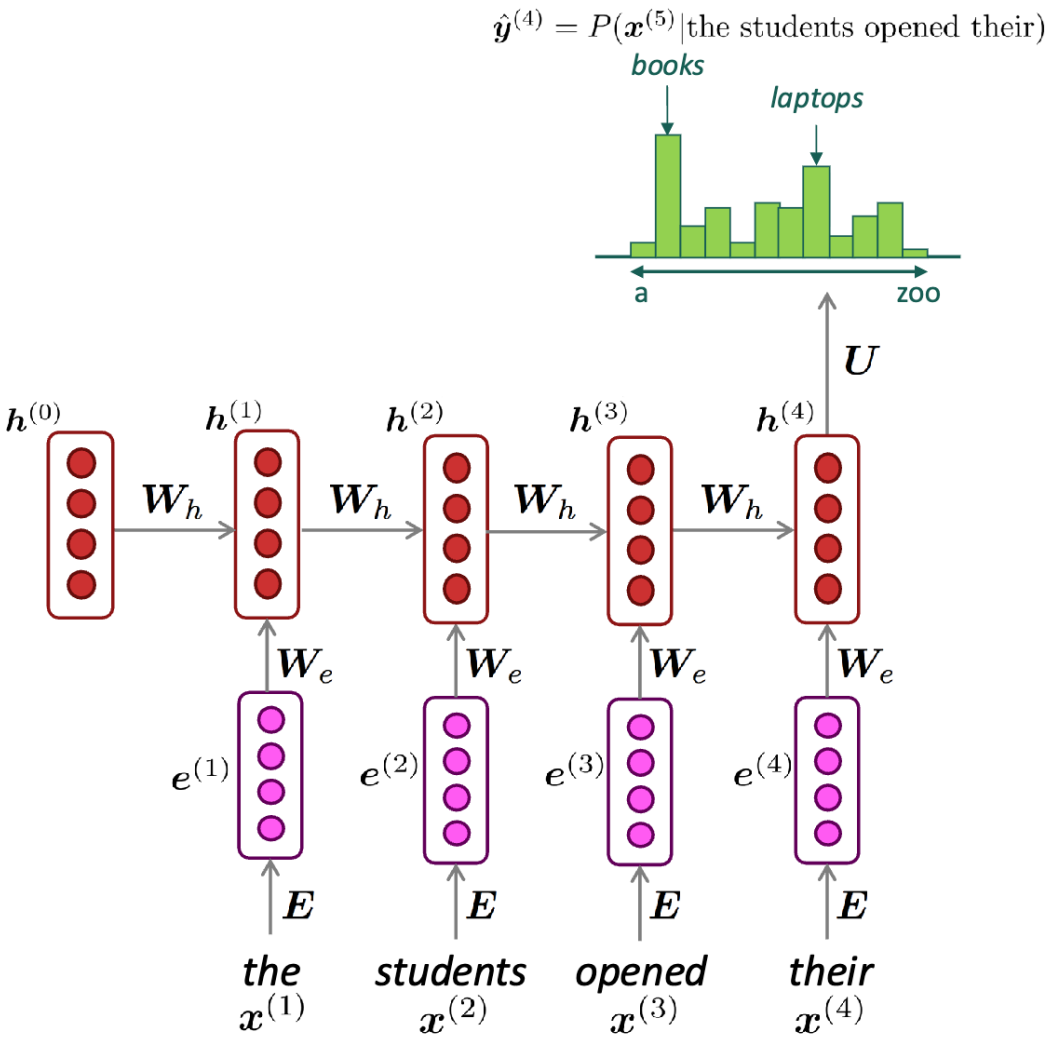
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

$$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$$

words / one-hot vectors

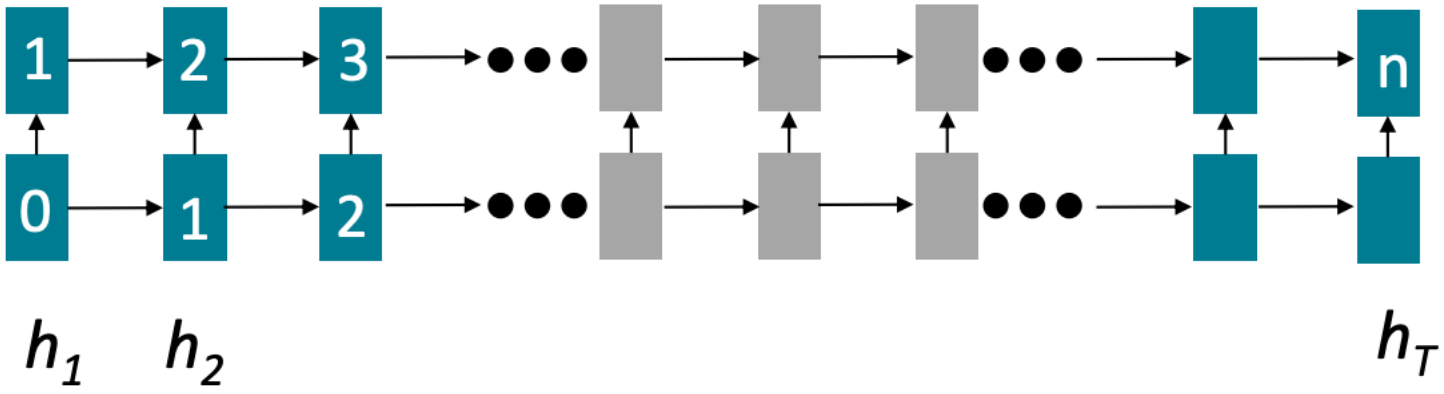
$$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$$



E, W_e, W_h are shared, so the model size no longer scales with the length of the context window.

In theory, this model can process **infinitely long** sequences.

In practice, however,



Inherently sequential

Could not "remember" context

Language Models Recap

Language modelling is the foundational task of predicting the next word in a sequence by computing a probability distribution over a fixed vocabulary.

Early attempts used n-grams, which assumed a word only depends on the previous n-1 words and estimated probabilities by counting occurrences in a corpus.

However, n-grams faced significant issues, as they could not handle sequences never seen in training and their storage needs scaled poorly with both the sequence length and the corpus size. Deep learning architectures solved the problem of missing sequences, but early models still scaled in size based on the context window.

Recurrent Neural Networks (RNNs) introduced shared weights so that model size no longer scaled with the window length, theoretically allowing for infinite sequences. Despite this, RNNs were inherently sequential and proved unable to effectively "remember" long contexts in practice.

Modern systems like GPT and Gemini have evolved beyond these limitations to provide high-performance solutions for summarization, translation, and sentiment analysis.

Transformers

2017

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

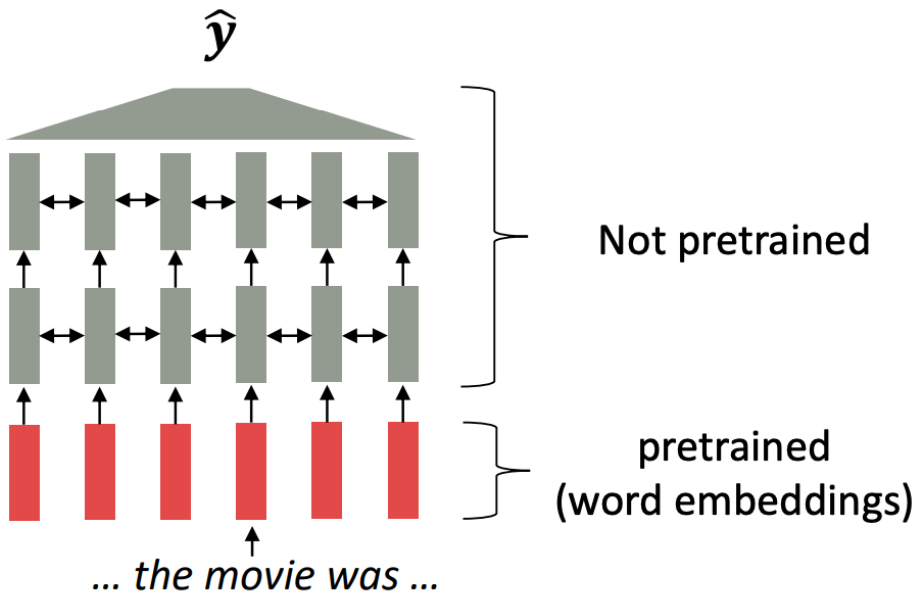
v:1706.03762v5 [cs.CL] 6 Dec 2017

Abstract

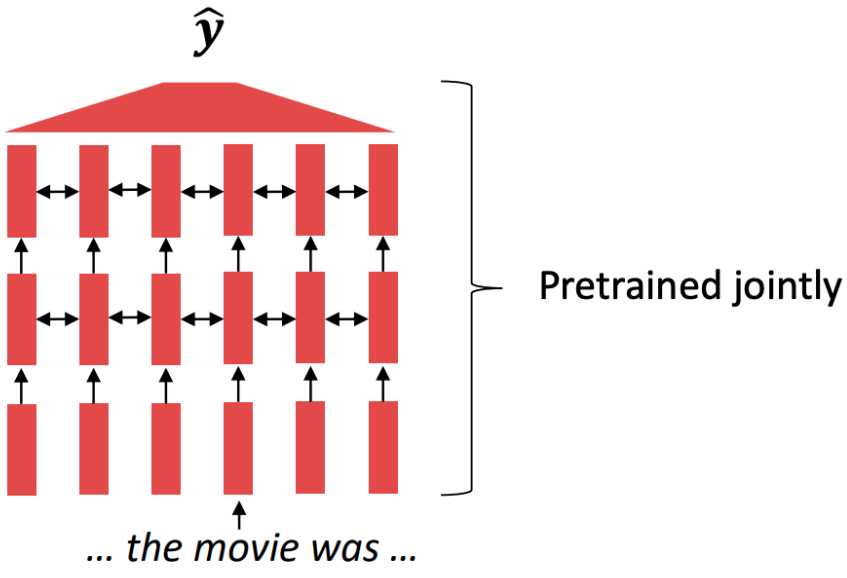
The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with

