

Artificial intelligence

Text data and NLP

- cleaning with REGEX, scraping with BeautifulSoup
- tokenization and vectorization (BoW, TF-IDF,...)
- recurrent neural networks: LSTM with Keras
- Transformers: seq2seq models
- **Kaggle competition: binary text classification SPAM/NOT_SPAM**

Carl McBride Ellis

 carl.mcbride@u-tad.com

Recommended reading

- François Chollet "*Deep Learning with Python*" 2nd Edition (2021)
– chapter 11
- Aurélien Géron "*Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*", 3rd Edition O'Reilly Media (2022)
– chapter 16
- also the excellent [free pdf](#):

Speech and Language Processing

An Introduction to Natural Language Processing,
Computational Linguistics, and Speech Recognition

Third Edition draft

Daniel Jurafsky
Stanford University

James H. Martin
University of Colorado at Boulder

Natural language vs formal language

- Formal language: mathematics, chemical formulas, computer programs such as FORTRAN or python
- Natural language: Spanish, English, ...

Natural language has ambiguity, redundancy, ...

Remember: machines cannot *understand* text, they simply *work with* text data, with the aim of creating “useful” output.

Note: the terms understand (and intelligence) are under constant debate, and the meaning of ‘useful’ is highly subjective.

Typical tasks in NLP:

- “What’s the topic of this text?”: **text classification**
- “Does this text contain abuse?”: **content filtering**
- “Does this text sound positive or negative?”: **sentiment analysis**
- “What should be the next word in this incomplete sentence?”: **language modeling**
- “How would you say this in German?”: **translation**
- “How would you summarize this article in one paragraph?”: **summarization**

Which of these themes are supervised and which are unsupervised?

Unlike supervised machine learning, where we have *ground truth* values (regression) or labels (classification) often in NLP there is no ‘correct’ answer.

Even with supervised tasks the ground truth labels are often subject to uncertainty.

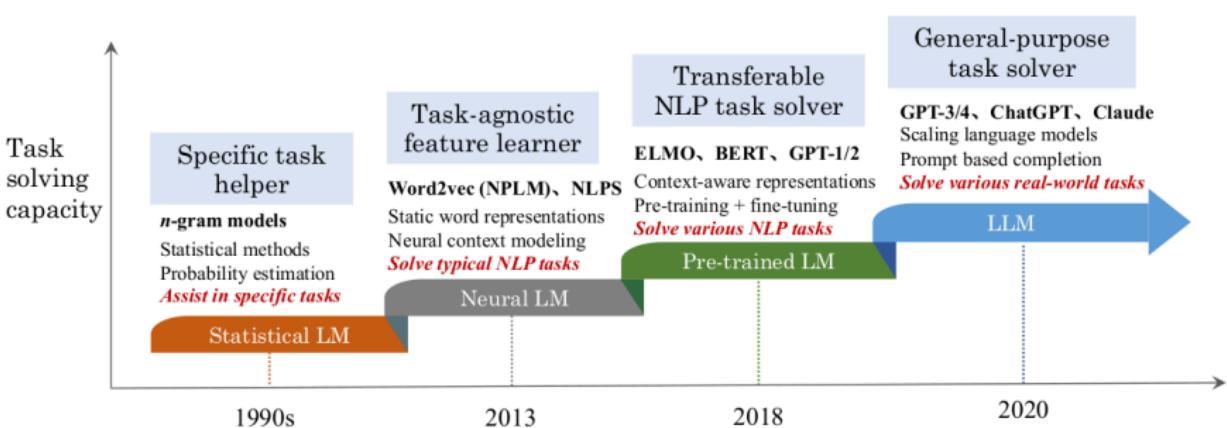
¿First NLP model?

Andrey Markov in 1913 studied 20,000 Russian letters from the book *Eugene Onegin* and used his model to calculate

- $p(\text{vowel}, \text{vowel}, \text{vowel})$
- $p(\text{vowel}, \text{consonant}, \text{vowel})$
- $p(\text{consonant}, \text{vowel}, \text{vowel})$
- $p(\text{consonant}, \text{consonant}, \text{vowel})$

(paper: "An Example of Statistical Investigation of the Text Eugene Onegin Concerning the Connection of Samples in Chains")

NLP timeline:



Text normalization (cleaning)

An example of text ‘normalization’ (*i.e.* cleaning) is converting everything to lowercase, or removing all punctuation and accents.

This can be done with regular expressions (regex) for example via the python `re` library:

```
import re
```

Let us take a look at

[notebook_NLP_REGEX.ipynb](#)

(Note that Keras, *i.e.* François Chollet, uses the term ‘standardization’ rather than normalization)

Practical session

1. pull a basic web page and parse it into plain text using REGEX

```
import urllib.request

url = 'https://ai.stanford.edu/~amaas/data/sentiment/'
page = urllib.request.urlopen(url).read()
print(page)
```

(just for fun, also take a quick look at the source of url = 'https://elpais.com')

2. quiz: [RegexOne: Learn Regular Expressions with simple, interactive exercises](#)

3. now do the same again, but this time starting from the page scraped using [Beautiful Soup](#)

```
from bs4 import BeautifulSoup

soup = BeautifulSoup(page)
soup.get_text(strip=True)
```

The use of regex goes all the way back to 1968.

