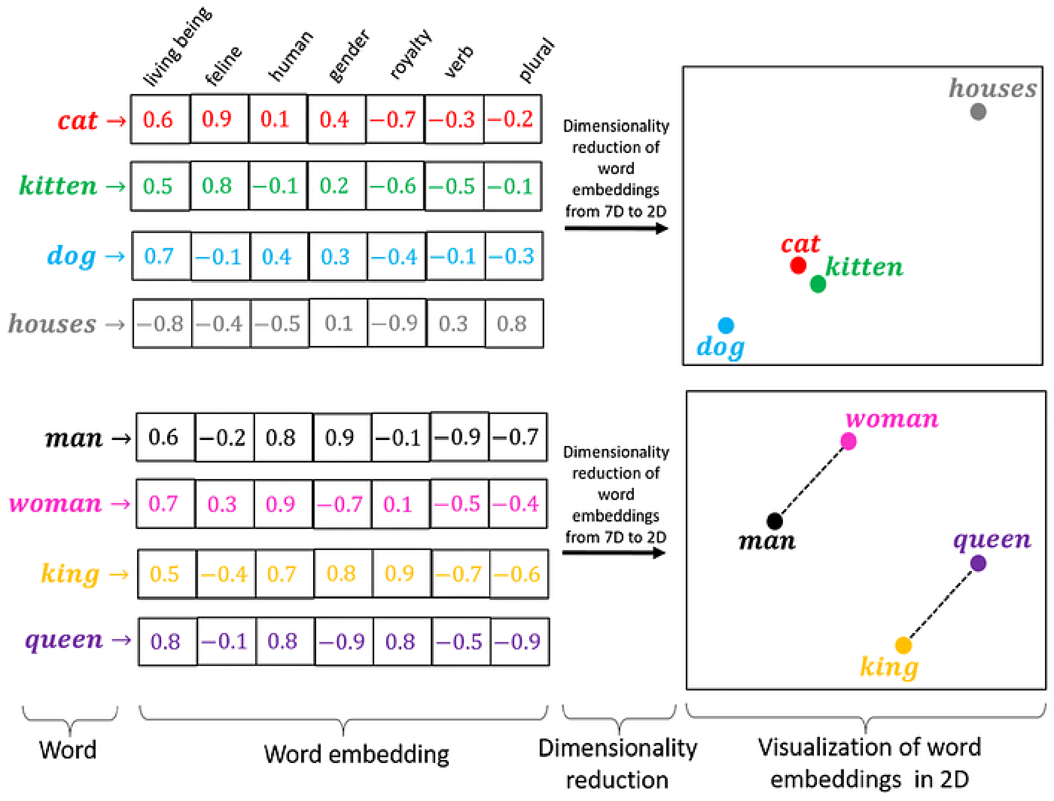
307307
Part 2 – Introduction to Large Language Models

Preface

- In the previous section, we used Bow and TDIDF to convert documents and words into numerical representations.
- These representations were simple, they had no semantics for words, just on/off switches for existing and non-existing words in a document.
- In this part of the course, we want to convert words into meaningful list of numbers.
- These numbers are called **Word Embeddings**.
- We will use Neural Networks to create these Word Embeddings.



	living being	feline	human	gender	royalty	verb	plural
cat →	0.6	0.9	0.1	0.4	−0.7	−0.3	−0.2
kitten →	0.5	0.8	−0.1	0.2	−0.6	−0.5	−0.1
dog →	0.7	−0.1	0.4	0.3	−0.4	−0.1	−0.3
houses →	−0.8	−0.4	−0.5	0.1	−0.9	0.3	0.8

man →	0.6	−0.2	0.8	0.9	−0.1	−0.9	−0.7
woman →	0.7	0.3	0.9	−0.7	0.1	−0.5	−0.4
king →	0.5	−0.4	0.7	0.8	0.9	−0.7	−0.6
queen →	0.8	−0.1	0.8	−0.9	0.8	−0.5	−0.9

Dimensionality reduction of word embeddings from 7D to 2D
houses
cat
kitten
dog
Dimensionality reduction of word embeddings from 7D to 2D
woman
man
queen
king
Word
Word embedding
Dimensionality reduction
Visualization of word embeddings in 2D

Introduction to Neural Networks

The Perceptron

Outcomes

- Fundamentals of neural networks
- Evolution from single perceptrons to MLPs
- Detailed MLP architecture (input, hidden, and output layers)
- Mathematical representations
- Various activation functions (Sigmoid, ReLU, etc.)
- Backpropagation and training methodologies
- Loss functions and optimization techniques
- Architecture design considerations
- Real-world applications
- Advantages and limitations
- Modern MLP variants and implementations

History of Neural Networks

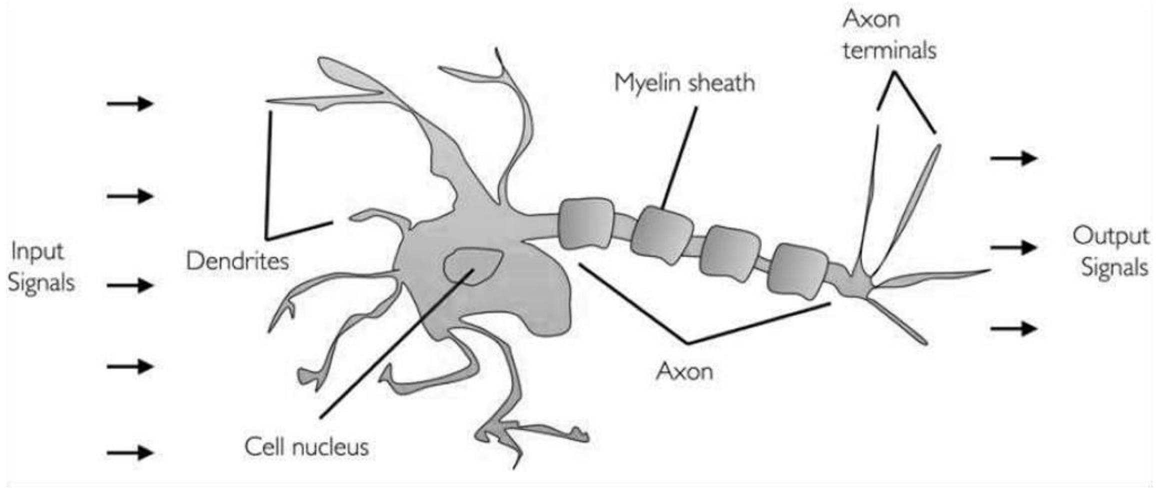
- In 1943, researchers Warren McCullock and published their first concept of simplified brain cell.
- This was called McCullock-Pitts (MCP) neuron.
- They described such a nerve cell as a simple logic gate with binary outputs.
- Multiple signals arrive at the dendrites and are then integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

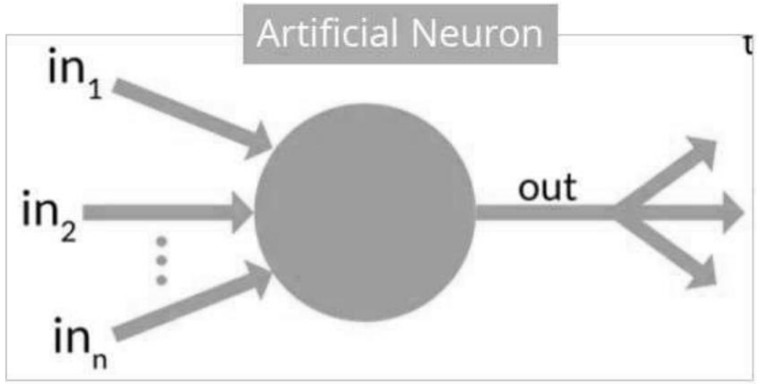
Warren Sturgis McCulloch

(1898 – 1969)

Walter Harry Pitts, Jr.

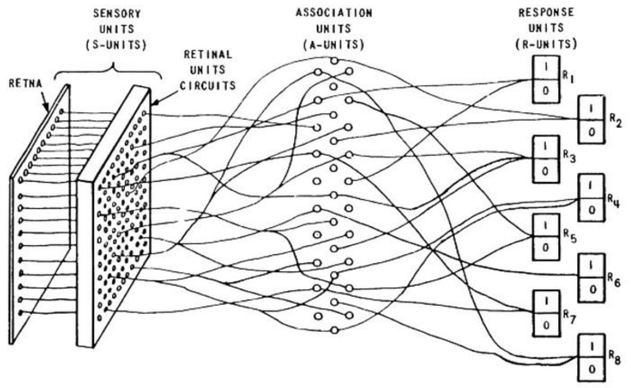
(1923 – 1969)



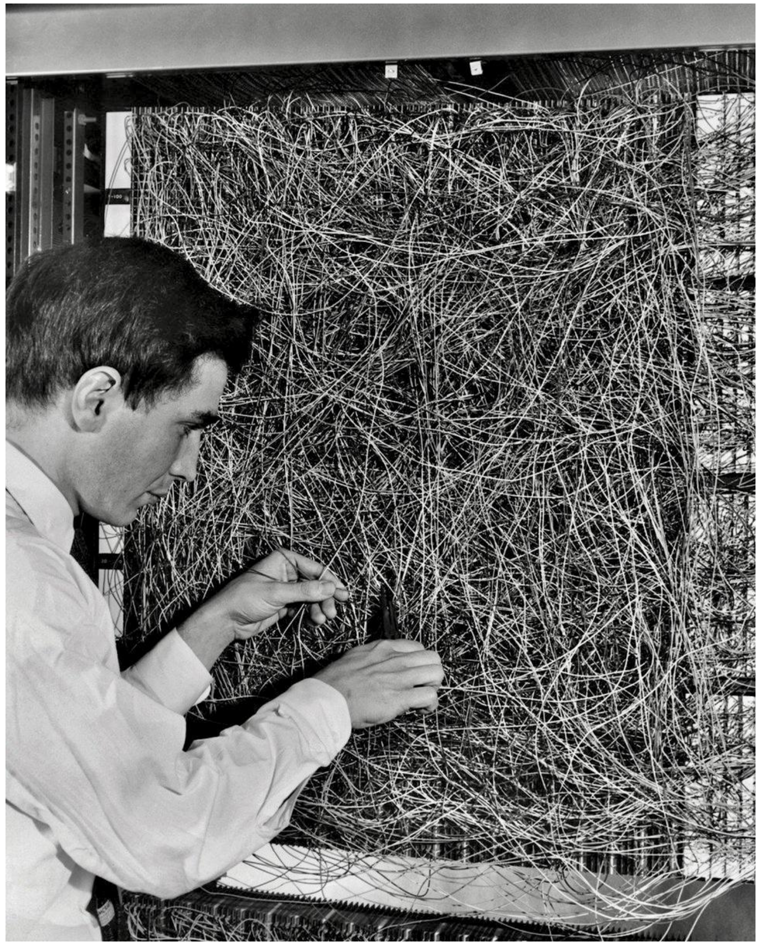


The Perceptron: Building Block of Neural Networks

- In 1953, inspired by McCullock work, Frank Rosenblatt invented the Perceptron.
- The Perceptron is the simplest form of a neural network
- Binary classifier: separates data into two categories
- Models a single neuron with multiple inputs and one output