https://arxiv.org/pdf/1706.03762.pdf

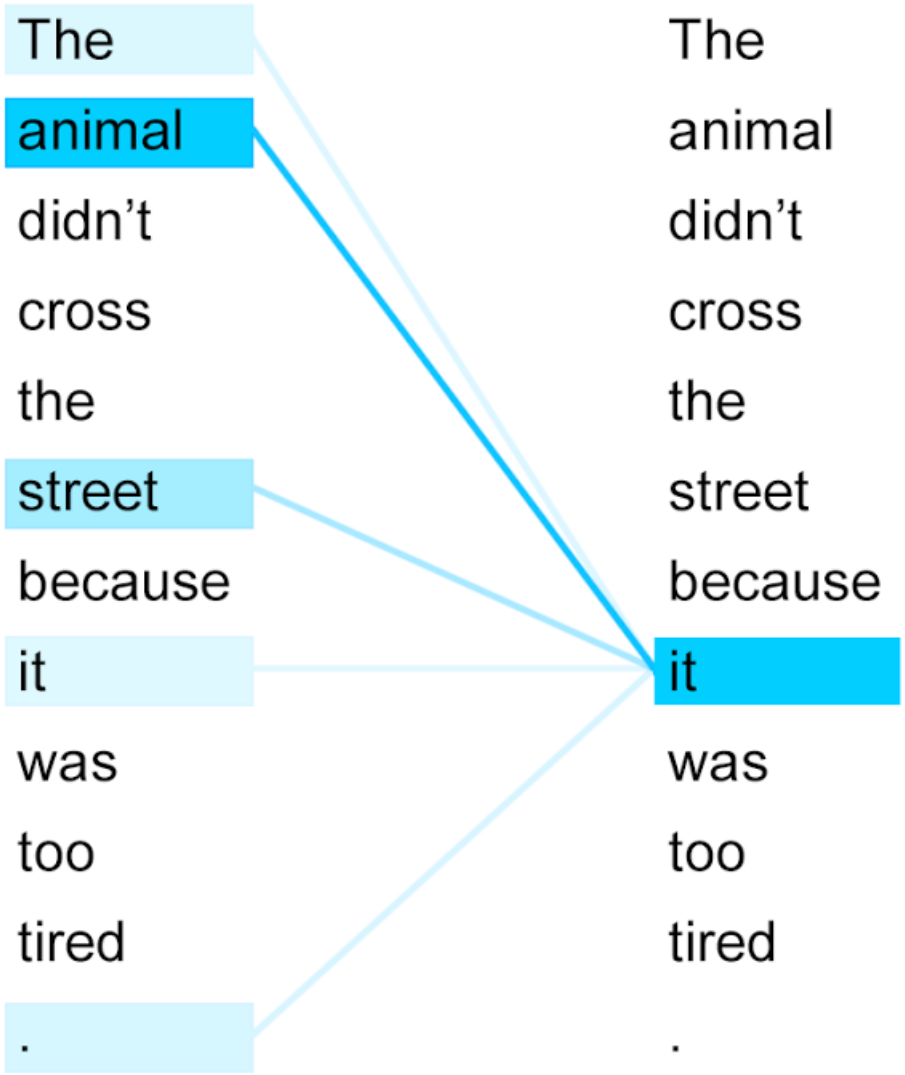
Before 2018



After 2018



https://web.stanford.edu/class/cs224n/



https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

Self-Attention

Let $\boldsymbol{w}_{1:n}$ be a sequence of words in vocabulary V, like *Zuko made his uncle tea*.

For each $\boldsymbol{w}_i$, let $\boldsymbol{x}_i = E\boldsymbol{w}_i$, where $E \in \mathbb{R}^{d \times |V|}$ is an embedding matrix.

1. Transform each word embedding with weight matrices $\mathrm{Q}, \mathrm{K}, \mathrm{V}$, each in $\mathbb{R}^{d \times d}$

$\boldsymbol{q}_i = Q\boldsymbol{x}_i$ (queries) $\boldsymbol{k}_i = K\boldsymbol{x}_i$ (keys) $\boldsymbol{v}_i = V\boldsymbol{x}_i$ (values)

2. Compute pairwise similarities between keys and queries; normalize with softmax

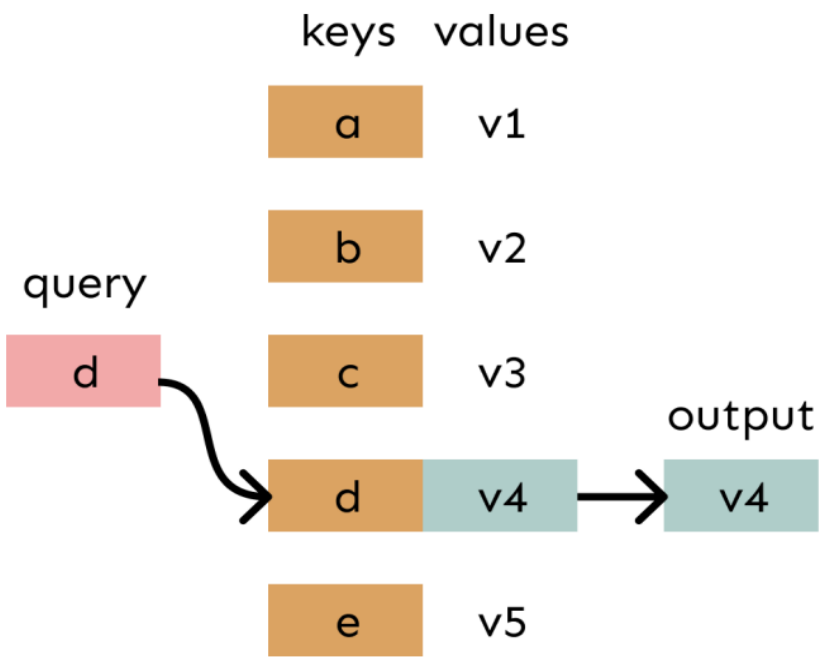
$$\boldsymbol{e}_{ij} = \boldsymbol{q}_i^\top \boldsymbol{k}_j \qquad \boldsymbol{\alpha}_{ij} = \frac{\exp(\boldsymbol{e}_{ij})}{\sum_{j'} \exp(\boldsymbol{e}_{ij'})}$$

3. Compute output for each word as weighted sum of values

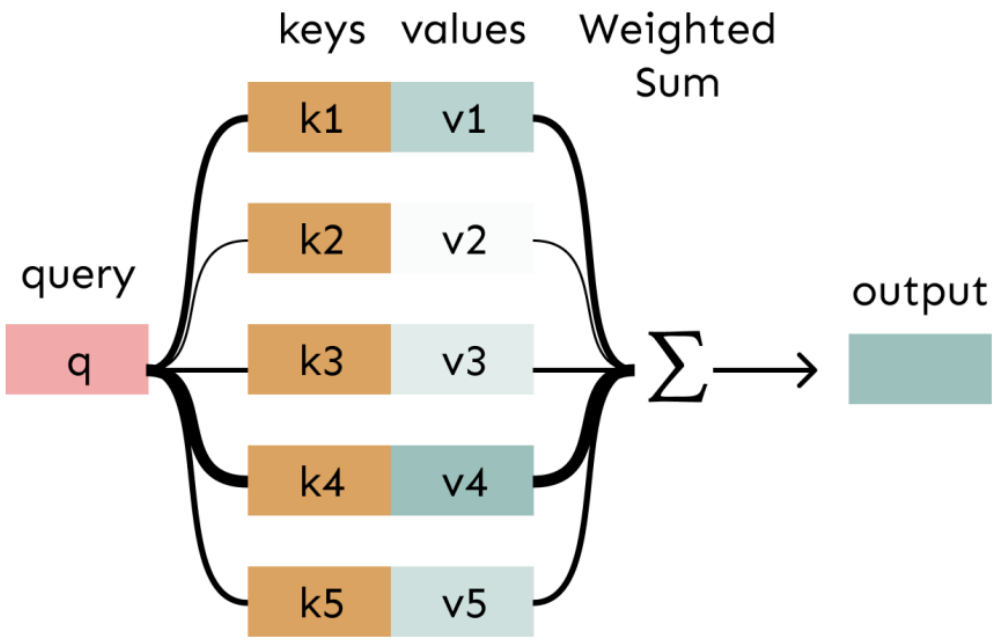
$$\boldsymbol{o}_i = \sum_j \boldsymbol{\alpha}_{ij} \boldsymbol{v}_i$$

https://web.stanford.edu/class/cs224n/

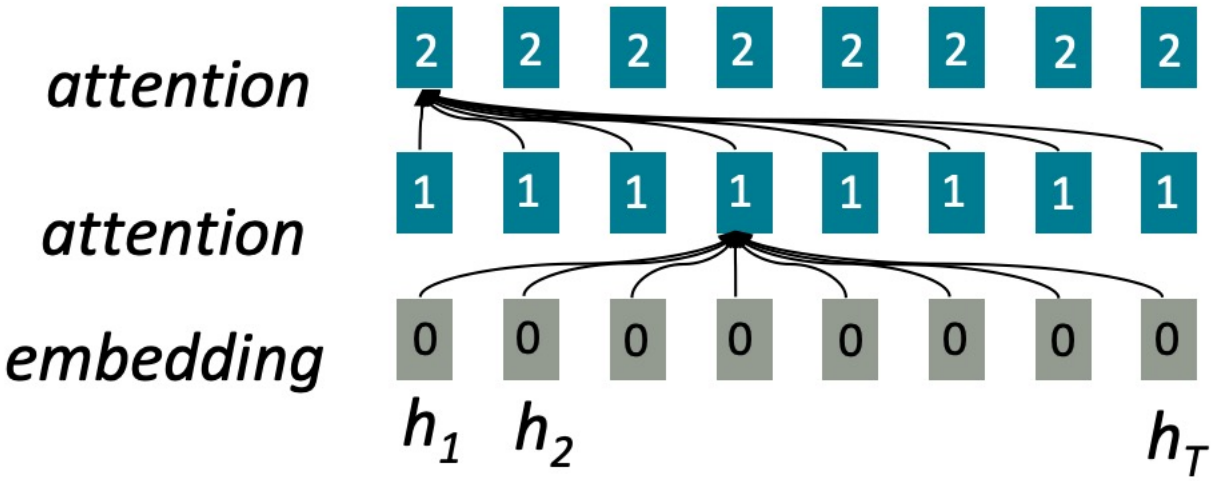
In a standard **lookup table**, we have a set of keys mapping to values. The query matches one of the keys, allowing to retrieve the value.