Some people, when confronted with a problem, think
“I know, I’ll use regular expressions.”
Now they have two problems.

Jamie Zawinski (1997)

Useful tool: [Online interactive REGEX tester](#)

REGEX can even be used to [find prime numbers!](#)

```
import re
import sys

n=13
if (len(sys.argv)>1):
    n=int(sys.argv[1])

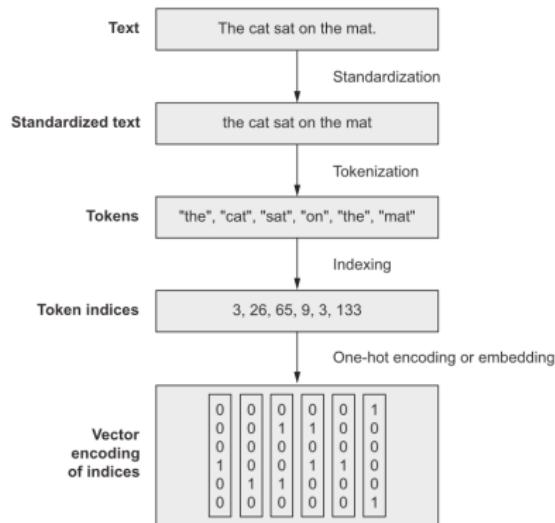
matches=re.match(r'^.?$|^(..+?)\1+$', '1'*n)

if matches:
    print(f"{n} is not a prime")
else:
    print(f"{n} is a prime")
```



Tokenization and vectorization

Machine learning *only* works with numbers. Getting the text ready for a neural network model: split text into ‘tokens’ and then ‘vectorize’



Tokenization is the fancy name for splitting up our cleaned text:

- character-level tokenization (not really used any more)
- word-level tokenization (unigrams)
- N -gram tokenization

(Note: token 0 is often reserved for padding (which can then be ignored using `mask_zero=True`), and token 1 is reserved for 'unknown')

Two types of models:

- bag-of-words models: no word order at all
- sequence models: word order is important

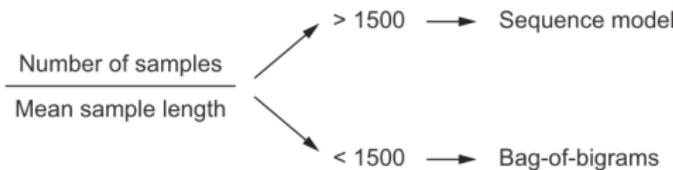


Figure 11.11 A simple heuristic for selecting a text-classification model: the ratio between the number of training samples and the mean number of words per sample

Bag-of-words model

N -grams are groups of up to N consecutive words.

For example, the 2-gram of the phrase “*the cat sat on the mat*” is

```
{"the", "the cat", "cat", "cat sat", "sat", "sat on", "on", "on the", "the mat", "mat"}
```

In a large piece of text there will be many N -grams. It is usual to just use, for example, the 30,000 most popular and discard the rest.

In NLP one can improve performance by first removing common words of little value. These are known as **stop words**.

For example

- articles: a, an, the
- conjunctions: and, but, or
- prepositions: in, on, at, with
- pronouns: he, she, it, they
- common verbs: is, am, are, was, were, be, being, been

For example, one can filter out stop words using those from the [Natural Language Toolkit \(NLTK\)](#)

```
import nltk
from nltk.corpus import stopwords

nltk.download('stopwords')
```

Nomenclature:

tabular	vision	NLP
dataset	album(?)	corpus
row	image	document

A 'document' could be a single sentence, a tweet, a movie review....

The Count Vectorizer creates vectors based on term frequency (TF)

```
corpus = ["Hello World!",
          "That was my Hello World",
          "This now has world + another hello world",
          "Spaceship world"
      ]
```

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()
count_matrix = vectorizer.fit_transform(corpus)
count_array = count_matrix.toarray()
pd.DataFrame(data = count_array,
              columns = vectorizer.get_feature_names_out())
```

	another	has	hello	my	now	spaceship	that	this	was	world
0	0	0	1	0	0		0	0	0	1
1	0	0	1	1	0		0	1	0	1
2	1	1	1	0	1		0	0	1	0
3	0	0	0	0	0		1	0	0	1

TF-IDF weighting

TF-IDF = term frequency - inverse document frequency

it balances term frequency in the document (row) with the frequency in the corpus (dataset)

```
tf-idf(t, d) = tf(t, d) * log [ n / df(t) ] + 1
```

thus common words are now given less weight than rare words