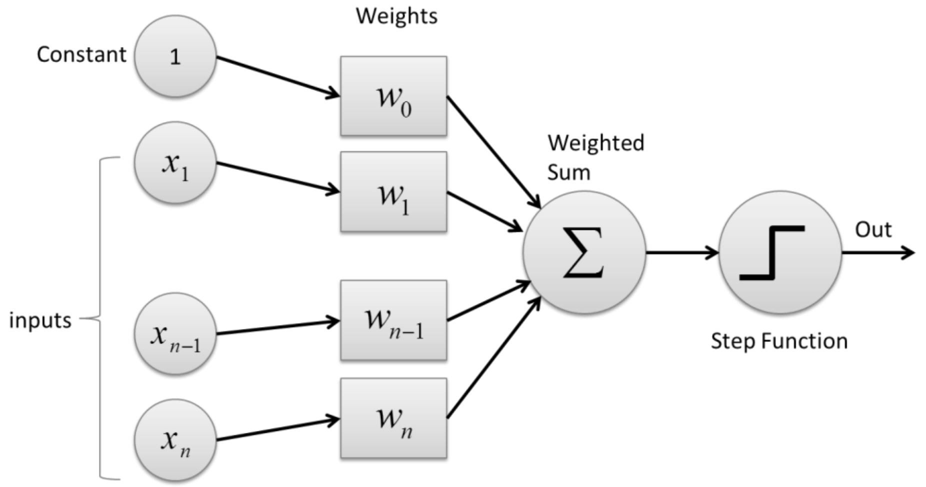


F. Rosenblatt



The Perceptron

- Inputs: $x_1, x_2, \ldots, x_n$
- Weights: $w_1, w_2, \ldots, w_n$
- Bias: b
- Activation function: Step function
- Output: 1 if weighted sum > threshold, 0 otherwise

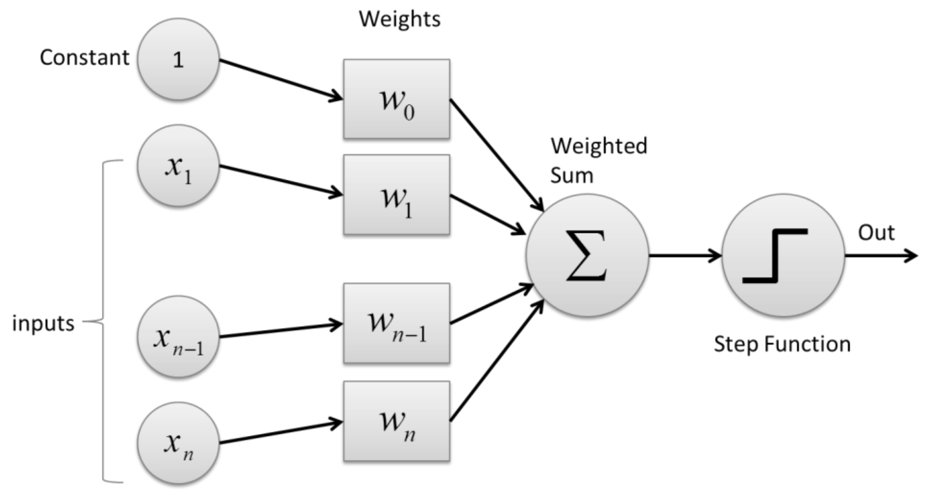


How a Perceptron Works

1. Multiply each input by its corresponding weight
2. Sum all weighted inputs
3. Add the bias term
4. Apply the activation function
5. Output the result

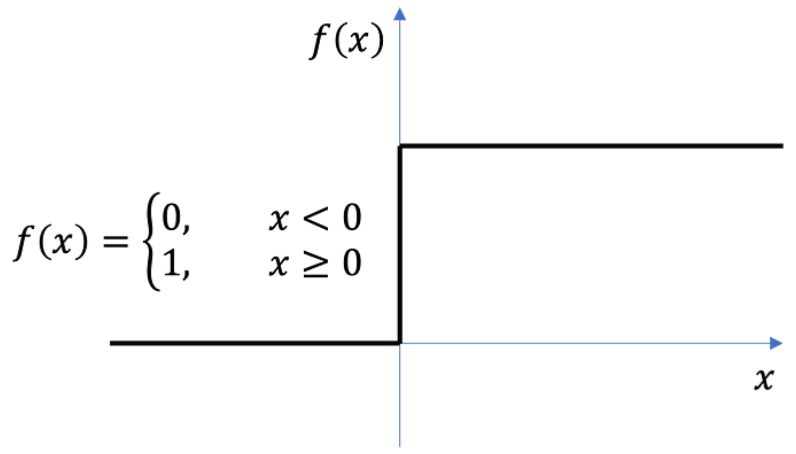
Mathematically:

- $z = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$
- output = activation(z)



Perceptron Activation Function

- **Step Function**:
  - Output: 1 if z ≥ 0, 0 if z < 0
  - Used in original perceptrons
  - Not differentiable at 0



Perceptron Learning Rule

For each training example:

1. Calculate predicted output y_pred
2. Calculate error: error = y_true - y_pred
3. Update weights: w_new = w_old + learning_rate * error * x
4. Update bias: b_new = b_old + learning_rate * error

Step-by-Step Hand Calculation for AND Gate

Let's work through the perceptron learning algorithm by hand for the AND gate:

- Training data: X = [[0,0], [0,1], [1,0], [1,1]], y = [0, 0, 0, 1]
- Learning rate (η) = 0.1
- Initial weights (randomly assigned): $w_1 = 0.3$, $w_2 = -0.1$
- Initial bias: $b = 0.2$

First Iteration:

Example 1: (0,0) → 0

- Inputs: $x_1 = 0$, $x_2 = 0$
- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(0) + (-0.1)(0) + 0.2 = 0.2$
- Activation: output = 1 (since $z > 0$)
- True output: $y = 0$
- Error: error = y - output = 0 - 1 = -1
- Weight updates:
    - $w_1 = w_1 + \eta * \text{error} * x_1 = 0.3 + 0.1 * (-1) * 0 = 0.3$
    - $w_2 = w_2 + \eta * \text{error} * x_2 = -0.1 + 0.1 * (-1) * 0 = -0.1$
    - $b = b + \eta * \text{error} = 0.2 + 0.1 * (-1) = 0.1$

Step-by-Step Hand Calculation for AND Gate

Example 2: (0,1) → 0

- Inputs: $x_1 = 0, x_2 = 1$
- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(0) + (-0.1)(1) + 0.1 = 0$
- Activation: output = 1 (since $z \geq 0$)
- True output: $y = 0$
- Error: error = y - output = 0 - 1 = -1
- Weight updates:
  - $w_1 = w_1 + \eta * \text{error} * x_1 = 0.3 + 0.1 * (-1) * 0 = 0.3$
  - $w_2 = w_2 + \eta * \text{error} * x_2 = -0.1 + 0.1 * (-1) * 1 = -0.2$
  - $b = b + \eta * \text{error} = 0.1 + 0.1 * (-1) = 0$

Example 3: (1,0) → 0

- Inputs: $x_1 = 1, x_2 = 0$
- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(1) + (-0.2)(0) + 0 = 0.3$
- Activation: output = 1 (since $z > 0$)
- True output: $y = 0$
- Error: error = y - output = 0 - 1 = -1
- Weight updates:
  - $w_1 = w_1 + \eta * \text{error} * x_1 = 0.3 + 0.1 * (-1) * 1 = 0.2$
  - $w_2 = w_2 + \eta * \text{error} * x_2 = -0.2 + 0.1 * (-1) * 0 = -0.2$
  - $b = b + \eta * \text{error} = 0 + 0.1 * (-1) = -0.1$

Step-by-Step Hand Calculation for AND Gate

Example 4: (1,1) → 1

- Inputs: $x_1 = 1$, $x_2 = 1$
- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.2(1) + (-0.2)(1) + (-0.1) = -0.1$
- Activation: output = 0 (since $z < 0$)
- True output: $y = 1$
- Error: error = y - output = 1 - 0 = 1
- Weight updates:
  - $w_1 = w_1 + \eta * \text{error} * x_1 = 0.2 + 0.1 * 1 * 1 = 0.3$
  - $w_2 = w_2 + \eta * \text{error} * x_2 = -0.2 + 0.1 * 1 * 1 = -0.1$
  - $b = b + \eta * \text{error} = -0.1 + 0.1 * 1 = 0$

End of Iteration 1:

- Updated weights: $w_1 = 0.3$, $w_2 = -0.1$
- Updated bias: $b = 0$

Second Iteration

Example 1: (0,0) → 0

- Inputs: $x_1 = 0$, $x_2 = 0$
- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(0) + (-0.1)(0) + 0 = 0$
- Activation: output = 1 (since $z \geq 0$)
- True output: $y = 0$
- Error: error = y - output = 0 - 1 = -1
- Weight updates:
    - $w_1 = w_1 + \eta * \text{error} * x_1 = 0.3 + 0.1 * (-1) * 0 = 0.3$
    - $w_2 = w_2 + \eta * \text{error} * x_2 = -0.1 + 0.1 * (-1) * 0 = -0.1$
    - $b = b + \eta * \text{error} = 0 + 0.1 * (-1) = -0.1$

Example 2: (0,1) → 0

- Inputs: $x_1 = 0$, $x_2 = 1$
- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(0) + (-0.1)(1) + (-0.1) = -0.2$
- Activation: output = 0 (since $z < 0$)
- True output: $y = 0$
- Error: error = y - output = 0 - 0 = 0
- Weight updates (no change as error = 0):
    - $w_1 = 0.3$
    - $w_2 = -0.1$
    - $b = -0.1$
- **After several iterations**, the perceptron will converge to weights that correctly classify all AND gate examples.

Python Implementation Perceptron from Scratch

```python
from sklearn.linear_model import Perceptron
import numpy as np

   # Training data for AND gate
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 0, 0, 1])

   # Initialize and train Perceptron
model = Perceptron(max_iter=100, eta0=0.1, random_state=42)
model.fit(X, y)
# Results
print("Weights:", model.coef_)
print("Bias:", model.intercept_)
print("Predictions:", model.predict(X))
```

```
Weights: [[0.2 0.2]]
Bias: [-0.2]
Predictions: [0 0 0 1]
```

The code shows a scikit-learn Perceptron implementation for the AND gate problem.

The code:

1.Imports NumPy, scikit-learn's Perceptron, and matplotlib

2.Sets up the training data for the AND gate

3.Initializes a Perceptron with 100 max iterations and a random seed of 42

4.Trains the perceptron on the AND gate data

5.Prints the learned weights, bias, and predictions

The output shows:

- **Weights: [[0.2 0.2]]** - The perceptron learned to assign a weight of 0.2 to both inputs
- **Bias: [-0.2]** - The bias is -0.2
- **Predictions: [0 0 0 1]** - The perceptron correctly classified all four examples of the AND gate

With these weights and bias, the decision function is: 0.2×(input1) + 0.2×(input2) – 0.2

For the four input combinations:

- [0,0]: 0.2×0 + 0.2×0 - 0.2 = -0.2 < 0 → output 0
- [0,1]: 0.2×0 + 0.2×1 – 0.2 = 0 → output 0
- [1,0]: 0.2×1 + 0.2×0 – 0.2 = 0 → output 0
- [1,1]: 0.2×1 + 0.2×1 – 0.2 = 0.2 > 0 → output 1

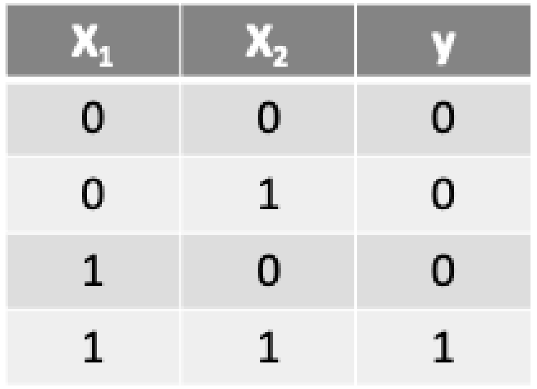
This perceptron implements the AND gate logic.

The decision boundary is the line $2x_1 + 2x_2 - 0.2 = 0$, which separates the point (1,1) from the other three points.