In **attention**, the query has a "match score" with each key. The values are retrieved and averaged, using the "match scores" as weights.



https://web.stanford.edu/class/cs224n/



Completely parallelizable

Chooses which context to "remember"

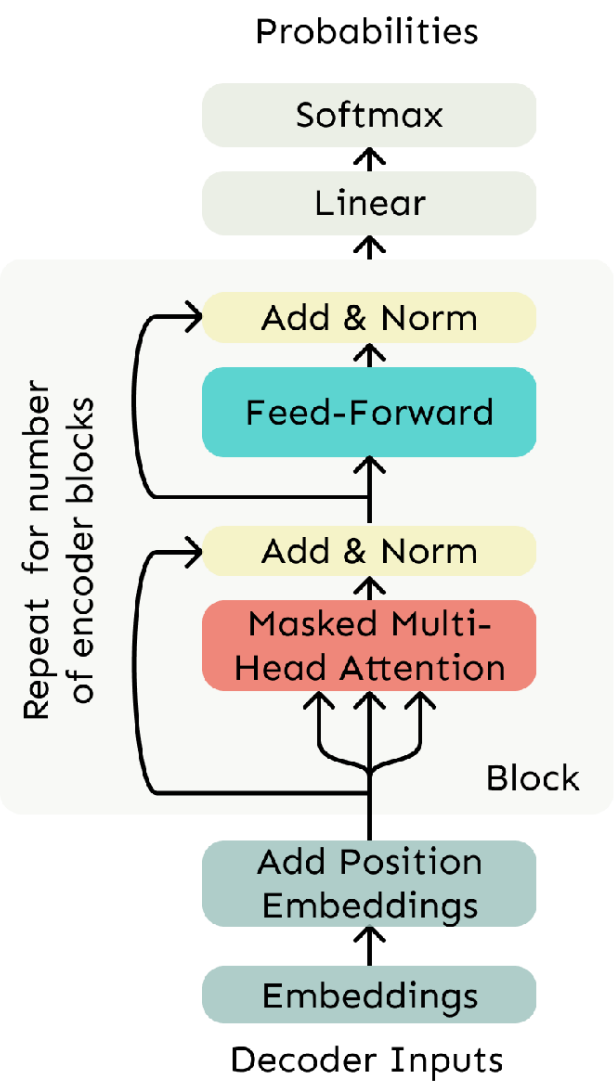
https://web.stanford.edu/class/cs224n/

The Transformer

Note: we focus only on the transformer decoder, which underlies all modern language models. Earlier architectures, such as BERT and previous generations of models, used mainly encoder–decoder designs.

A transformer-based language model consists of multiple transformer blocks.

Each block relies on the self-attention mechanism described above, with a few important improvements.

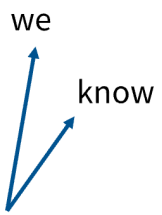


https://web.stanford.edu/class/cs224n/

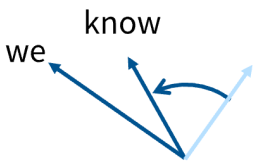
Positional Embeddings

Self-attention has no inherent notion of word order in a sequence—but word order is crucial.

Solution: rotate the embeddings based on their positions. Since embeddings are vectors, this rotation can be implemented simply by multiplying them by an appropriate matrix.

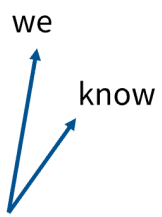


Position independent embedding



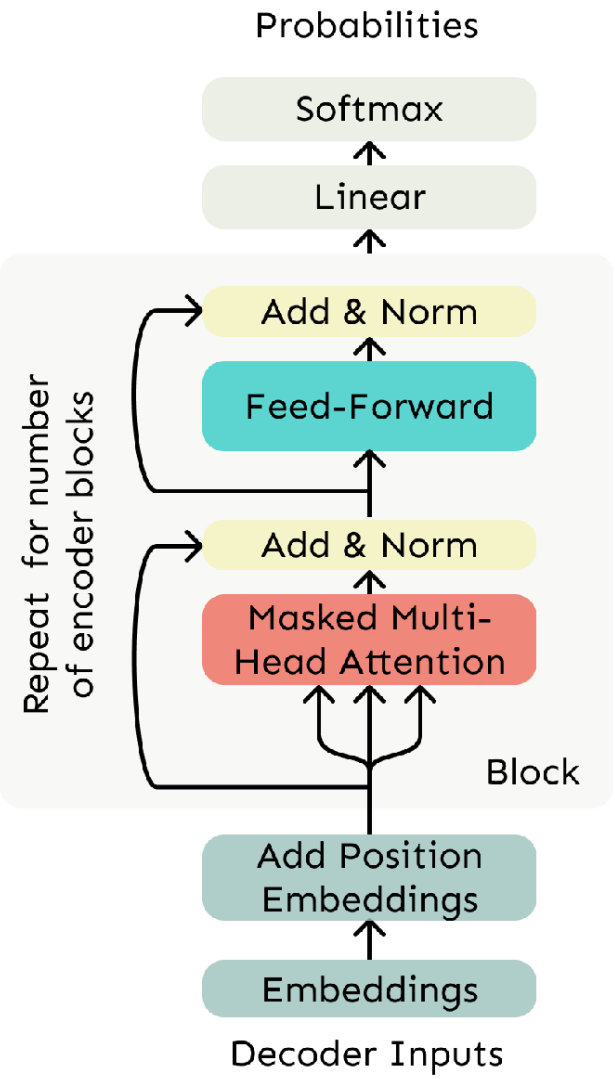
Embedding
“of course we know”

Rotate by ‘2 positions’



Embedding
“we know that”

Rotate by ‘0 positions’



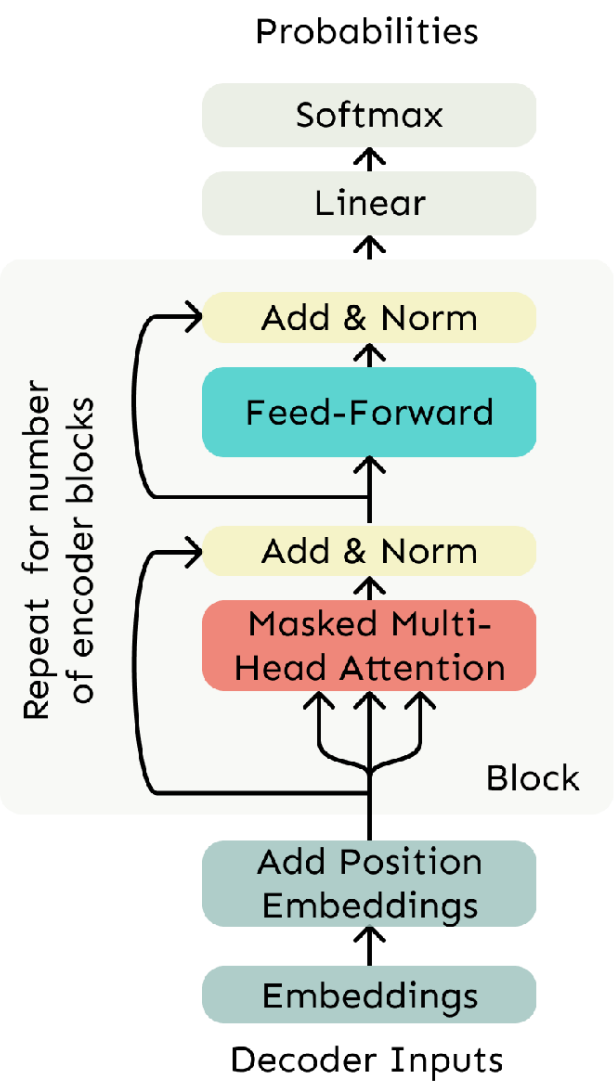
https://web.stanford.edu/class/cs224n/

Feed-Forward

Self-attention is essentially a weighted sum, which by itself is a linear operation.

By the universal approximation theorem, we know that non-linear functions are necessary to model complex patterns.

Solution: add a feed-forward (dense) neural network after each attention layer. In modern large language models, most of the parameters are concentrated in these feed-forward modules.

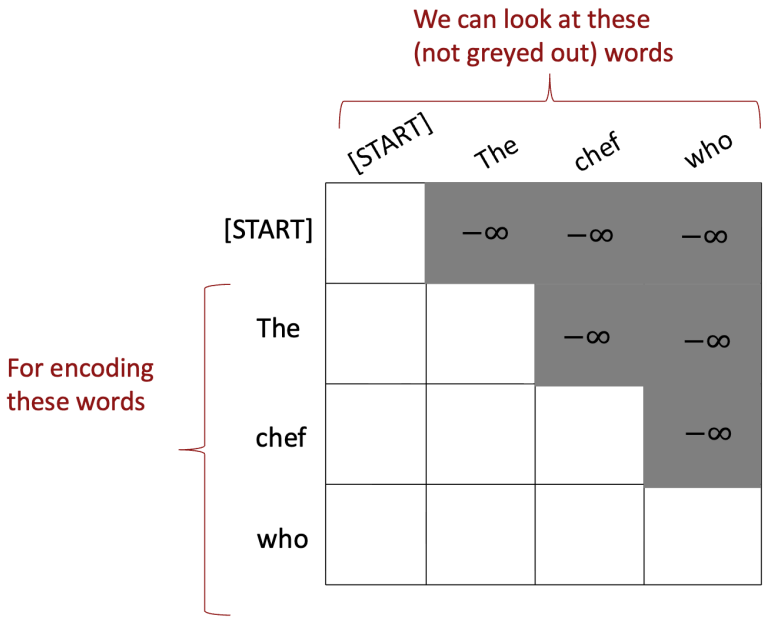


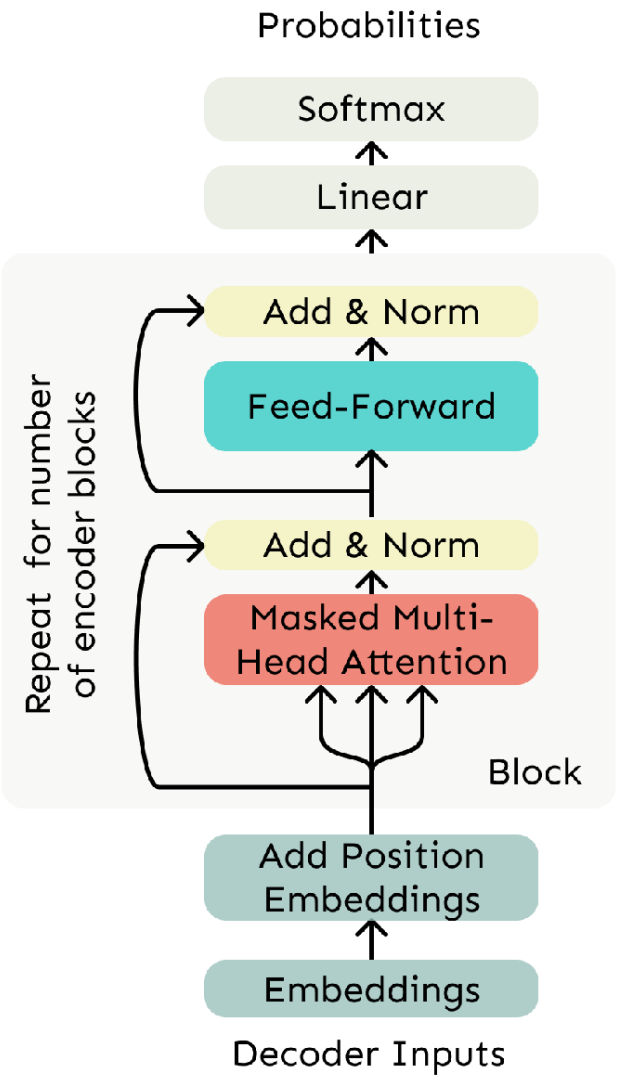
https://web.stanford.edu/class/cs224n/

Seeing the Future

To ensure computational efficiency, the dot products between keys and queries are expressed as matrix multiplications. However, care must be taken to avoid using parts of the text that have not yet been generated.