```
corpus = ["Hello World!",
          "That was my Hello World",
          "This now has world + another hello world",
          "Spaceship world"
      ]
```

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(smooth_idf=False, norm=None)

count_matrix = vectorizer.fit_transform(corpus)
count_array = count_matrix.toarray()
pd.DataFrame(data = count_array,
              columns = vectorizer.get_feature_names_out())
```

	another	has	hello	my	now	spaceship	that	this	was	world
0	0.000000	0.000000	1.287682	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.0
1	0.000000	0.000000	1.287682	2.386294	0.000000	0.000000	2.386294	0.000000	2.386294	1.0
2	2.386294	2.386294	1.287682	0.000000	2.386294	0.000000	0.000000	2.386294	0.000000	2.0
3	0.000000	0.000000	0.000000	0.000000	0.000000	2.386294	0.000000	0.000000	0.000000	1.0

For example, the term *spaceship* occurs only once in the document, and there are 4 documents in the corpus

$$\text{TF} \times \text{IDF} = 1 \times \left(\ln \left(\frac{4}{1} \right) + 1 \right) = 2.386294$$

Using the scikit-learn `TfidfVectorizer`:

```
vectorizer = TfidfVectorizer(ngram_range=(1, 2),
                             tokenizer=lambda x: re.findall(r'[\w]+', x),
                             strip_accents='unicode',
                             max_features=20_000,
                             )
X = vectorizer.fit_transform(all_text)
```

(Note: the `TfidfVectorizer` is equivalent to `CountVectorizer` followed by `TfidfTransformer`.)

Using Keras `TextVectorization` layer in a NN

```
text_vectorization = keras.layers.TextVectorization(  
    ngrams=2,  
    max_tokens=20_000,  
    output_mode="tf_idf",  
)
```

Let us take a look at examples of vectorization in:

[notebook_NLP_vectorization.ipynb](#)

Practical session

Perform a ‘bag-of-words’ classification

[`notebook_NLP_BoW_classifier.ipynb`](#)

Things to do:

- try different N -gram ranges
- try removing stop words
- try out the `TfidfVectorizer`

Insert your results (metrics) into the table at the end of the notebook

Word embeddings

One-hot encoding assumes all individual word vectors are orthogonal. This is a high dimensional sparse representation: if we have 20,000 N -grams, we have a 20,000 dimensional space.

However, words are not orthogonal; the words 'car' and 'engine' do have a semantic relationship.

Our task is to encode a semantic relationship into a geometric relationship

We can learn the embedding for our dataset using an [embedding layer](#)

```
embedded = layers.Embedding(input_dim=max_tokens,  
                             output_dim=256,  
                             mask_zero=True,)
```

This may be good if we have a specific task or type of dataset.

or we can use pre-trained word embeddings trained on huge datasets, for example

- Linguistic Regularities in Continuous Space Word Representations (2013)
- Word2vec (2013) - Continuous Bag-of-Words (CBoW) model
- GloVe (Global Vectors for Word Representation) (2014)
- fastText (2016) ([paper](#))

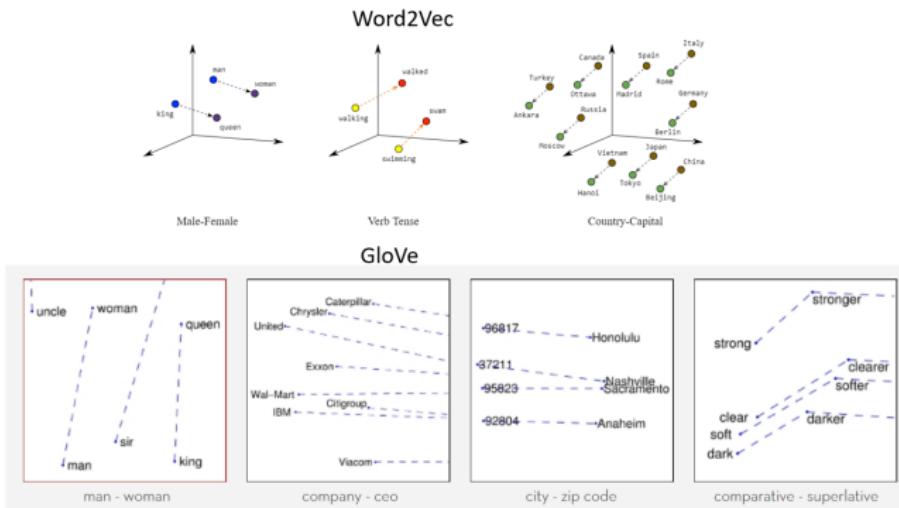


Table 8: Examples of the word pair relationships, using the best word vectors from Table 4 (Skip-gram model trained on 783M words with 300 dimensionality).

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

Cosine similarity

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

where \mathbf{A} and \mathbf{B} are two (word) vectors.