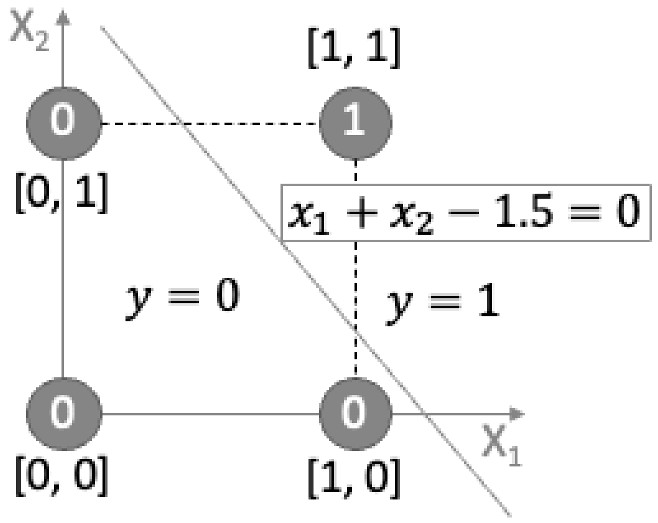
Open in Colab

Decision Boundary

- The perceptron learns a decision boundary: $w_1x_1 + w_2x_2 + b = 0$
- Points above the line are classified as 1
- Points below the line are classified as 0
- For AND gate, only the point (1,1) should be above the line

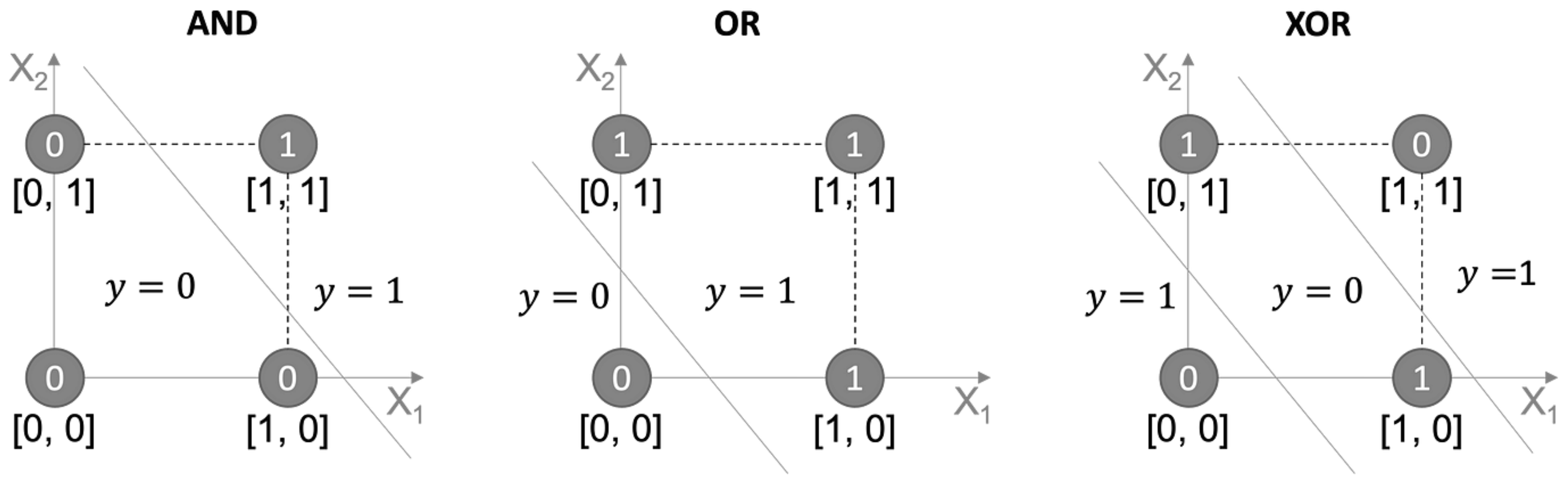


X_1	X_2	y
0	0	0
0	1	0
1	0	0
1	1	1



Limitations of Simple Perceptron

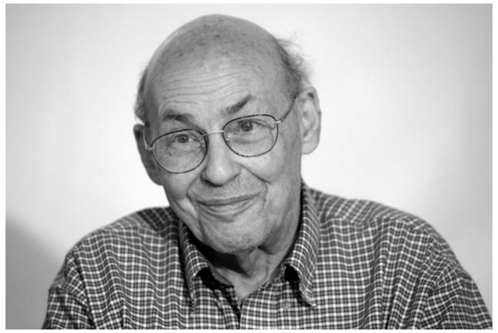
- Can only learn linearly separable patterns
- Cannot solve XOR problem (need multiple layers)
- No probabilistic output
- Simple update rule isn't suitable for complex problems



Introduction to Neural Networks

Multi-Layer Neural Network

The Multi-Layer Percecptron (MLP)



Limitations of the Perceptron

While useful for linearly separable problems, the single perceptron cannot solve complex problems like XOR classification, as demonstrated by Minsky and Papert in their 1969 book "Perceptrons.“

The Multi-Layer Perceptron

The Multi-Layer Perceptron addresses the limitations of the single perceptron by introducing:

- Multiple layers of neurons
- Non-linear activation functions
- More sophisticated learning algorithms

Structure of an MLP

Definition: An MLP is a class of feedforward artificial neural network that consists of at least three layers of nodes: **input**, **hidden**, and **output** layers.

Key Feature: Each neuron in one layer is connected to every neuron in the next layer (fully connected).

1. Input Layer

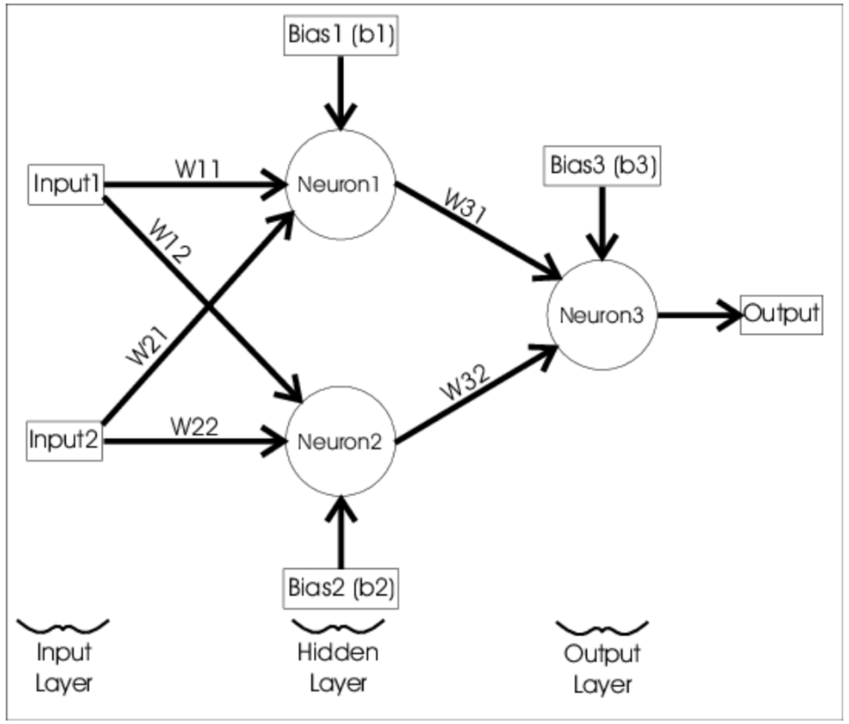
- Receives the raw input features
- One neuron per input feature
- No computation occurs here; inputs are simply passed forward

2. Hidden Layer(s)

- One or more layers between input and output
- Each neuron in a hidden layer:
- Receives inputs from all neurons in the previous layer
- Computes a weighted sum
- Applies a non-linear activation function
- Passes the result to the next layer

3. Output Layer

- Produces the final prediction or classification
- Structure depends on the task:
    - Regression: Often a single neuron with linear activation
    - Binary classification: One neuron with sigmoid activation
    - Multi-class classification: Multiple neurons (one per class) with softmax activation



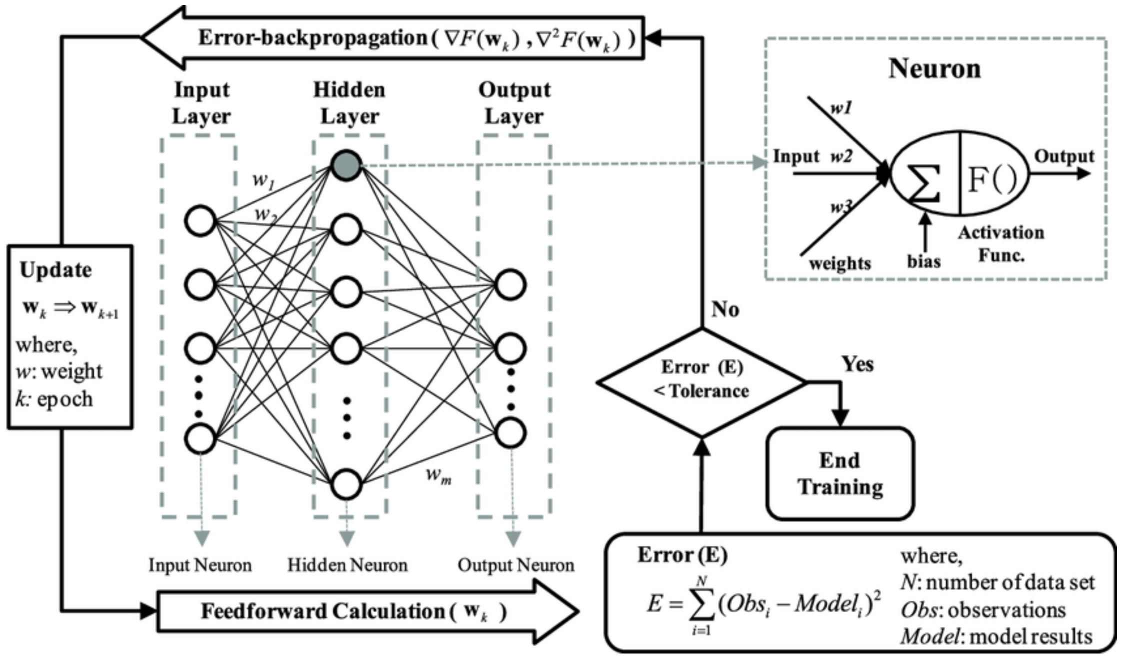
The Neural Network Model to solve the XOR Logic (from: https://stopsmokingaids.me/)

How Neural Networks Learn

Training Process:

1. Feed data into the network.
2. Compute the output using weights.
3. Compare the output with the correct answer (loss calculation).
4. Adjust weights using **backpropagation** & **gradient descent** to improve accuracy.

- **Loss Function**: MSE for regression, Cross-Entropy for classification.
- **Optimization**: Backpropagation + Gradient Descent (or Adam).