Solution: mask the key–query products corresponding to words that have not yet appeared in the sentence.



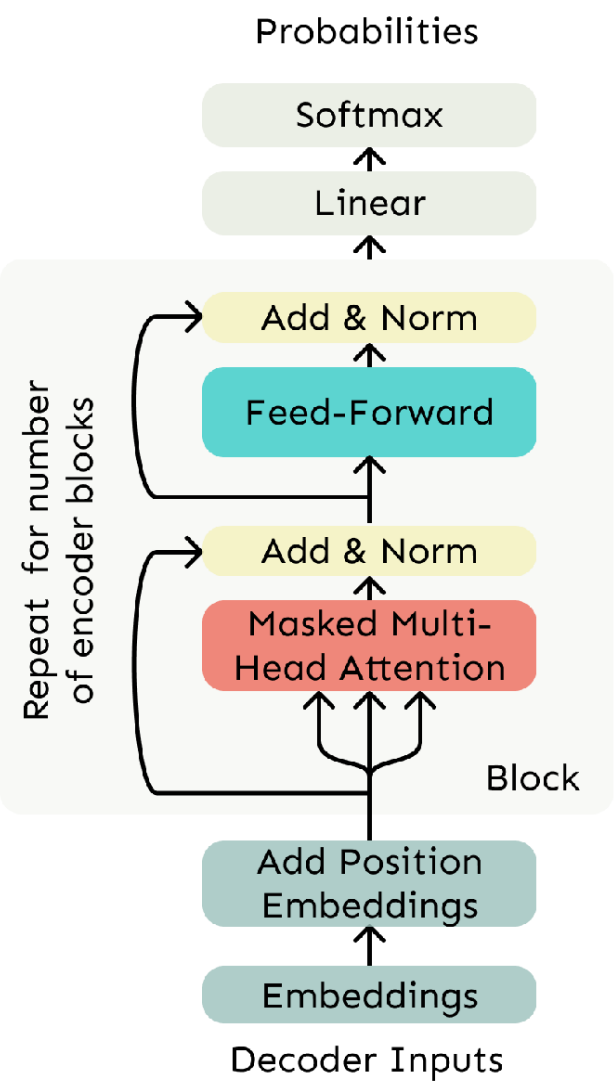


https://web.stanford.edu/class/cs224n/

Controlling Numbers

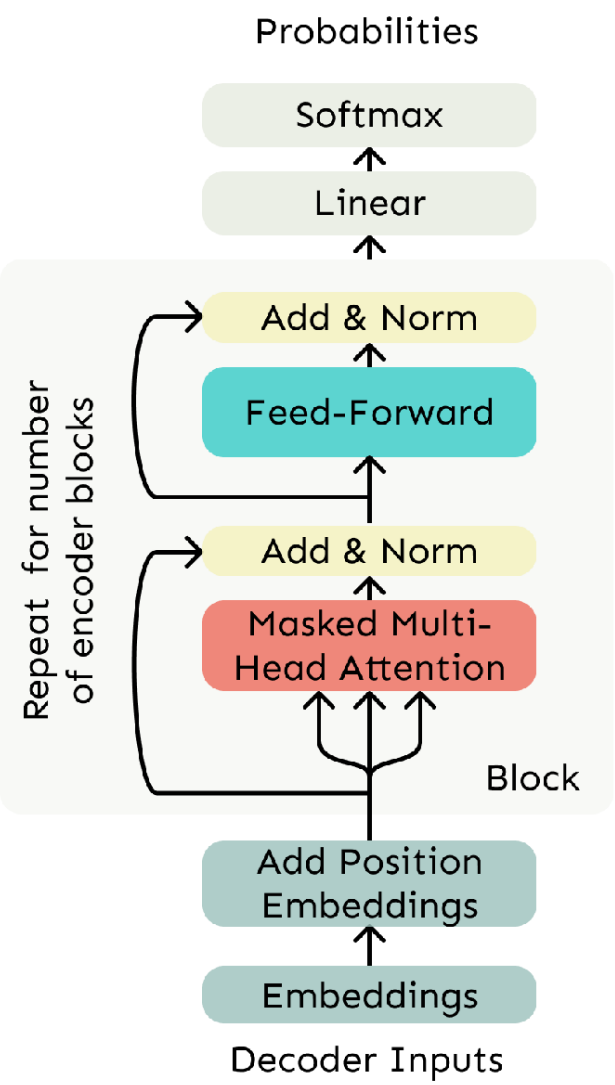
During training, dot products can produce values with very different magnitudes, which slows down the convergence of gradient descent.

Solution: normalize the output of each layer (add & norm) to have zero mean and unit standard deviation. This stabilizes the scale of activations and helps optimization methods converge faster.



https://web.stanford.edu/class/cs224n/

Multi-head Attention

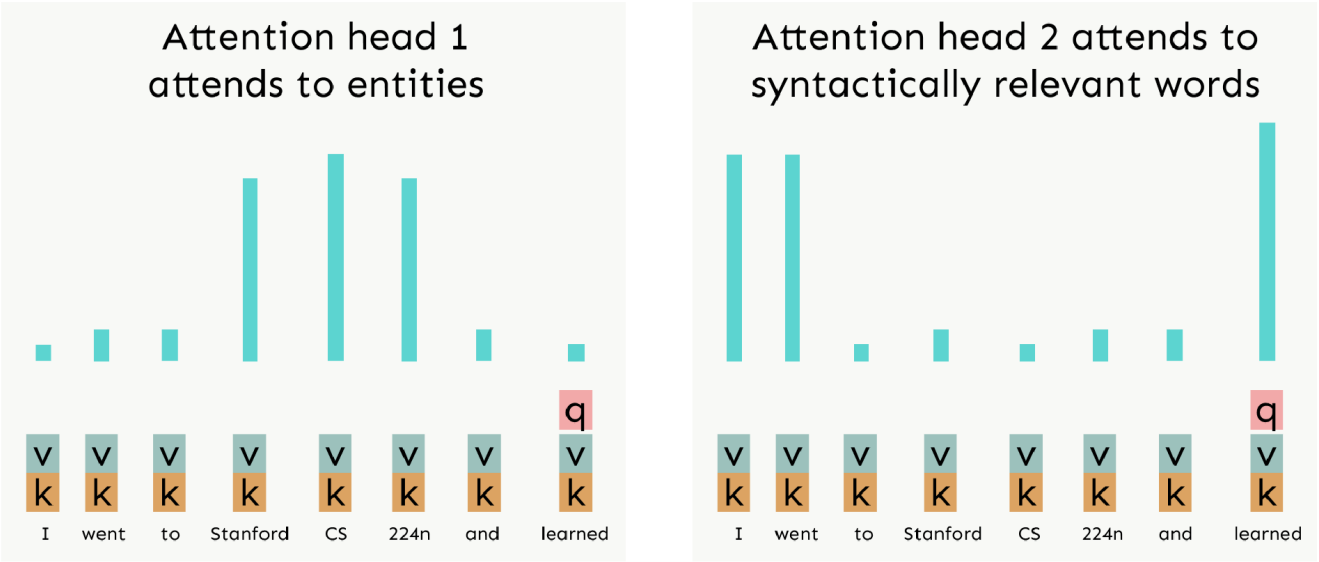


A word's importance depends on multiple factors. Using a single weighted average forces the attention module to handle everything at once.

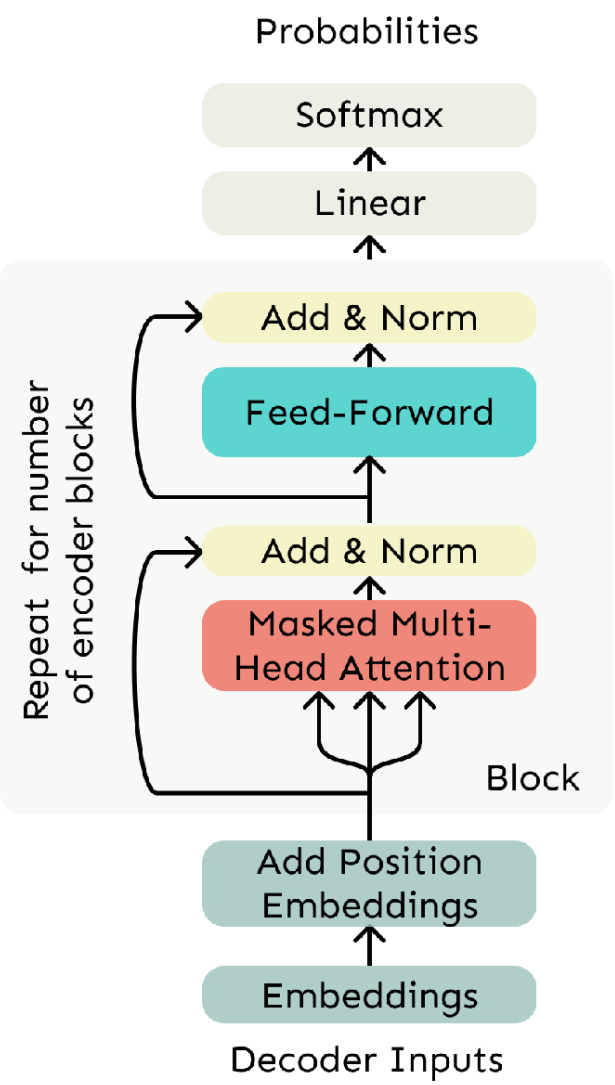
Solution: use multiple "attention heads," each specializing in different aspects. This specialization is learned automatically, no rules are set by the model designers.