Using numpy

```
np.dot(A,B)/(np.linalg.norm(A)*np.linalg.norm(B))
```

The distance between two vectors can be calculated using the (Frobenius) norm:

```
np.linalg.norm(A-B)
```

spaCy converts each individual word (token) into a 300 dimensional vector

```
nlp("King").vector[:3] # show first three components  
array([ 0.31542, -0.35068,  0.42923], dtype=float32)  
  
nlp("Queen").vector[:3]  
array([ 0.4095 , -0.22693,  0.25362], dtype=float32)
```

each document also becomes a 300 dimensional vector, composed of the average of each of the word components

```
document = "King Queen"  
nlp(document).vector[:3]  
array([ 0.36246002, -0.288805 ,  0.341425 ], dtype=float32)
```

Practical session

Classification using embeddings saved in `en_core_web_md`

`notebook_NLP_embedding.ipynb`

with the help of `spaCy` ("*Industrial-Strength Natural Language Processing in Python*")

Sequence models

Recurrent neural networks (RNN): The LSTM and the bidirectional LSTM

The word 'recurrent' comes from the work of [Santiago Ramón y Cajal](#) who won the [Nobel Prize in 1906](#).

RNNs were originally developed for time series problems

Keras has three main types of recurrent layers: [SimpleRNN](#), [LSTM](#), and [GRU](#)

The output of an RNN is a function of the current input **and** the previous input (state)

ANN:

$$\hat{y} = a(\mathbf{w} \cdot \mathbf{x} + b)$$

RNN:

$$\hat{y} = a(\mathbf{w}_t \cdot \mathbf{x}_t + \mathbf{w}_{t-1} \cdot \mathbf{x}_{t-1} + b)$$

where a is our activation function (*i.e.* ReLU *etc.*)

The simple RNN architecture

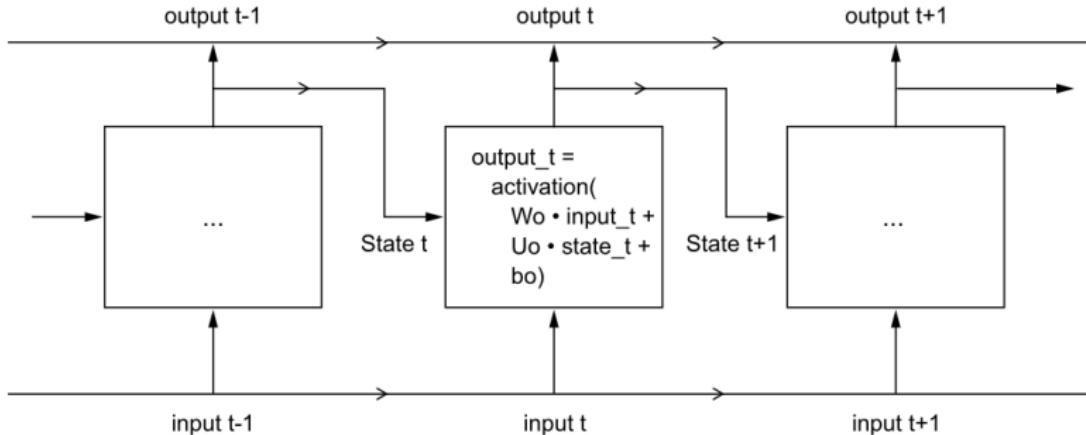


Figure 10.7 A simple RNN, unrolled over time

However, it was found empirically that the simple RNN architecture is not particularly effective; it cannot learn very far back in time.

(Paper: "Learning long-term dependencies with gradient descent is difficult" (1994))

The LSTM architecture (1997)

(Paper: "Long Short-Term Memory" (1997))