Open in Colab

Multi-Layer Perceptron

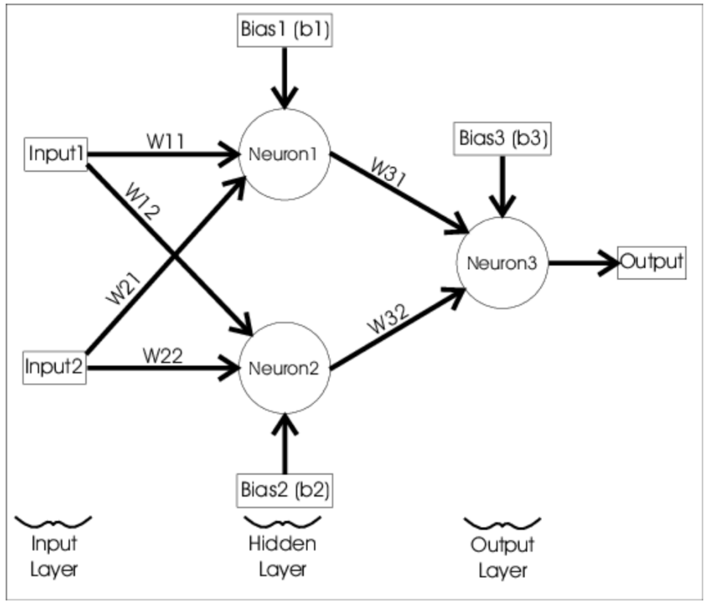
```python
from sklearn.neural_network import MLPClassifier
import numpy as np
# XOR input and output
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 1, 1, 0])
# Define MLP with 1 hidden layer of 2 neurons (minimal config for XOR)
mlp = MLPClassifier(hidden_layer_sizes=(2,), activation='tanh', solver='adam', learning_rate_init=0.01,
    max_iter=10000, random_state=42)
# Train the model
mlp.fit(X, y)
# Make predictions
predictions = mlp.predict(X)
print("Predictions:\n", predictions)
print("\nWeights (input to hidden):\n", mlp.coefs_[0])
print("\nBias hidden:\n", mlp.intercepts_[0])
print("\nWeights (hidden to output):\n", mlp.coefs_[1])
print("\nBias output:\n", mlp.intercepts_[1])
```

```
Weights (input to hidden):
 [[ 2.7144501   3.27401218]
 [-2.73418453 -3.17014048]]

Bias hidden:
 [ 1.21994174 -1.63451199]
```

```
Weights (hidden to output):
 [[-4.37775211]
 [ 4.46553876]]

Bias output:
 [3.61855675]
```



The Neural Network Model to solve the XOR Logic (from: https://stopsmokingaids.me/)

Open in Colab

BUSINESS INTELLIGENCE AND DATA ANALYTICS
– WE PREDICT THE FUTURE –

Institute of Analytics
Recognized Programme

University of Petra

Introduction to Word Embeddings

How Did We Represent Words Pre-2013

- Traditional models like Bag-of-Words (BoW) or TF-IDF, treat words as independent, ignoring semantic similarity.
- **One-hot encoding**: Sparse, binary vectors (dimension = vocabulary size)
- Example: "king" and "queen" are as unrelated as "king" and "banana" in BoW.

Word	Dimension 1 (cat)	Dimension 2 (dog)	Dimension 3 (fish)	Dimension 4 (bird)
cat	1	0	0	0
dog	0	1	0	0
fish	0	0	1	0
bird	0	0	0	1

The Evolution of Word Representations

- **Problem**: How do we represent meaning mathematically?
- **Solution**: Distributional hypothesis - "You shall know a word by the company it keeps" (J.R. Firth, 1957)
- J.R. Firth **did not** provide a detailed technical implementation like an algorithm or computational method. His statement was more of a **linguistic philosophy** or a **theoretical principle**, not a specific engineering method.
- He worked mostly in the 1930s–1950s, before computers were widely used for language processing. His idea influenced the later development of:
  - **Collocational analysis** (studying which words often appear together)
  - **Contextual analysis** in linguistics
- And much later, it inspired the **distributional hypothesis** in computational linguistics, especially by scholars like Zellig Harris and later computational models (Word2Vec, etc.)
- In short:
  - **Firth introduced the principle → Later researchers and engineers built models based on it.**

Word2Vec (Tomas Mikolov et al., 2013)

- Developed by Tomas Mikolov and team at Google.

Key Innovation

- Transformed NLP by creating dense vector representations through prediction-based models

Two Architectures

- **Continuous Bag of Words (CBOW)**:
  - Predicts target word from context words
  - Faster training, better for frequent words
- **Skip-gram**:
  - Predicts context words from target word
  - Better for rare words, captures more semantic information

Characteristics

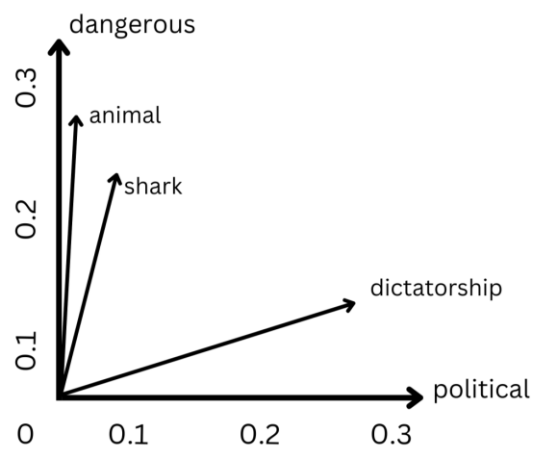
- Uses **shallow neural networks** and trains on local context windows.
- Typically, 100-300 dimensions (vs. vocabulary size)
- Linear relationships: king - man + woman ≈ queen
- Efficient training through negative sampling
- Limitations: Fixed vectors, one vector per word regardless of context

Vector Spaces and Word Embeddings

What is a Vector Space?

- A mathematical space where each word is represented as a point (or vector) in multi-dimensional space.
- Words are encoded as dense numerical vectors instead of one-hot or sparse representations (**Word Embeddings**) e.g., "king" → [0.21, 0.72,….,…., 0.35]
- Word Embeddings captures semantic and syntactic relationships about/between words.
- Each dimension potentially captures semantic meaning.
- These vectors are learned from text by models like Word2Vec or GloVe.

Word	Dimension 1 (political)	Dimension 2 (dangerous)
shark	0.05	0.22
animal	0.03	0.25
dangerous	0.07	0.32
political	0.31	0.04
dictatorship	0.28	0.15



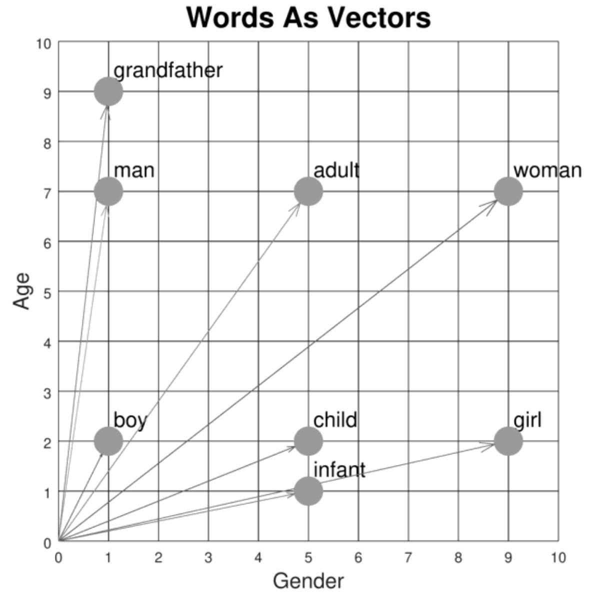
Vector Spaces and Word Embeddings

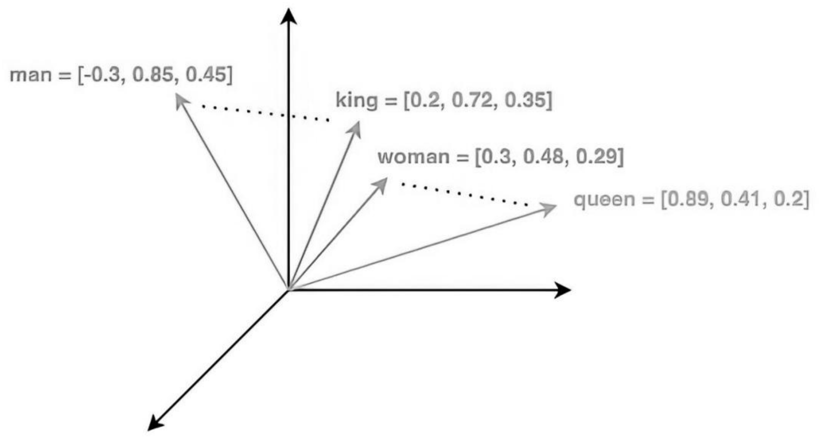
Why Use a Vector Space?

- Makes it possible to compare, visualize, and manipulate meanings of words using math.
- Enables operations like:
  - Similarity: "king" is close to "queen"
  - Analogy: "king" - "man" + "woman" ≈ "queen"

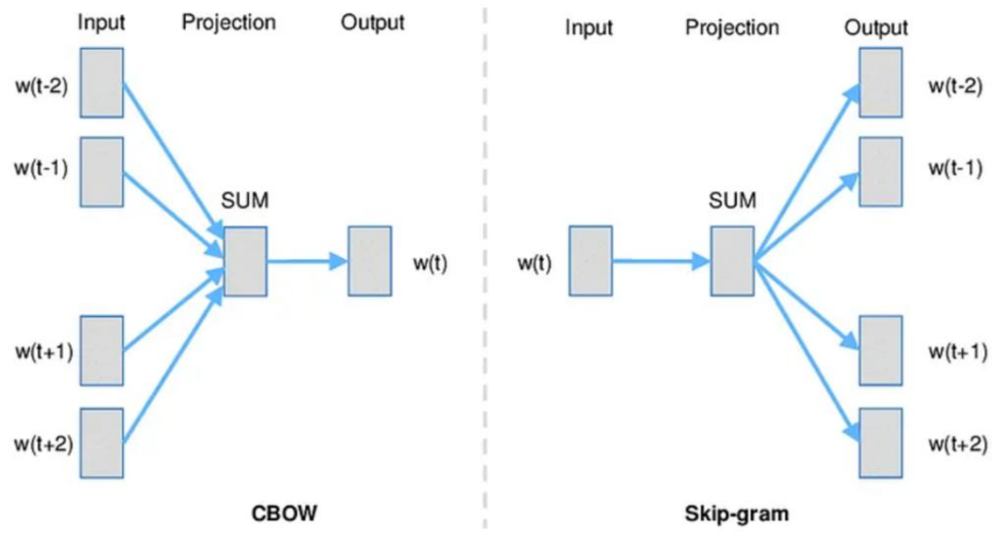
Properties of Vector Space

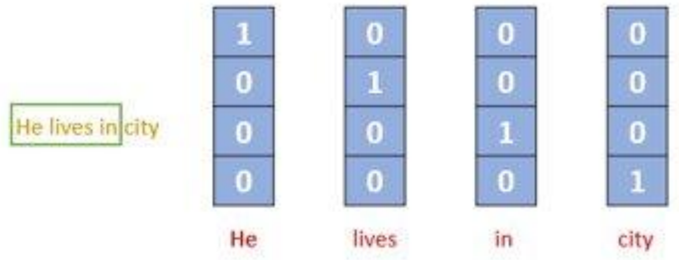
- Semantic relationships are preserved (e.g., "shark" is closer to "dangerous" than "political").
- Similar meanings → closer vectors.
- Dissimilar meanings → vectors farther apart.