https://web.stanford.edu/class/cs224n/

Multi-head Attention

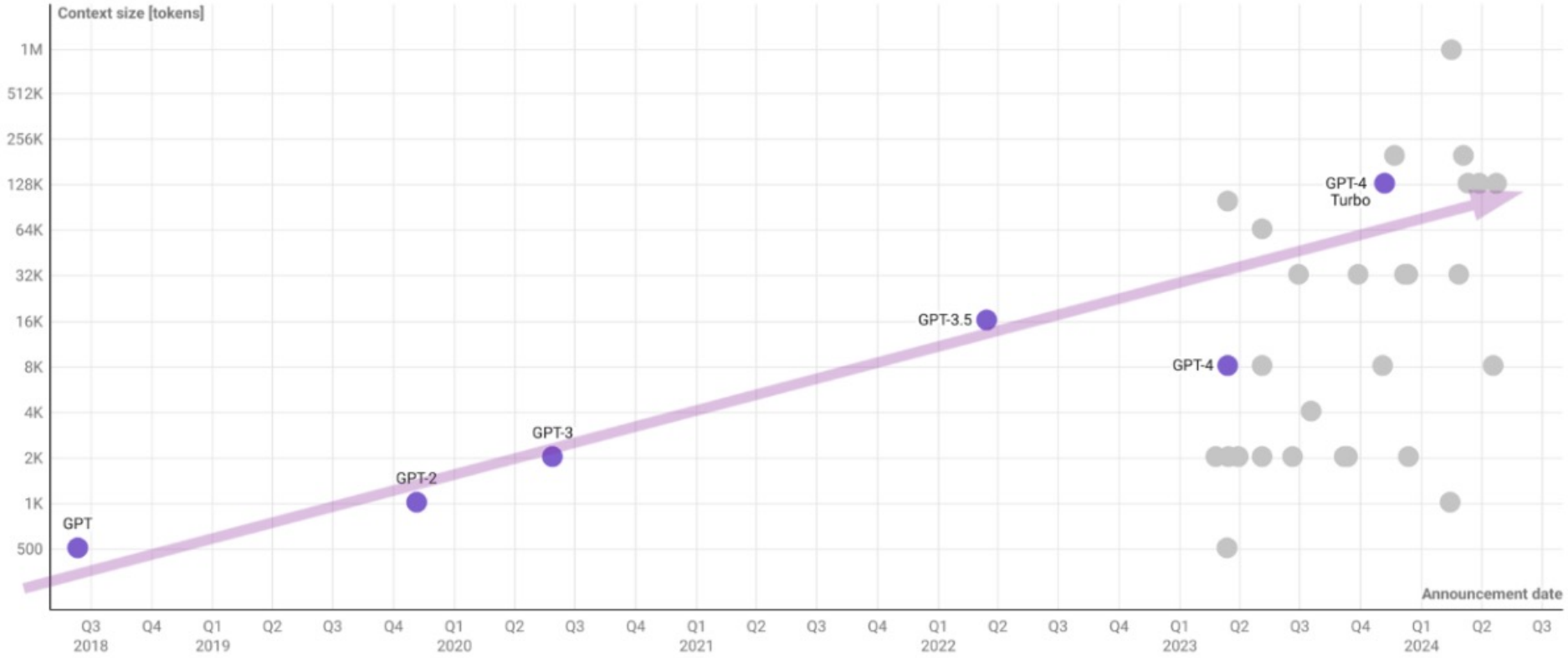


I went to Stanford CS 224n and learned



https://web.stanford.edu/class/cs224n/

Context Length



Transformers Recap

The Transformer architecture, introduced in 2017, dispensed with recurrence and convolutions entirely in favour of attention mechanisms.

This design is completely parallelizable and allows the model to choose which parts of a context window to "remember," making it more efficient to train than previous models.

The core of the Transformer is self-attention, which transforms word embeddings into queries, keys, and values to compute pairwise similarities and generate a weighted sum of values for each word.

Because self-attention has no inherent notion of word order, positional embeddings are added by rotating vectors based on their sequence position.

Additionally, Transformers use feed-forward neural networks after attention layers to model complex patterns, and they employ "multi-head attention" to allow different heads to specialize in various linguistic aspects automatically.

During training, masking is applied to ensure that the model does not "see" future words that have not yet been generated.

Tokenizer

How much does GPT-4 cost?

Written by Joshua J.
Updated over a week ago

We are excited to announce GPT-4 has a new pricing model, in which we have reduced the price of the prompt tokens.

For our models with 8k context lengths (e.g. `gpt-4` and `gpt-4-0314`), the price is:

- $0.03/1k prompt tokens
- $0.06/1k sampled tokens

For our models with 32k context lengths (e.g. `gpt-4-32k` and `gpt-4-32k-0314`), the price is:

- $0.06/1k prompt tokens
- $0.12/1k sampled tokens

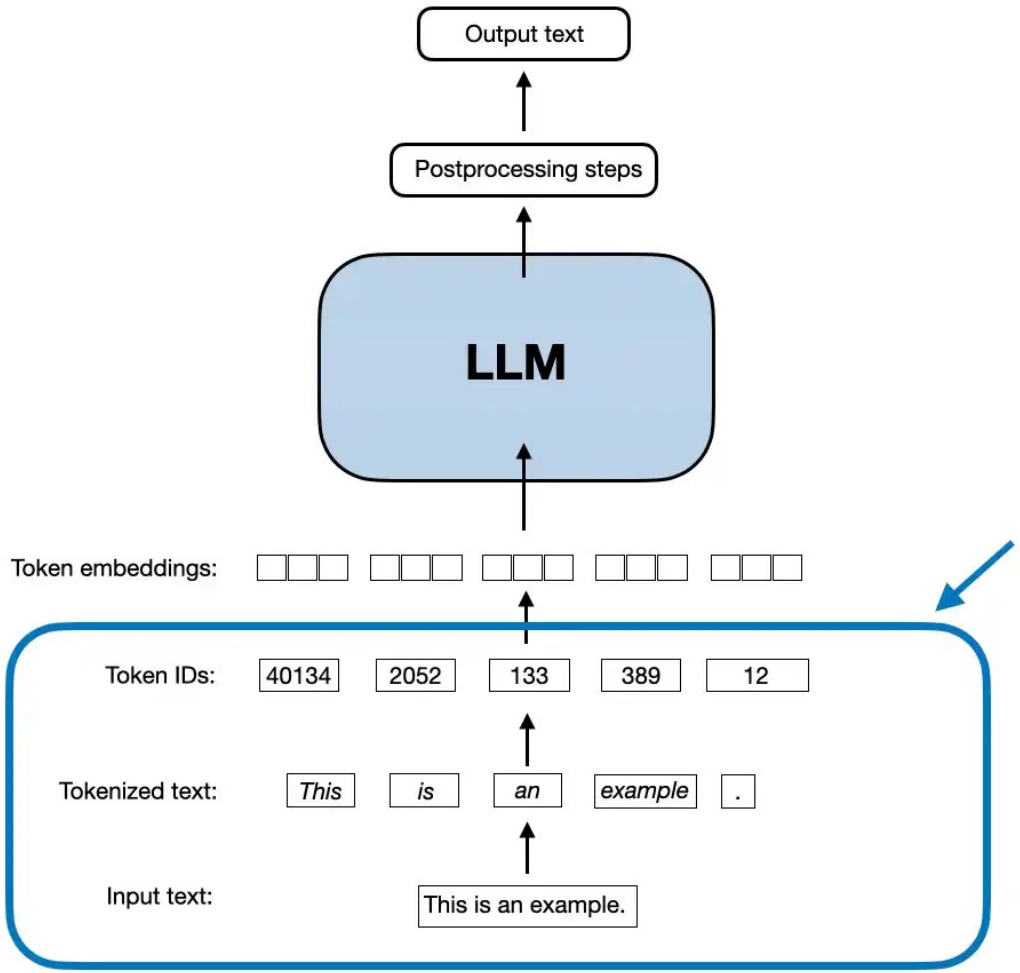
Tokenizer

Modern language models operate on **tokens**, not words.

A token is a piece of a word. The **tokenizer** is the software that breaks text into tokens and converts them into **token IDs**. These IDs are then used by the language model to create embeddings, which flow through the transformer layers.

Most tokenizers today are **deterministic** and rely on a **vocabulary**, a fixed set of tokens recognized in the input.

They also include special tokens to mark the start of text, unknown strings, and other functions.



Why not Characters?

Because attention has quadratic complexity, increasing the context window becomes very **computationally expensive**.

To manage this, it's better to split text into fewer elements.

Also, embedding a single character is challenging, since a character alone carries **no inherent meaning**.

Tokens	Characters
94	382

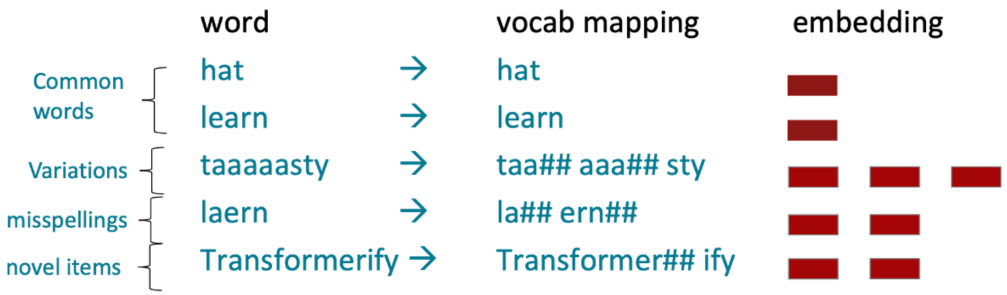
We're no strangers to love
You know the rules and so do I (do I)
A full commitment's what I'm thinking of
You wouldn't get this from any other guy
I just wanna tell you how I'm feeling
Gotta make you understand
Never gonna give you up
Never gonna let you down
Never gonna run around and desert you
Never gonna make you cry
Never gonna say goodbye
Never gonna tell a lie and hurt you

Text Token IDs

Why not Words?

Modern Large Language Models (LLMs) employ tokenization to resolve the out-of-vocabulary problem, as decomposing rare or novel strings into **frequent sub-units** ensures that every possible input has a mathematical representation.