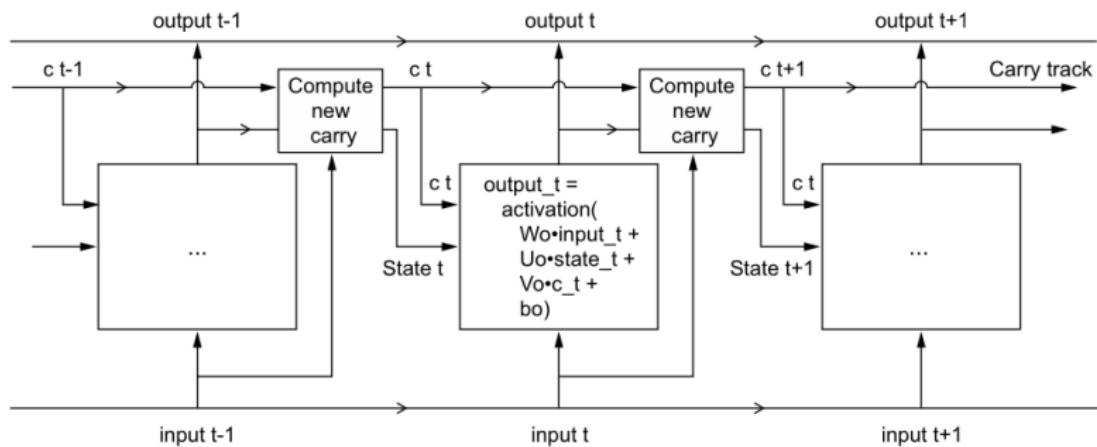


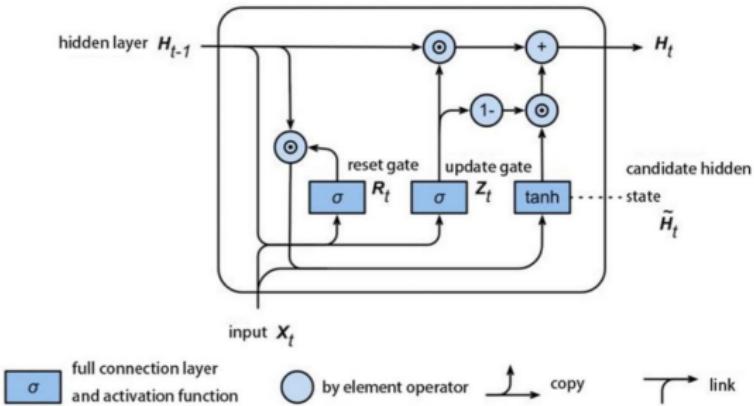
Figure 10.10 Anatomy of an LSTM

(see also: "xLSTM: Extended Long Short-Term Memory" (2024))

A basic LSTM model implemented in Keras

```
model = Sequential()
model.add(Input(shape=(look_back, n_features)))
model.add(LSTM(n_neurons))
model.add(Dense(1))
```

Another architecture is the **Gated Recurrent Units (GRU)**, which has similar performance to the LSTM architecture.



(Paper: "On the Properties of Neural Machine Translation: Encoder–Decoder Approaches" (2014))

Practical session

We shall take a look at an LSTM model

LSTM time series prediction: sine wave example

(notebook hosted on Kaggle)

To do:

- try replacing the sine wave function for a sawtooth function:

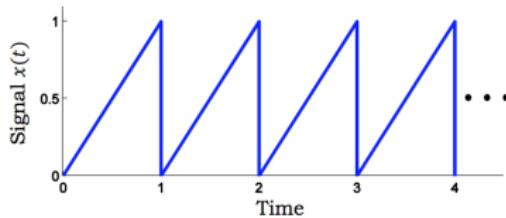
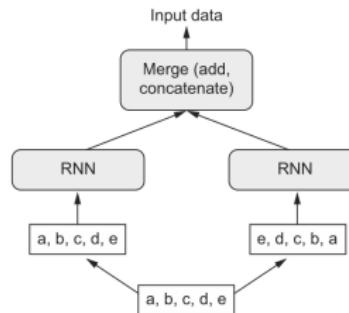


Figure 1: Sawtooth wave

For obvious reasons the LSTM was designed to work chronologically.

However, when it comes to NLP, it can be beneficial to also work antichronologically, and combine the two.

Hence the 'bidirectional' LSTM.



A basic bidirectional LSTM model

```
model = Sequential([
    Input(shape=(5, 10)),
    Bidirectional(LSTM(10, return_sequences=True),
    Bidirectional(LSTM(10)),
    Dense(5, activation="softmax"),
    ])

model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop')
```

Practical session

Use a bidirectional LSTM + own word embeddings to perform binary classification on the [ackImdb Movie Review dataset](#)

[notebook_NLP_LSTM_bidirectional](#)

(This dataset+notebook is hosted on Kaggle)

Kaggle competition #2

Binary text classification

[U-Tad] SPAM/NOT SPAM 2025 edition

For your coursework assignment you should upload a notebook to BlackBoard. PLEASE use the following name for your notebook *before* you upload to Blackboard (*Do not rename in Blackboard!*):

- `NLP_Your_Full_Name.ipynb`

Note: Please use the underscore (_) as the separator and **not blank spaces** (i.e. use the `snake_case` naming convention).

PLEASE also include your name WITHIN the notebook (at the top)

Transformers (SoTA)

The Transformer architecture was originally developed for machine translation. For example, translating a sentence (*i.e.* a sequence of words) in English into a sentence (*i.e.* sequence) in Spanish.