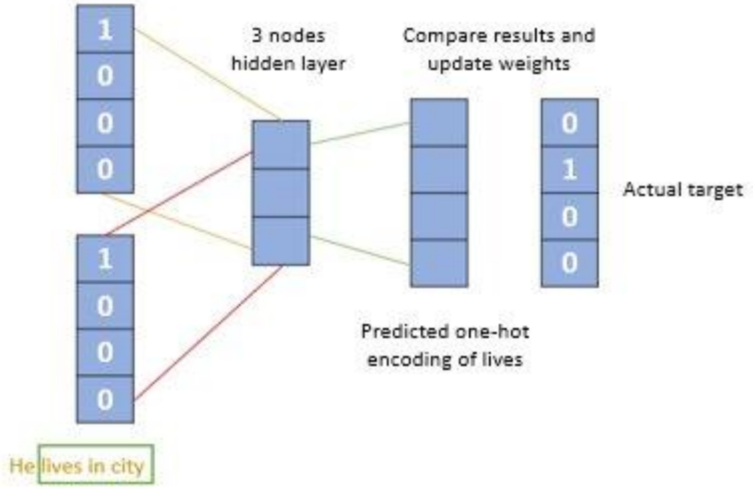




CBow and Skip-Gram Models



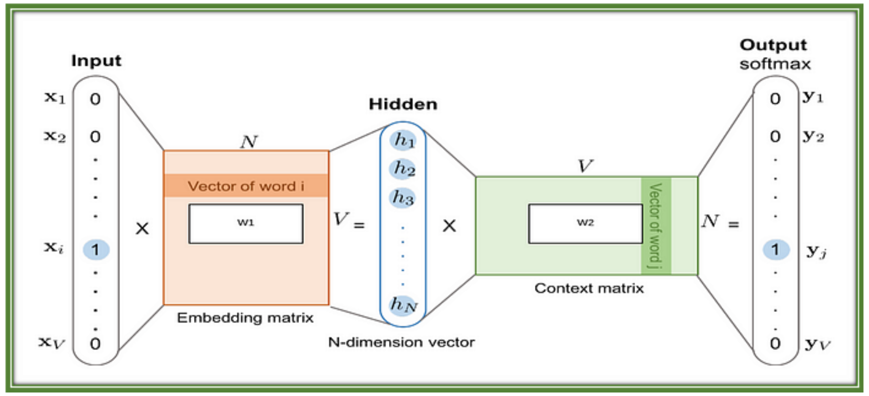




Skip-gram Architecture (Word2Vec)

This diagram illustrates how Word2Vec's **Skip-gram model** works:

- **Input**: A one-hot encoded vector for the center word (word *i*).
- **Embedding Matrix**: Multiplies the input vector to produce a **dense embedding** (N-dimensional vector) — this becomes the **vector representation of the input word**.
- **Context Matrix**: The dense vector is then multiplied with another matrix to predict surrounding context words via softmax output.
- **Output**: A probability distribution over the vocabulary, aiming to maximize the likelihood of actual context words.
- This training process helps learn **meaningful word vectors** based on how words appear in context.



https://python.plainenglish.io/understanding-word-embeddings-tf-idf-word2vec-glove-fasttext-996a59c1a8d3

Glove (Pennington, Socher, Manning 2014)

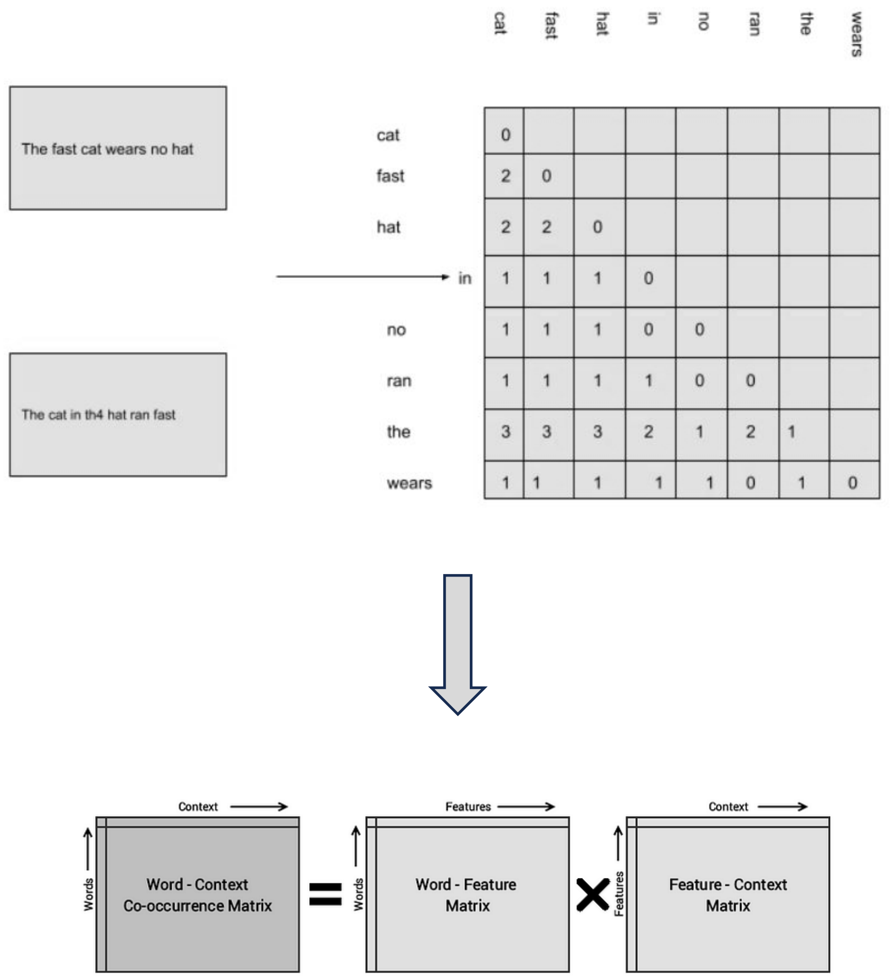
- Developed by Stanford NLP Group (Pennington, Socher, Manning)
- **Key Innovation**

Bridges the gap between count-based methods and prediction-based methods by using global co-occurrence statistics to learn word vectors

- **Approach**
- Builds a word-word co-occurrence matrix over a large corpus
- Learns embeddings by factorizing the matrix using a weighted least squares objective

Characteristics

- Captures global statistical information while maintaining useful properties of local context
- Produces dense word vectors (typically 100–300 dimensions)
- Linear relationships in vector space are preserved: king - man + woman ≈ queen
- Trained on massive corpora (Wikipedia, Common Crawl)
- **Limitations: Ignores context variability—still one vector per word regardless of usage**



The fast cat wears no hat

The cat in th4 hat ran fast

	cat	fast	hat	in	no	ran	the	wears
cat	0							
fast	2	0						
hat	2	2	0					
in	1	1	1	0				
no	1	1	1	0	0			
ran	1	1	1	1	0	0		
the	3	3	3	2	1	2	1	
wears	1	1	1	1	1	0	1	0

Measuring Similarity Between Word Vectors

Why Compare Word Vectors?

- Word embeddings map words into a vector space.
- **Words with similar meanings** are placed **close together** in that space.
- To quantify this "closeness," we use **vector similarity**.

Cosine Similarity

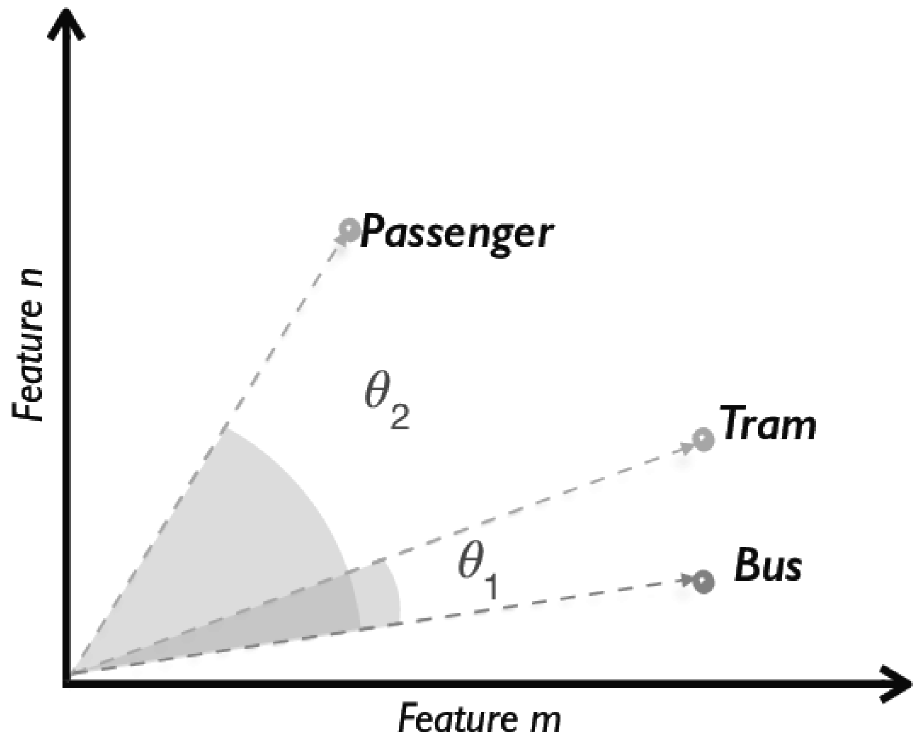
Most common metric used to compare word vectors:

$$\text{cosine\_similarity}(\vec{A}, \vec{B}) = \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\| \|\vec{B}\|}$$

- Measures the **angle** between two vectors (not their magnitude).
- Ranges from **-1 to 1**:
  - 1 → Same direction (very similar)
  - 0 → Orthogonal (unrelated)
  - -1 → Opposite directions (very different)

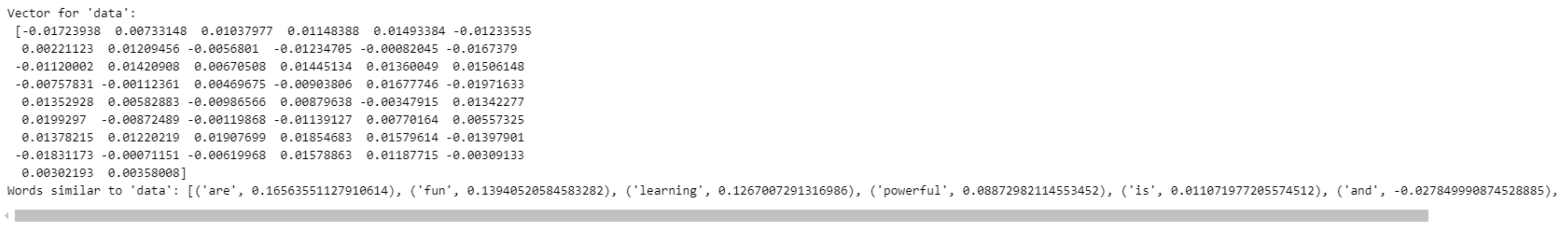
Intuition

- Vectors for `"king"` and `"queen"` will have high cosine similarity.
- Vectors for `"apple"` and `"keyboard"` will have low similarity.