Furthermore, this approach optimizes the computational trade-off between lexicon dimensionality and sequence length, facilitating a compact, **fixed-size vocabulary** that generalizes across diverse languages and morphological variations without the memory overhead of a word-level dictionary.



un + please + ed → unpleased
dis + please → displease
please + ing → pleasing

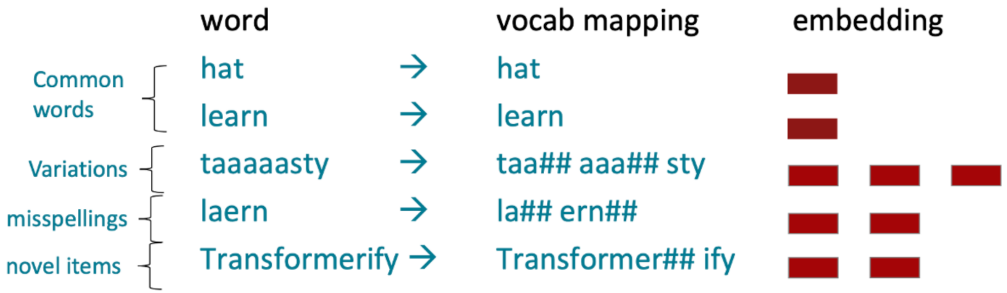
un + please + ant + ly → unpleasantly
please + ure → pleasure
please + ant → pleasant

Byte-Pair Encoding

Byte-Pair Encoding (BPE) is an **iterative tokenization algorithm** that optimizes a vocabulary by progressively merging the most frequent adjacent pairs of symbols into a single new unit.

Starting from a base set of individual characters, the process repeatedly scans the corpus to identify the **most common "byte pair"** and replaces all occurrences with a newly minted token until a predefined vocabulary size is reached.

This bottom-up approach enables the model to represent common words as single tokens while gracefully handling rare or unseen terms by decomposing them into their constituent subword components.



Byte-Pair Encoding Example

Dataset	Corpus = [‘hug’, ‘pug’, ‘pun’, ‘bun’, ‘hugs’]

Byte-Pair Encoding Example

Dataset	Corpus = [‘hug’, ‘pug’, ‘pun’, ‘bun’, ‘hugs’]
Iteration 1	Corpus = [(‘h’, ‘u’, ‘g’), (‘p’, ‘u’, ‘g’), (‘p’, ‘u’, ‘n’), (‘b’, ‘u’, ‘n’), (‘h’, ‘u’, ‘g’, ‘s’)] Vocabulary = [‘b’, ‘g’, ‘h’, ‘n’, ‘p’, ‘s’, ‘u’]

Byte-Pair Encoding Example

Dataset	Corpus = [‘hug’, ‘pug’, ‘pun’, ‘bun’, ‘hugs’]
Iteration 1	Corpus = [(‘h’, ‘u’, ‘g’), (‘p’, ‘u’, ‘g’), (‘p’, ‘u’, ‘n’), (‘b’, ‘u’, ‘n’), (‘h’, ‘u’, ‘g’, ‘s’)] Vocabulary = [‘b’, ‘g’, ‘h’, ‘n’, ‘p’, ‘s’, ‘u’]
Iteration 2	Corpus = [(‘h’, ‘**ug**’), (‘p’, ‘**ug**’), (‘p’, ‘u’, ‘n’), (‘b’, ‘u’, ‘n’), (‘h’, ‘**ug**’, ‘s’)] Vocabulary = [‘b’, ‘g’, ‘h’, ‘n’, ‘p’, ‘s’, ‘u’, ‘**ug**’]

Byte-Pair Encoding Example

Dataset	Corpus = [‘hug’, ‘pug’, ‘pun’, ‘bun’, ‘hugs’]
Iteration 1	Corpus = [(‘h’, ‘u’, ‘g’), (‘p’, ‘u’, ‘g’), (‘p’, ‘u’, ‘n’), (‘b’, ‘u’, ‘n’), (‘h’, ‘u’, ‘g’, ‘s’)] Vocabulary = [‘b’, ‘g’, ‘h’, ‘n’, ‘p’, ‘s’, ‘u’]
Iteration 2	Corpus = [(‘h’, ‘**ug**’), (‘p’, ‘**ug**’), (‘p’, ‘u’, ‘n’), (‘b’, ‘u’, ‘n’), (‘h’, ‘**ug**’, ‘s’)] Vocabulary = [‘b’, ‘g’, ‘h’, ‘n’, ‘p’, ‘s’, ‘u’, ‘**ug**’]
Iteration 3	Corpus = [(‘h’, ‘ug’), (‘p’, ‘ug’), (‘p’, ‘**un**’), (‘b’, ‘**un**’), (‘h’, ‘ug’, ‘s’)] Vocabulary = [‘b’, ‘g’, ‘h’, ‘n’, ‘p’, ‘s’, ‘u’, ‘ug’, ‘**un**’]

Byte-Pair Encoding Example

Dataset	Corpus = [‘hug’, ‘pug’, ‘pun’, ‘bun’, ‘hugs’]
Iteration 1	Corpus = [(‘h’, ‘u’, ‘g’), (‘p’, ‘u’, ‘g’), (‘p’, ‘u’, ‘n’), (‘b’, ‘u’, ‘n’), (‘h’, ‘u’, ‘g’, ‘s’)] Vocabulary = [‘b’, ‘g’, ‘h’, ‘n’, ‘p’, ‘s’, ‘u’]
Iteration 2	Corpus = [(‘h’, ‘**ug**’), (‘p’, ‘**ug**’), (‘p’, ‘u’, ‘n’), (‘b’, ‘u’, ‘n’), (‘h’, ‘**ug**’, ‘s’)] Vocabulary = [‘b’, ‘g’, ‘h’, ‘n’, ‘p’, ‘s’, ‘u’, ‘**ug**’]
Iteration 3	Corpus = [(‘h’, ‘ug’), (‘p’, ‘ug’), (‘p’, ‘**un**’), (‘b’, ‘**un**’), (‘h’, ‘ug’, ‘s’)] Vocabulary = [‘b’, ‘g’, ‘h’, ‘n’, ‘p’, ‘s’, ‘u’, ‘ug’, ‘**un**’]
Iteration 4	Corpus = [(‘**hug**’), (‘p’, ‘ug’), (‘p’, ‘un’), (‘b’, ‘un’), (‘**hug**’, ‘s’)] Vocabulary = [‘b’, ‘g’, ‘h’, ‘n’, ‘p’, ‘s’, ‘u’, ‘ug’, ‘un’, ‘**hug**’]

Discussion

Why can’t modern LLMs count the letters in a word?

Tokenizer Recap

Large Language Models (LLMs) operate on mathematics, meaning they can only process numbers, not the raw text of human language. To bridge this gap, text must be converted into a numerical format through a process called tokenization.

A tokenizer breaks down a sentence into manageable units called tokens, which are then assigned a unique ID number. Many modern tokenizers use a method called Byte-Pair Encoding (BPE), which creates a vocabulary by starting with individual characters and iteratively merging the most frequently occurring pairs. This results in an efficient system where common words like "apple" become a single token, while rarer or compound words like "tokenization" can be broken down into recognizable subword tokens like "token" and "ization".

This has a profound effect on the model's capabilities: because the LLM processes these pre-defined tokens rather than individual letters, it fundamentally lacks a character-level understanding of language. This is why a powerful model can write a sonnet but may fail at a seemingly simple task like counting the number of 'p's in "apple" or spelling a word backwards; it doesn't see the individual letters 'a', 'p', 'p', 'l', 'e', but only the single, indivisible token representing the entire word.

RLHF

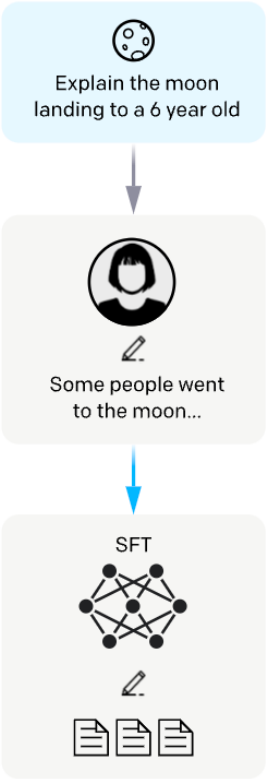
Step 1

Collect demonstration data, and train a supervised policy.

A prompt is sampled from our prompt dataset.

A labeler demonstrates the desired output behavior.

This data is used to fine-tune GPT-3 with supervised learning.



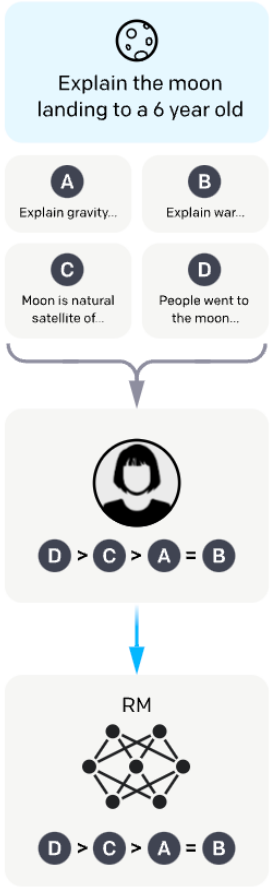
Step 2

Collect comparison data, and train a reward model.

A prompt and several model outputs are sampled.

A labeler ranks the outputs from best to worst.

This data is used to train our reward model.



Step 3

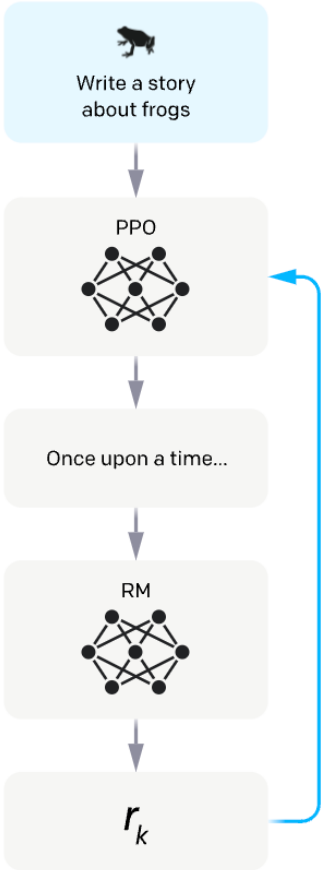
Optimize a policy against the reward model using reinforcement learning.

A new prompt is sampled from the dataset.

The policy generates an output.

The reward model calculates a reward for the output.