Hence a transformer is known as a *sequence-to-sequence* (seq2seq) model

Blog: [Visualizing A Neural Machine Translation Model \(Mechanics of Seq2seq Models With Attention\)](#) by Jay Alammar

The 2017 paper "*Attention Is All You Need*" with the [Tensor2Tensor library](#) by Google led to a revolution in NLP, dispensing with any recurrent or convolutional layers

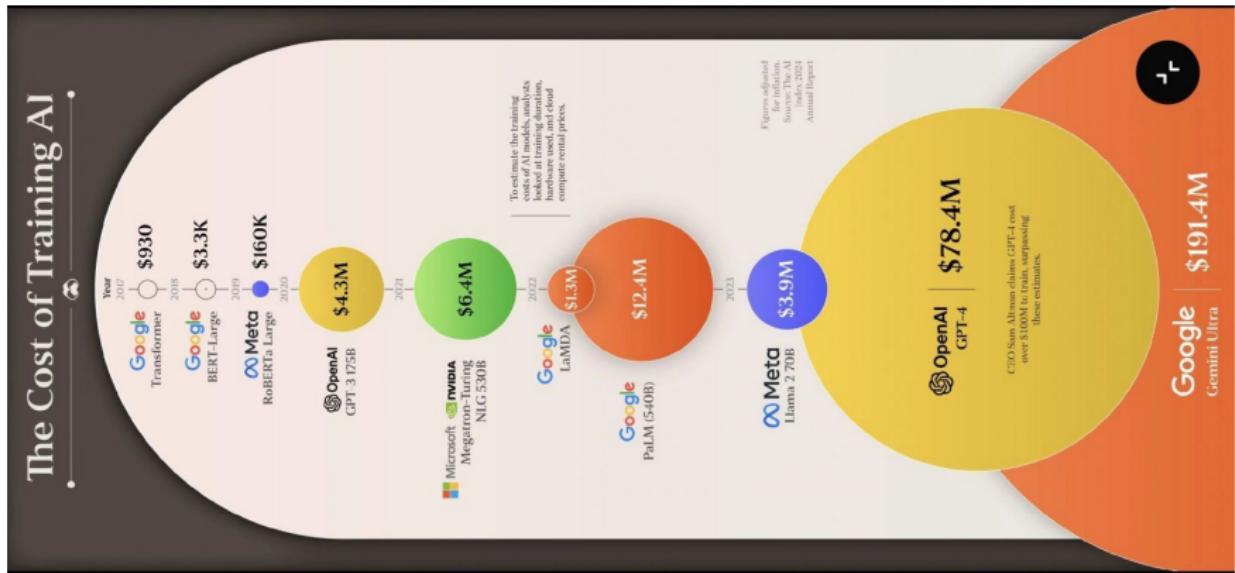
- [ELMo](#) (Embeddings from Language Models) (2018)
 - word representations are functions of the entire input sentence
- [ULMFiT](#) (Universal Language Model Fine-Tuning) (2018)
- [BERT](#) (Bidirectional Encoder Representations from Transformers) (2018)
- [GPT-1](#) (Generative Pre-trained Transformer 1) (2018)

Also noteworthy is **PaLM** (Pathways Language Model) (2022), which has 540 billion parameters, and was trained on 6144 TPU v4 chips for 1200 hours and then 3072 TPU v4 chips for 336 hours.

To put that in context, on [Kaggle we have 30 hours per week access to one single TPU v3.8](#), so it would take us more than **5,388 years** to train PaLM ourselves.

GPT-4 is estimated to have around 1760 billion parameters.

LLM models are *very* expensive to train



We now have 'contextual embeddings'

- word embeddings (Word2vec, GloVe) use a global-level embedding (the vectors are calculated from a huge corpus of text)
- contextual embeddings (ELMo, BERT) use sequence-level semantics

Why do contextual embeddings work better?

For example, in English the word 'set' has well over 400 different meanings.

A train set has noting to do with a set in tennis. Thus for 'set' a global vector is not very useful, whereas a local meaning is.



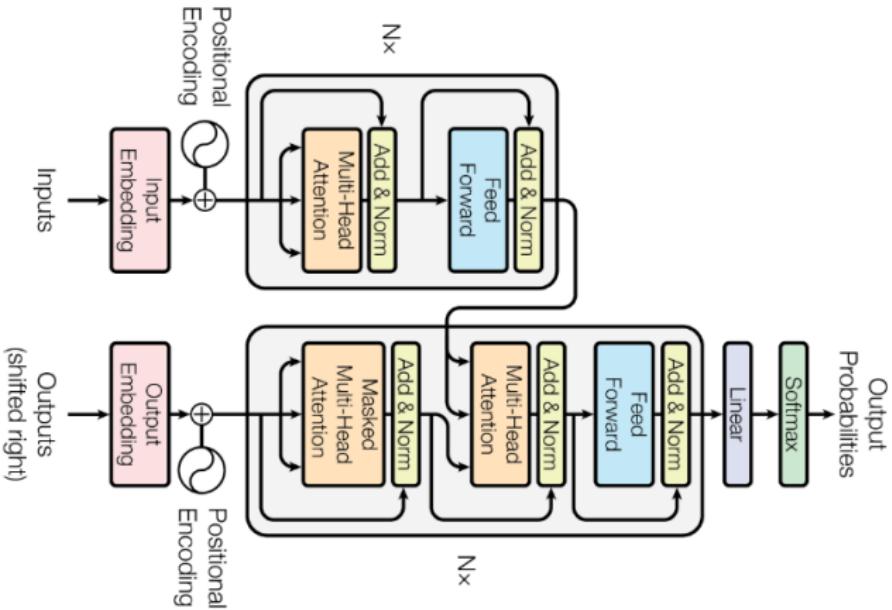
Attention leads to context-aware token representations; It adjusts the original global-level embedding based on the local context.