Learn Embeddings based on a New Corpus

```python
from gensim.models import Word2Vec
# Sample corpus
sentences = [['data', 'science', 'is', 'fun'],
    ['machine', 'learning', 'is', 'powerful'],
    ['data', 'and', 'learning', 'are', 'related']]
# Train the model
model = Word2Vec(sentences, vector_size=50, window=2, min_count=1, workers=2)
# Access the embedding for a word
print("Vector for 'data':\n", model.wv['data’])
# Find similar words
print("Words similar to 'data':", model.wv.most_similar('data'))
```



```
Vector for 'data':
 [-0.01723938  0.00733148  0.01037977  0.01148388  0.01493384 -0.01233535
  0.00221123  0.01209456 -0.0056801  -0.01234705 -0.00082045 -0.0167379
 -0.01120002  0.01420908  0.00670508  0.01445134  0.01360049  0.01506148
 -0.00757831 -0.00112361  0.00469675 -0.00903806  0.01677746 -0.01971633
  0.01352928  0.00582883 -0.00986566  0.00879638 -0.00347915  0.01342277
  0.0199297  -0.00872489 -0.00119868 -0.01139127  0.00770164  0.00557325
  0.01378215  0.01220219  0.01907699  0.01854683  0.01579614 -0.01397901
 -0.01831173 -0.00071151 -0.00619968  0.01578863  0.01187715 -0.00309133
  0.00302193  0.00358008]
Words similar to 'data': [('are', 0.16563551127910614), ('fun', 0.13940520584583282), ('learning', 0.1267007291316986), ('powerful', 0.08872982114553452), ('is', 0.011071977205574512), ('and', -0.027849990874528885),
```

Open in Colab

Compute Word Similarities

Use Fake Embeddings as an Example

```
import numpy as np

from sklearn.metrics.pairwise import cosine_similarity

# Fake word vectors (3D for simplicity)

word_vectors = {

    "king": np.array([0.8, 0.65, 0.1]),

    "queen": np.array([0.78, 0.66, 0.12]),

    "man": np.array([0.9, 0.1, 0.1]),

    "woman": np.array([0.88, 0.12, 0.12]),

    "apple": np.array([0.1, 0.8, 0.9]),

}

def similarity(w1, w2):

   return cosine_similarity([word_vectors[w1]], [word_vectors[w2]])[0][0]


print("Similarity(king, queen):", similarity("king", "queen"))

print("Similarity(man, woman):", similarity("man", "woman"))

print("Similarity(king, apple):", similarity("king", "apple"))
```

Open in Colab

Compute Word Similarities
Use Real Embeddings from Gensim Library

```python
import gensim.downloader as api
from gensim.models import Word2Vec

# Load pre-trained Word2Vec model
word2vec_model = api.load("word2vec-google-news-300")

# Find similar words
similar_words = word2vec_model.most_similar('computer', topn=5)
print("Words similar to 'computer':", similar_words)

# Word analogies
result = word2vec_model.most_similar(positive=['woman', 'king'],
   negative=['man'], topn=1)
print("king - man + woman =", result)

# Train your own Word2Vec model
sentences = [["cat", "say", "meow"], ["dog", "say", "woof"]]
model = Word2Vec(sentences, vector_size=100, window=5, min_count=1, workers=4)

# Get vector for a word
cat_vector = model.wv['cat']
print("Vector for 'cat':", cat_vector[:5])  # Show first 5 dimensions
```

Gensim
Gensim is a powerful open-source Python library designed specifically for unsupervised topic modeling and natural language processing tasks, with a strong focus on working with large corpora. It excels in handling word embeddings and semantic similarity, offering efficient implementations of models like Word2Vec, FastText, and Doc2Vec. Gensim is known for its memory-efficient, streaming-based approach, which allows it to process text data without loading everything into memory. This makes it especially useful for working with real-world, large-scale text data.

Open in Colab

Word Embeddings Example using Spacy Library

- pip install spacy
- python -m spacy download en_core_web_md

```python
import spacy
nlp = spacy.load("en_core_web_md")

word1 = nlp("king")
word2 = nlp("queen")
print("Similarity:", word1.similarity(word2))
```

spaCy
spaCy is a fast and robust natural language processing library for Python that provides industrial-strength tools for text preprocessing and linguistic analysis. It comes with pre-trained models for multiple languages and supports features like tokenization, part-of-speech tagging, named entity recognition, dependency parsing, and sentence segmentation. spaCy is designed for performance and ease of use in production environments and integrates well with deep learning libraries. While it's not primarily focused on word embeddings, it includes pre-trained word vectors and supports similarity comparisons out of the box.

Open in Colab

Contextual Word Embeddings (e.g., ELMo, BERT)

Key Innovation

Unlike static embeddings (Word2Vec, GloVe), contextual models generate **different vectors for the same word** depending on its context.

Approach

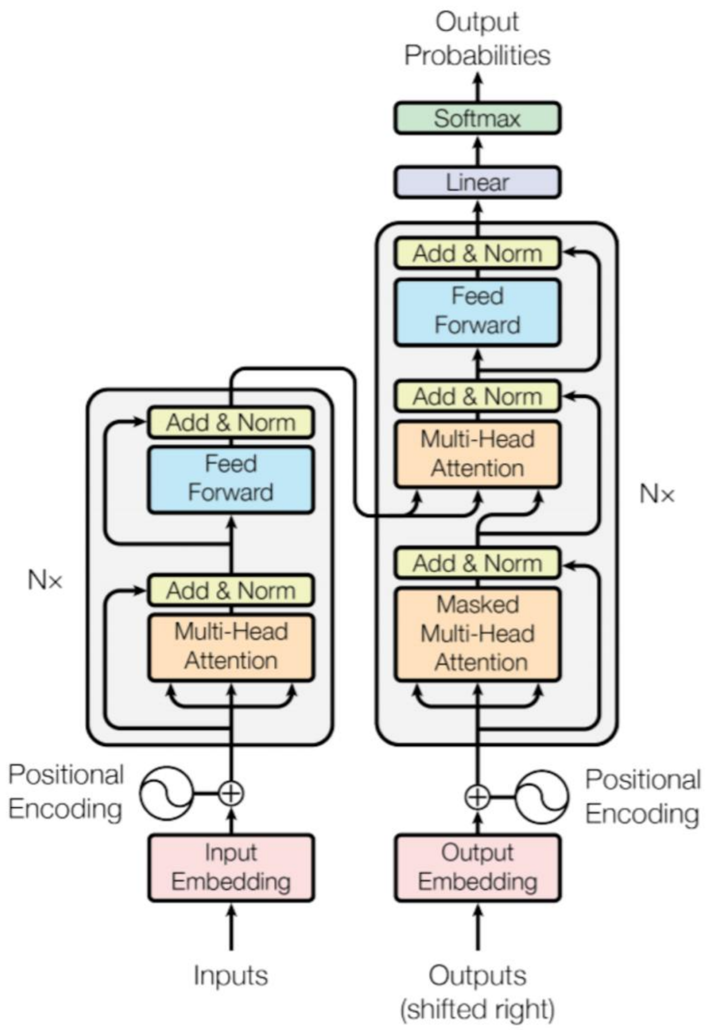
Uses deep, pre-trained neural networks (often transformer-based)
Embeddings are derived from entire sentences, capturing syntax and semantics dynamically

Examples

- **ELMo (2018)**: Contextualizes word representations using deep bi-directional LSTMs Neural Networks
- **BERT (2018)**: Transformer-based neural networks trained with masked language modeling and next sentence prediction
- GPT (2018): Transformer-based unidirectional language model focused on generation.

Characteristics

- Embeddings are **context-sensitive** (e.g., “bank” in “river bank” vs. “savings bank”)
- Each word is embedded based on its role in the sentence.
- Embeddings vary for the same word depending on its position and meaning.
- Significantly improve performance on downstream NLP tasks.
- **Limitations: Computationally intensive; harder to interpret than static embeddings**



Practical Implementation: BERT in Python

```
import torch
from transformers import BertModel, BertTokenizer
# Load pre-trained BERT model and tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')
# Input text
text = "The quick brown fox jumps over the lazy dog."
# Tokenize input
inputs = tokenizer(text, return_tensors="pt")
# Get BERT embeddings
with torch.no_grad():
    outputs = model(**inputs)
# Last hidden states contain contextual embeddings for each token
last_hidden_states = outputs.last_hidden_state
# Get embedding for the first token (after [CLS])
word_embedding = last_hidden_states[0, 1].numpy()
print(f"BERT embedding for 'The' (first 5 dimensions): {word_embedding[:5]}")
# Get embeddings for full sentence (CLS token)
sentence_embedding = last_hidden_states[0, 0].numpy()
```

Applications of Word Embeddings in NLP

1. **Semantic Similarity**
   Measure how similar two words, phrases, or documents are by comparing their vector representations.
   Example: Identifying that "doctor" and "physician" are closely related.
2. **Text Classification**
   Used as input features for tasks like spam detection, sentiment analysis, and topic classification.
   Embeddings provide rich, dense input for machine learning models.
3. **Named Entity Recognition (NER)**
   Help identify proper nouns and classify them into categories like person, location, or organization.
   Embedding-based models improve contextual understanding of named entities.
4. **Machine Translation**
   Map words from one language to another by aligning embeddings in multilingual space.
   Improves translation accuracy by leveraging semantic proximity.
5. **Question Answering & Chatbots**
   Used to understand queries and match them with appropriate answers or responses.
   Enable bots to interpret intent and context more accurately.

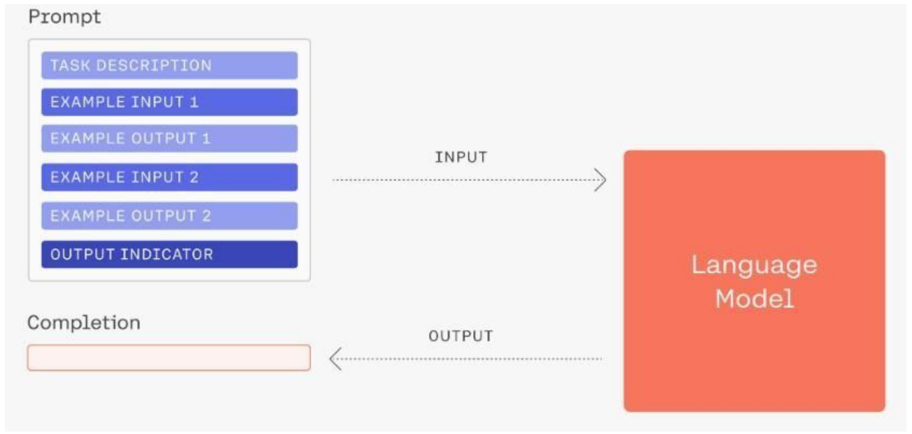
- **6. Information Retrieval**
  Enhance search engines by retrieving results based on semantic meaning, not just keyword matches.
  Example: Searching for “heart attack” returns documents containing “cardiac arrest.”

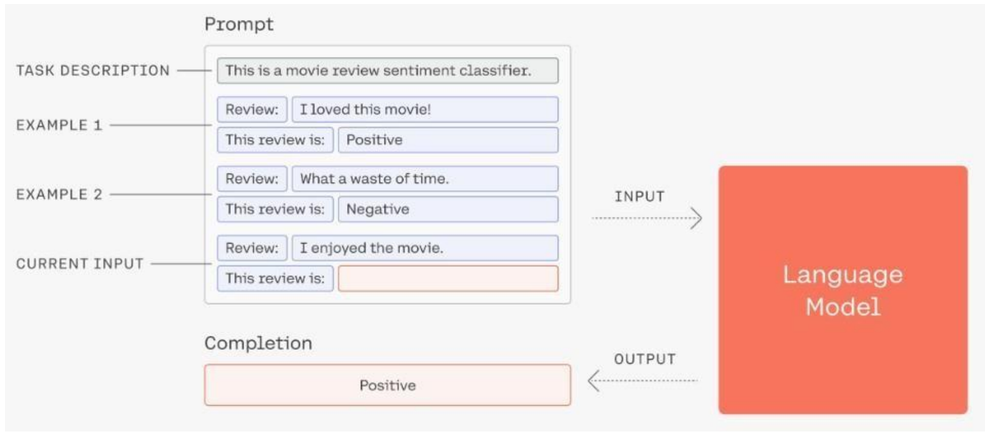
GenAI and Large Language Models

Overview

Generative AI (GenAI)

- Type of Artificial Intelligence that leverages AI to generate content or data
- Data can include text, images, audio, video, 3D models, code and video games
- Typically created in response to prompts (prompt engineering)
- Prompts are constructed inputs to language models to generate useful output
- Usually given with examples - zero shot versus few shot learning