The reward is used to update the policy using PPO.



RLHF Recap

Reinforcement Learning with Human Feedback (RLHF) is the current method used to align generative models with human ethics and preferences. This process involves three key steps:

1. Initial fine-tuning: The generative model is fine-tuned on a curated set of questions and answers.
2. Human evaluation: The model responds to new questions, and human evaluators rate these responses. This data is used to train a separate model that can automatically assess responses.
3. Iterative improvement: The generative model answers fresh questions, while the evaluation model scores these answers. Through this process, the generative model is incentivized to produce increasingly higher-quality responses.

When both the generative and evaluation models are properly trained, the result is a system that learns to generate responses that align well with human preferences and values.

Building GPT

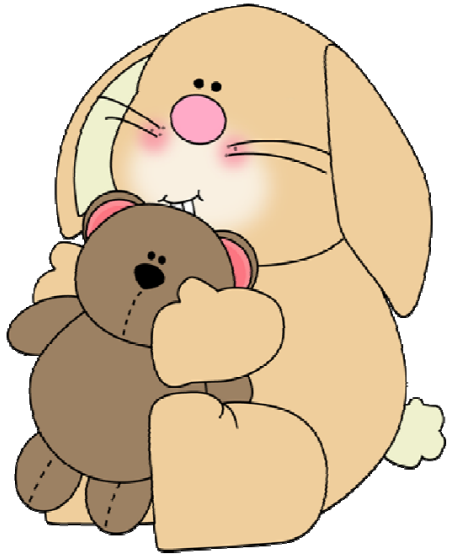
GPT Assistant training pipeline

Stage	Pretraining	Supervised Finetuning	Reward Modeling	Reinforcement Learning
Dataset	**Raw internet** text trillions of words low-quality, large quantity	**Demonstrations** Ideal Assistant responses, ~10-100K (prompt, response) written by contractors low quantity, high quality	**Comparisons** 100K –1M comparisons written by contractors low quantity, high quality	**Prompts** ~10K-100K prompts written by contractors low quantity, high quality
Algorithm	**Language modeling** predict the next token	**Language modeling** predict the next token	**Binary classification** predict rewards consistent w preferences	**Reinforcement Learning** generate tokens that maximize the reward
		init from	init from	init from SFT use RM
Model	**Base model**	**SFT model**	**RM model**	**RL model**
Notes	1000s of GPUs months of training ex: GPT, LLaMA, PaLM can deploy this model	1-100 GPUs days of training ex: Vicuna-13B can deploy this model	1-100 GPUs days of training	1-100 GPUs days of training ex: ChatGPT, Claude can deploy this model

Training Language Models

Read and fill in the missing words from the box.

bunny, only, parents, years, teddy bear, school, have got, garden, ~~name~~, birthday, nice



My _____*name*_____ is Lisa. I'm seven _____ old.

I go to _____ , my class is 2.b. I live in a house with my

_____ . I haven't got any brothers or sisters. I'm an

_____ child. My dad is really _____ and

my mum is great. We _____ a cat. It's name is Riki.

He likes to play in the _____ very much.

I have got a lot of toys, but I'd like to have a big _____

and a small _____ for my _____ .

____ / 10

GPT-3 vs InstructGPT

Prompt *Explain the moon landing to a 6 year old in a few sentences.*

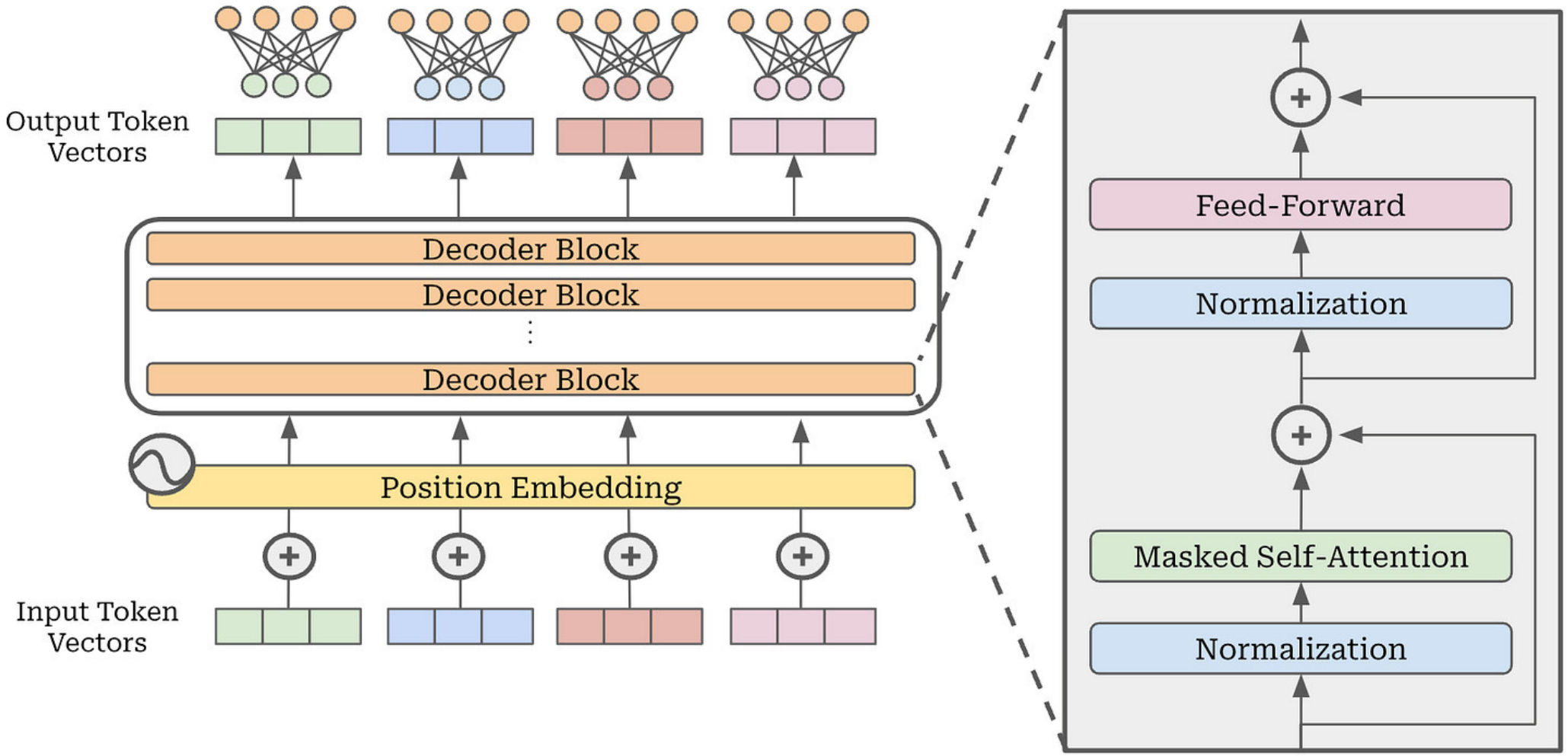
Completion GPT-3

```
Explain the theory of gravity to a 6 year old.

Explain the theory of relativity to a 6 year old in a few sentences.

Explain the big bang theory to a 6 year old.

Explain evolution to a 6 year old.
```

InstructGPT

```
People went to the moon, and they took pictures of what they saw, and sent them back to
the earth so we could all see them.
```

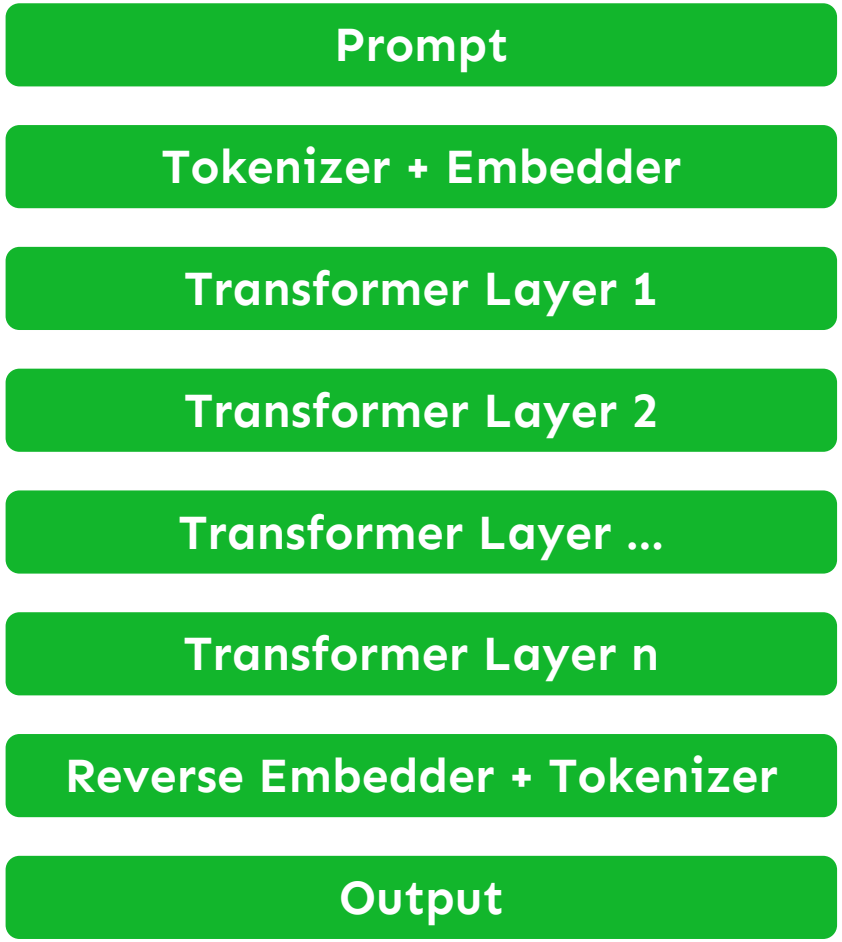
A Complete LLM



A Complete LLM

A quick breakdown of a **Transformer-based** LLM.

- **Tokenizer:** Raw text is decomposed into sub-word units (tokens) using algorithms like BPE.
- **Embedding:** Tokens are mapped to dense vectors and information about the position of each token is added.
- **Transformers**: Modules composed of attention and feed-forward modules that compute context-aware representations and apply non-linear transformations. Usually there are 10s of layers (96 in GPT-3).
- **De-tokenization:** The final hidden state is projected from the model dimension to the full vocabulary size (the number of possible tokens). A softmax converts the output logits into a probability distribution.
- **Reverse Tokenizer:** A sampling strategy over the token probability distribution (e.g., Top-P, Greedy) selects a token ID, which is then mapped back to human-readable text.