Watching
a model train



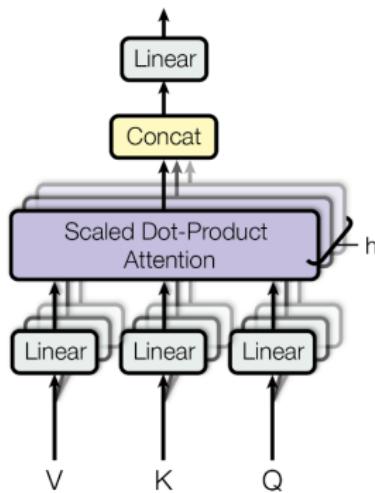
Watching
a model train





The Transformer architecture

Multi-Head Attention



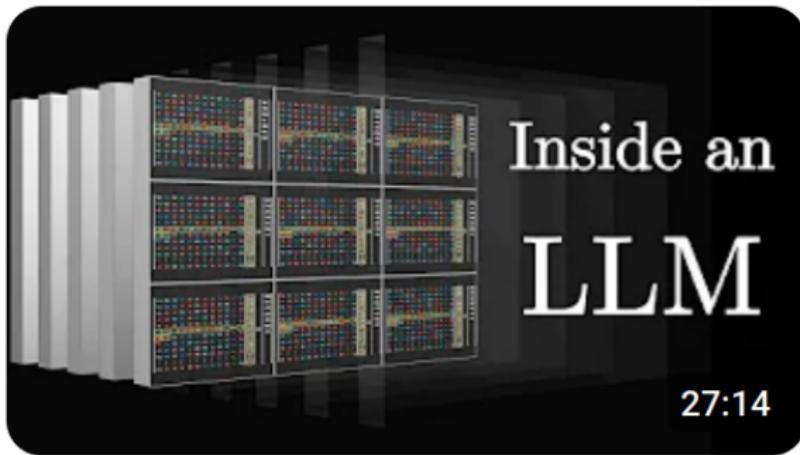
Value (V), Key (K), and Query (Q)

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

See the animation [transformer.gif](#) on BlackBoard

the website "[LLM Visualization](#)" by Brendan Bycroft

the blog post "[The Illustrated Transformer](#)" by Jay Alammar



Video: "[Transformers, the tech behind LLMs](#)"

- self-attention: for generating text
- cross-attention: for translation

Most LLM transformer models can be downloaded from [Hugging Face](#), which currently hosts over 1 million models



Hugging Face

as well as via [Kaggle models](#), most notably [Gemma 2](#)

(some) NLP performance metrics

- Translation: BLEU measures n-gram overlap between the translation and references
- Text summarization: ROUGE measures n-gram overlap between the generated and reference summaries.
- Text generation: perplexity
- Question Answering: Exact Match (EM)
- Text similarity: cosine similarity
- GLUE
- $i^{\text{e-score}}$?

The *ultimate* LLM performance benchmark?

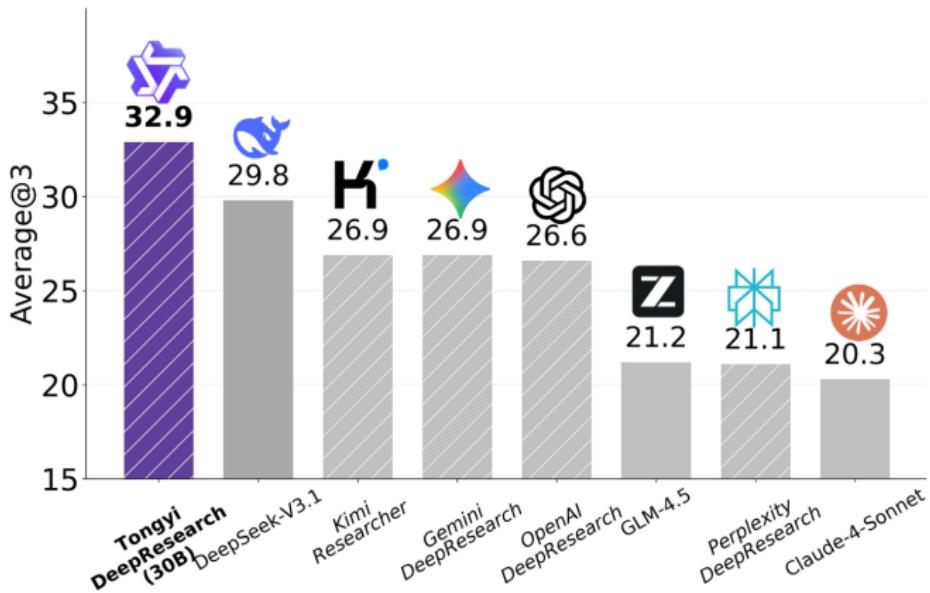
Humanity's Last Exam

“...consists of 2,500 questions across dozens of subjects, including mathematics, humanities, and the natural sciences. Each question has a known solution that is unambiguous and easily verifiable, but cannot be quickly answered via internet retrieval.”