Source: docs.cohere.com
https://docs.cohere.com/docs/prompt-engineering

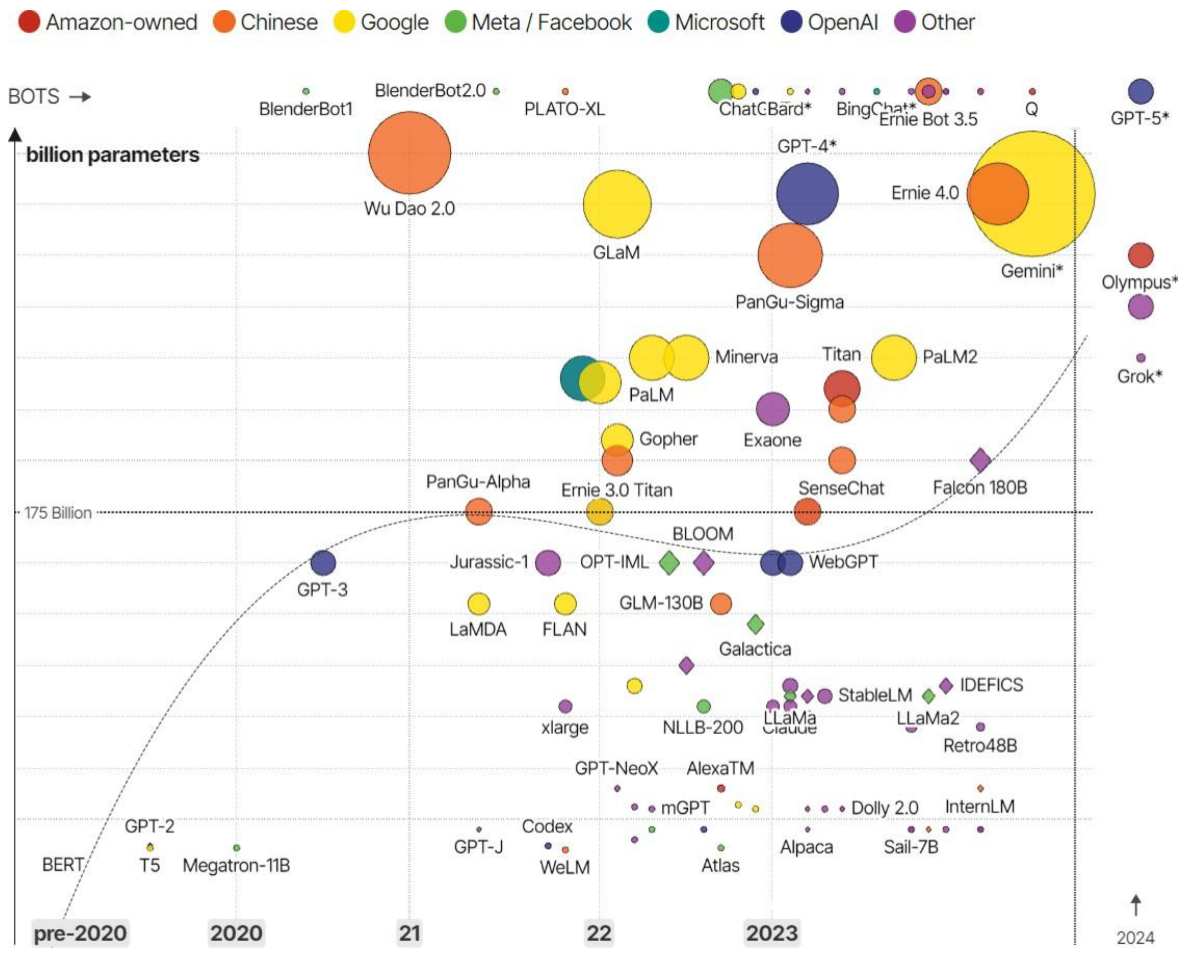
Large language models

- Type of machine learning model/algorithm
- Performs variety of natural language processing (NLP) tasks
- Learn, understand, and process human language efficiently
- E.g., generate/classify text, answer questions conversationally
- Uses hundreds of billions parameters

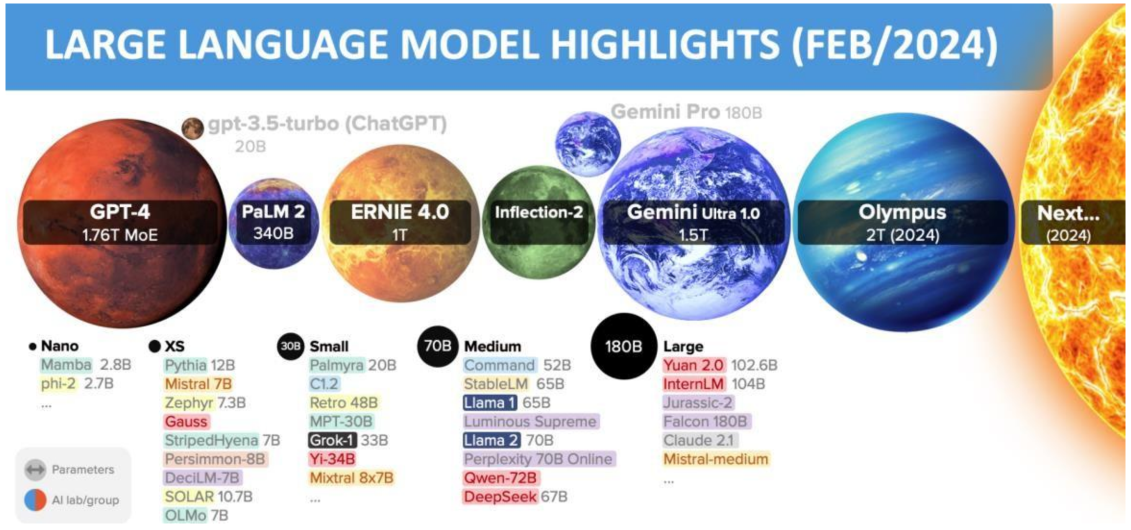
Large language models

- Trained with large amounts of data
- Based on neural networks (Transformers) that learn context and understanding through sequential data analysis
- Uses self-supervised learning to predict the next token in a sentence, given the surrounding context
- Process is repeated over and over until the model reaches acceptable level of accuracy
- GPT-4 (Generative Pre-trained Transformer)

Large language models (LLM)



Source: informationisbeautiful.net
https://informationisbeautiful.net/visualizations/the-rise-of-generative-ai-large-language-models-llms-like-chatgpt/

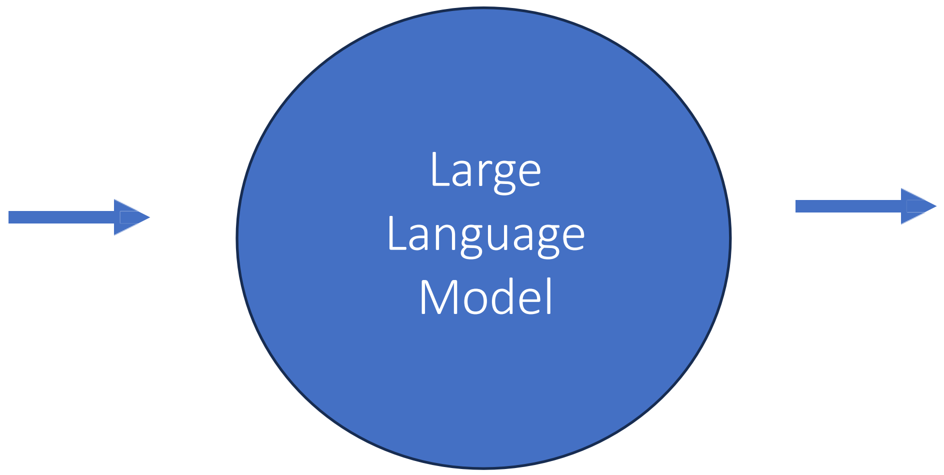


Source: https://lifearchitect.ai/models/

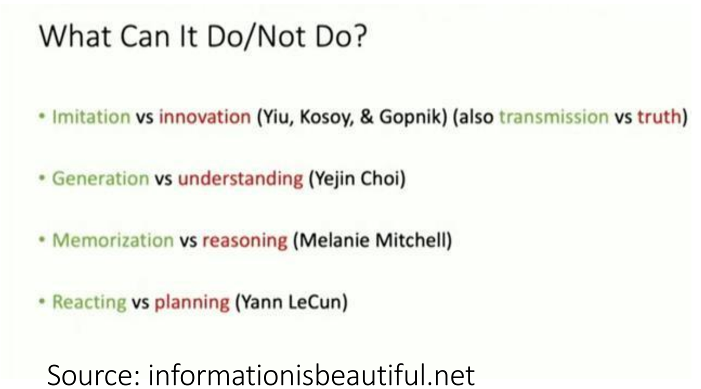
Large language models training

- LLMs return similar patterns to data it is trained on (not thinking)

- Wikipedia
- Common crawl
- Other sources



Text output



What Can It Do/Not Do?

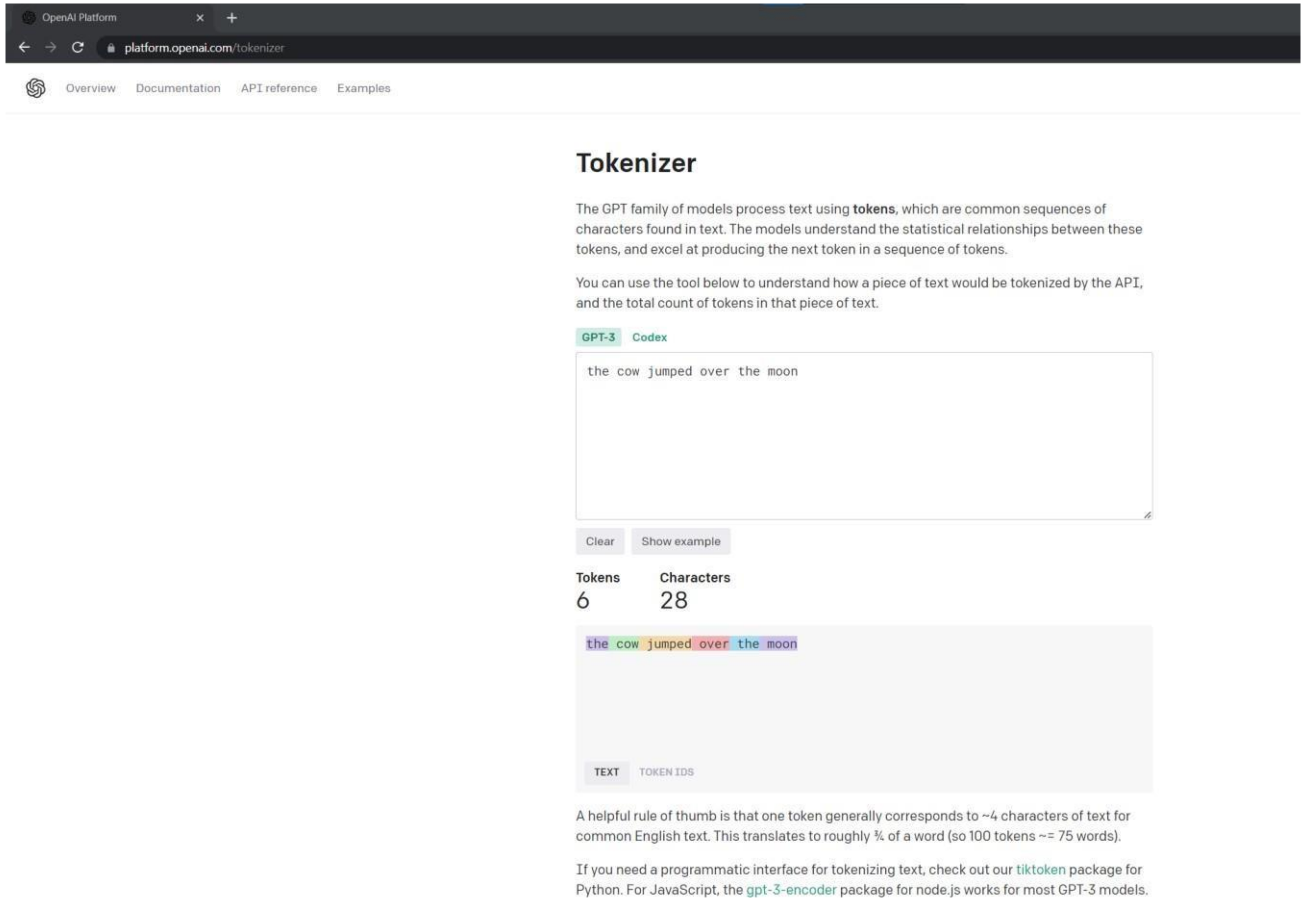
- Imitation vs innovation (Yiu, Kosoy, & Gopnik) (also transmission vs truth)
- Generation vs understanding (Yejin Choi)
- Memorization vs reasoning (Melanie Mitchell)
- Reacting vs planning (Yann LeCun)

Source: informationisbeautiful.net
Generative AI +: Education
https://www.youtube.com/@mit

Tokens

- Basic units of text/code LLM uses to process or generate language
- Can be characters, words, sub-words, segments of text or code
- Tokens generally = ~4 characters of text for common English
- ¾ of a word – 100 tokens ~= 75 words
- GPT models process text using tokens
- Common sequences of characters found in text
- Understands the statistical relationships between these tokens
- Used to predict next token in a sequence of tokens

Tokens



Tokenizer

The GPT family of models process text using **tokens**, which are common sequences of characters found in text. The models understand the statistical relationships between these tokens, and excel at producing the next token in a sequence of tokens.