Inference

A large language model (LLM) is a computational model notable for its ability to achieve general-purpose language generation and other natural language processing tasks such as classification. Based on language models, **LLMs** acquire these abilities by learning statistical relationships from text documents during a computationally intensive self-supervised and semi-supervised training process.[1] LLMs can be used for text generation, a form of generative AI, by taking an input text and repeatedly predicting the next token or word.

Inference

A large language model (LLM) is a computational model notable for its ability to achieve general-purpose language generation and other natural language processing tasks such as classification. Based on language models, LLMs **acquire** these abilities by learning statistical relationships from text documents during a computationally intensive self-supervised and semi-supervised training process.[1] LLMs can be used for text generation, a form of generative AI, by taking an input text and repeatedly predicting the next token or word.

Inference

A large language model (LLM) is a computational model notable for its ability to achieve general-purpose language generation and other natural language processing tasks such as classification. Based on language models, LLMs acquire **these** abilities by learning statistical relationships from text documents during a computationally intensive self-supervised and semi-supervised training process.[1] LLMs can be used for text generation, a form of generative AI, by taking an input text and repeatedly predicting the next token or word.

GPT Training Recap

ChatGPT's training process consists of four main phases:

1. Pretraining: The model learns the fundamentals of language by analyzing vast amounts of text data. This creates a "base model" and is the most resource-intensive step, requiring thousands of GPUs and months of processing time.
2. Supervised Fine-tuning: The model is refined using high-quality examples of human-written responses to various prompts. While limited in quantity, these examples help shape the model's output quality.
3. Reward Modeling: A separate model is developed to assess the quality of responses generated by the main model. This creates a framework for evaluating output.
4. Reinforcement Learning from Human Feedback (RLHF): The main model is further trained to maximize its score according to the reward model. This final phase produces deployment-ready models like ChatGPT or Claude.

This process combines large-scale machine learning with targeted refinement to create AI models capable of engaging in human-like dialogue.

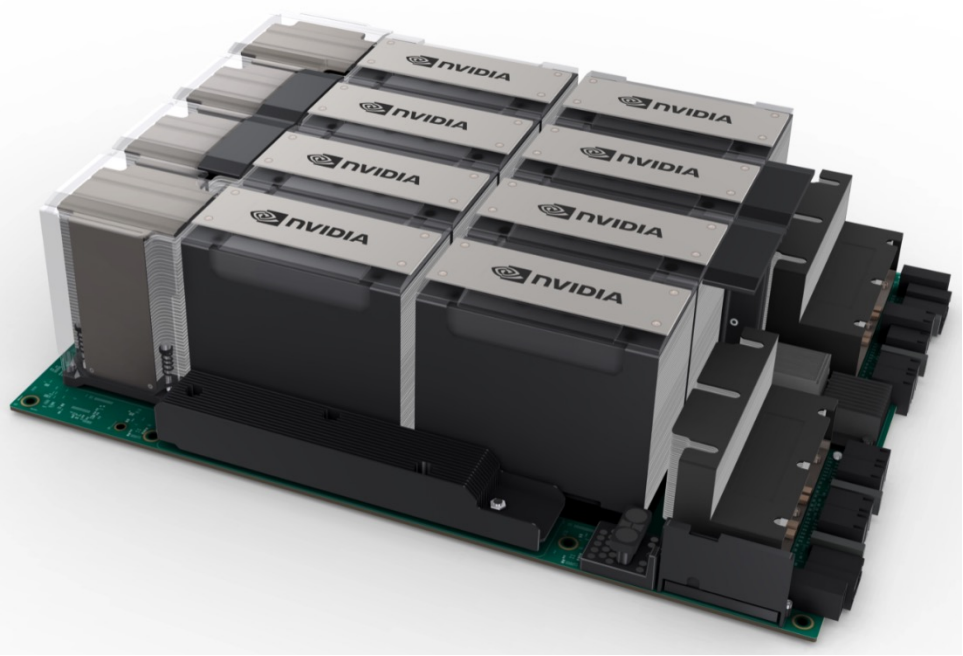
Running an LLM

Hardware

Running LLMs on a CPU is recommended only for experimentation and testing, and only with models smaller than 3B parameters.

For any realistic real-world application, **GPUs or TPUs are required**.

We will not cover GenAI-related hardware in this class, but GPUs are specialized chips designed to **efficiently perform massive parallel computations**, such as those needed to train or run LLMs.

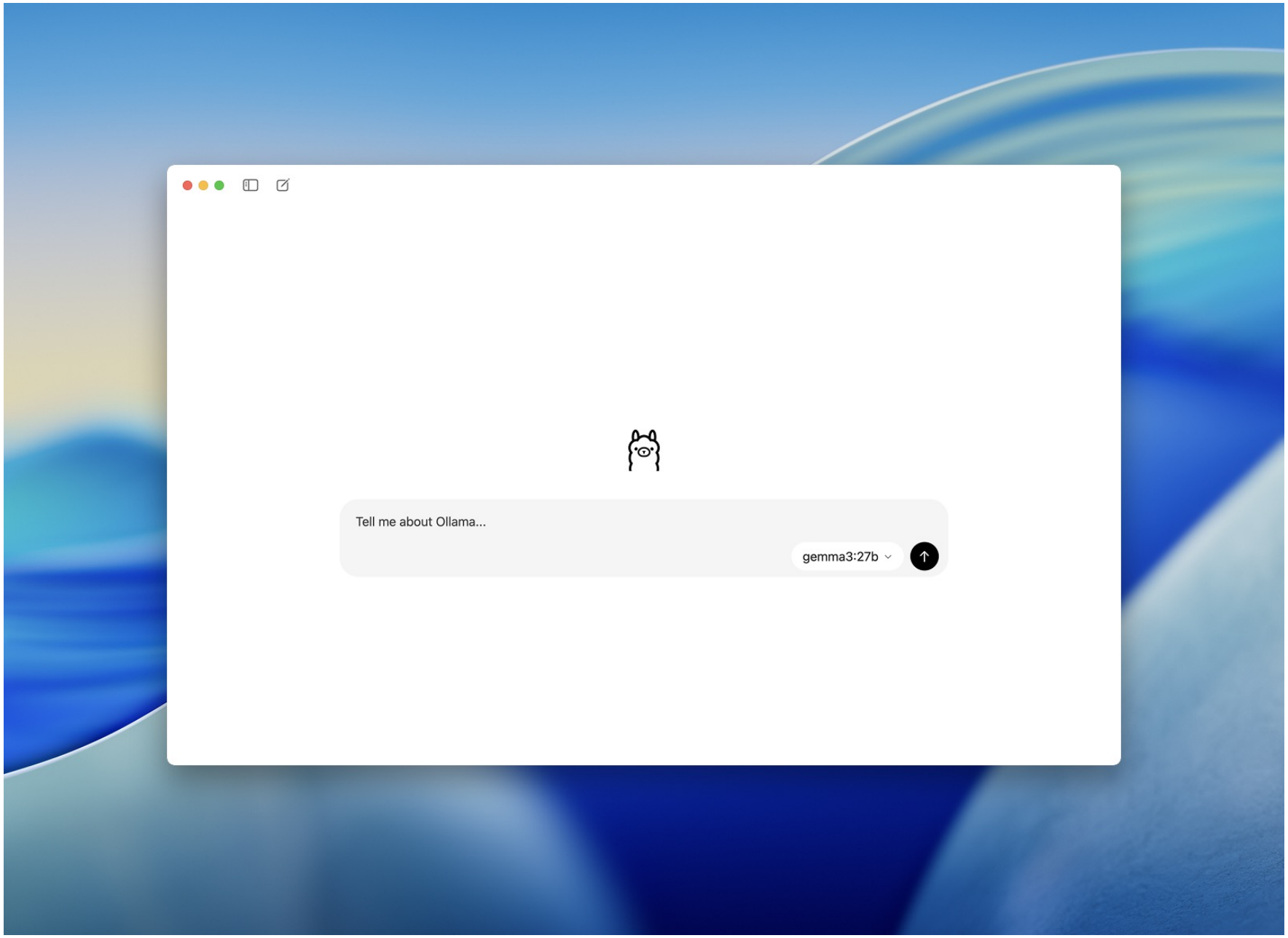


Hardware Requirements

Parameters	BF16 (16-bit)	SFP8 (8-bit)	Q4_0 (4-bit)
Gemma 3 270M (*text only*)	400 MB	297 MB	240 MB
Gemma 3 1B (*text only*)	1.5 GB	1.1 GB	892 MB
Gemma 3 4B	6.4 GB	4.4 GB	3.4 GB
Gemma 3 12B	20 GB	12.2 GB	8.7 GB
Gemma 3 27B	46.4 GB	29.1 GB	21 GB

Table 1. Approximate GPU or TPU memory required to load Gemma 3 models based on parameter count and quantization level.

https://ai.google.dev/gemma/docs/core


Tell me about Ollama...
gemma3:27b

Ollama

Ollama is an open-source **local runtime** that simplifies running Large Language Models (LLMs) on personal hardware. It packages model weights, configurations, and system prompts into a single, portable unit.

Ollama serves as the bridge between raw model files and your applications through three core mechanisms:

- Powered by **llama.cpp**, it uses highly optimized C++ to run models on both CPUs and GPUs (supporting Metal, CUDA, and ROCm).
- It utilizes quantization and special file formats to compress model weights (e.g., from 16-bit to 4-bit precision). This allows a model with billions of parameters to fit into consumer RAM.
- It runs a background service with a **Local REST API**. This allows any local application to communicate with the model using standard HTTP requests, effectively replacing cloud-based APIs like OpenAI.

Cloud-Based LLMs

Anthropic, OpenAI, and Google offer their flagship models (GPT, Claude, and Gemini) as services that can be accessed **through HTTP APIs**.

This means **no special hardware is required**: all computation and infrastructure management are handled by the provider.

In this class, we will use only cloud-based LLMs.

You are welcome to try Ollama on your laptop if you want to experiment on your own.

Generate text | Generate an image | Create vector embeddings

Create a human-like response to a prompt

```python
from openai import OpenAI
client = OpenAI()

completion = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {
            "role": "user",
            "content": "Write a haiku about recursion in programming."
        }
    ]
)

print(completion.choices[0].message)
```

Live Coding

- Using ChatGPT
- Interacting with OpenAI or Gemini models via API

colab

Open Notebook in Colab

The End