[arXiv paper](#)

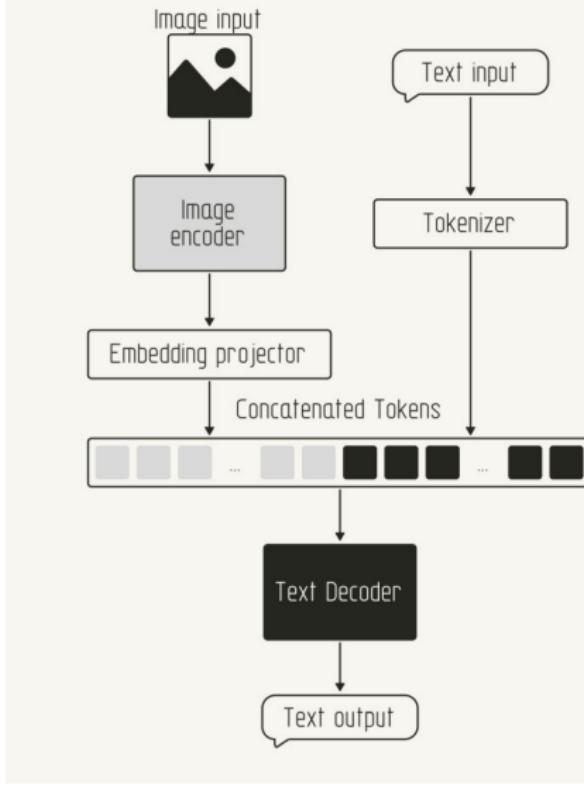
as of September 2025:

Humanity's Last Exam



Vision Language Model

A typical architecture



Practical session

Notebook: character by character text generation; it calculates the probability distribution of the next character in the sequence given a sequence of previous characters.

Text generation with an RNN

(This notebook is hosted on Kaggle)

Things to do:

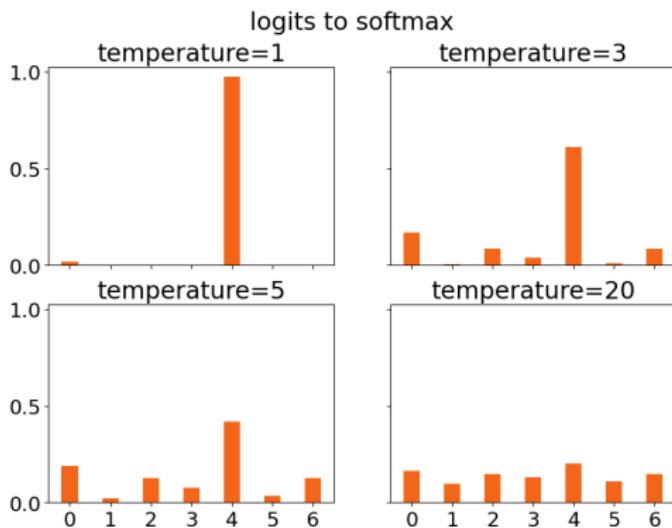
- explore the effect of the temperature parameter
- explore different source texts, for example creating a corpus from several books by the same author

(Suggested reading: "[The Unreasonable Effectiveness of Recurrent Neural Networks](#)" by Andrej Karpathy)

From logits to softmax with temperature:

```
softmax = np.exp(logits/temperature) /  
          np.sum(np.exp(logits/temperature))
```

The greater the temperature, the smoother the PDF



Sources of NLP training data

Books: Project Gutenberg has the raw txt of over 70,000 free books

For example to use the text of "*Alice's Adventures in Wonderland*"

```
url = 'https://www.gutenberg.org/cache/epub/11/pg11.txt'
path = keras.utils.get_file(origin=url)

with io.open(path, encoding="utf-8") as f:
    text = f.read().lower()

text = text.replace("\n", " ")

print("Corpus length:", len(text))
```

or [Wikipedia datasets](#) (20220301.simple is about 235MB)

```
from datasets import load_dataset
dataset = load_dataset("wikipedia", "20220301.simple",
                      split='train')
n_rows = 500 # use just 500 Wikipedia pages
text = ''.join(dataset['text'][:n_rows])
text = text.lower()
text = text.replace("\n", " ")
del dataset
```

See also: "[Datasets for Large Language Models: A Comprehensive Survey](#)" (February 2024) + [Awesome-LLMs-Datasets](#)

- A Survey of the Usages of Deep Learning in Natural Language Processing (December 2019)
- Large Language Models: A Survey (February 2024)
- A Survey of Large Language Models (October 2024)
- A Comprehensive Overview of Large Language Models (October 2024)