You can use the tool below to understand how a piece of text would be tokenized by the API, and the total count of tokens in that piece of text.

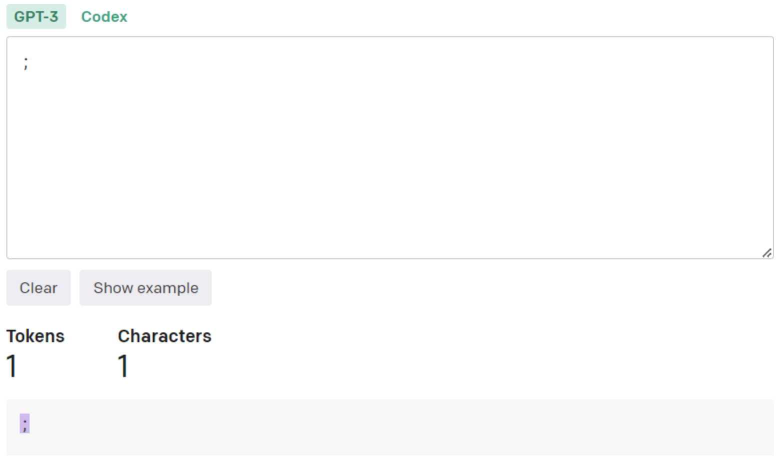
```
the cow jumped over the moon
```

Tokens 6 Characters 28

```
the cow jumped over the moon
```

A helpful rule of thumb is that one token generally corresponds to ~4 characters of text for common English text. This translates to roughly ¾ of a word (so 100 tokens ~= 75 words).

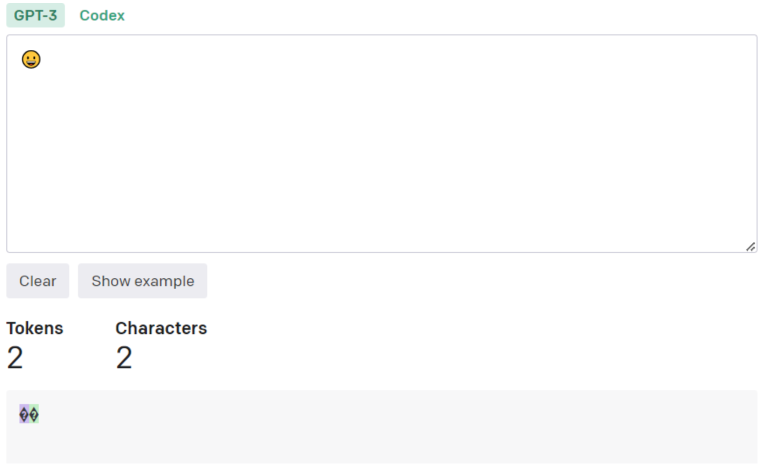
If you need a programmatic interface for tokenizing text, check out our tiktoken package for Python. For JavaScript, the gpt-3-encoder package for node.js works for most GPT-3 models.



```
;
```

Tokens 1 Characters 1

```
;
```



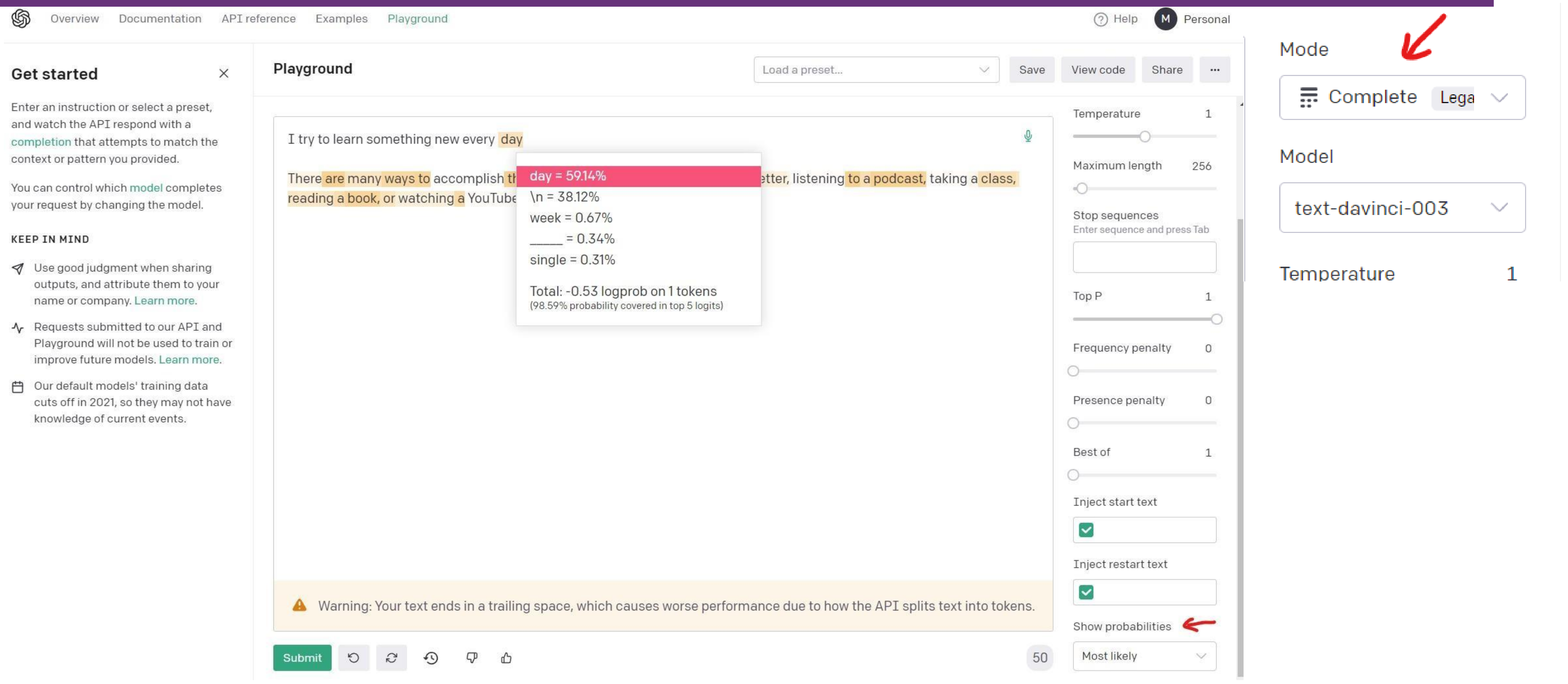
😀

Tokens 2 Characters 2

```
��
```

https://platform.openai.com/tokenizer
https://lunary.ai/openai-tokenizer

Tokens



Overview Documentation API reference Examples Playground

Help M Personal

Get started

Enter an instruction or select a preset, and watch the API respond with a completion that attempts to match the context or pattern you provided.

You can control which model completes your request by changing the model.

KEEP IN MIND

Use good judgment when sharing outputs, and attribute them to your name or company. Learn more.

Requests submitted to our API and Playground will not be used to train or improve future models. Learn more.

Our default models' training data cuts off in 2021, so they may not have knowledge of current events.

Playground

Load a preset... Save View code Share ···

I try to learn something new every day

There are many ways to accomplish th... ...etter, listening to a podcast, taking a class, reading a book, or watching a YouTube...

day = 59.14%
\n = 38.12%
week = 0.67%
_____ = 0.34%
single = 0.31%

Total: -0.53 logprob on 1 tokens
(98.59% probability covered in top 5 logits)

Warning: Your text ends in a trailing space, which causes worse performance due to how the API splits text into tokens.

Submit 50

Temperature 1
Maximum length 256
Stop sequences
Enter sequence and press Tab
Top P 1
Frequency penalty 0
Presence penalty 0
Best of 1
Inject start text ✓
Inject restart text ✓
Show probabilities
Most likely

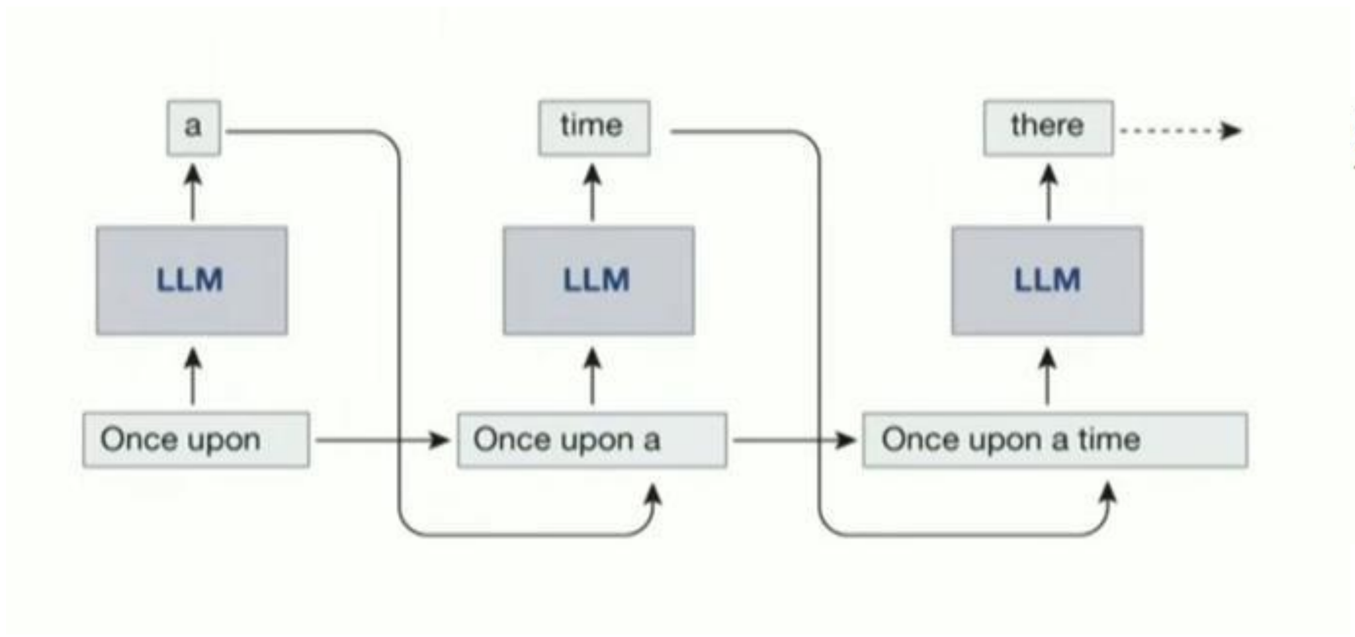
Mode
Complete Lega

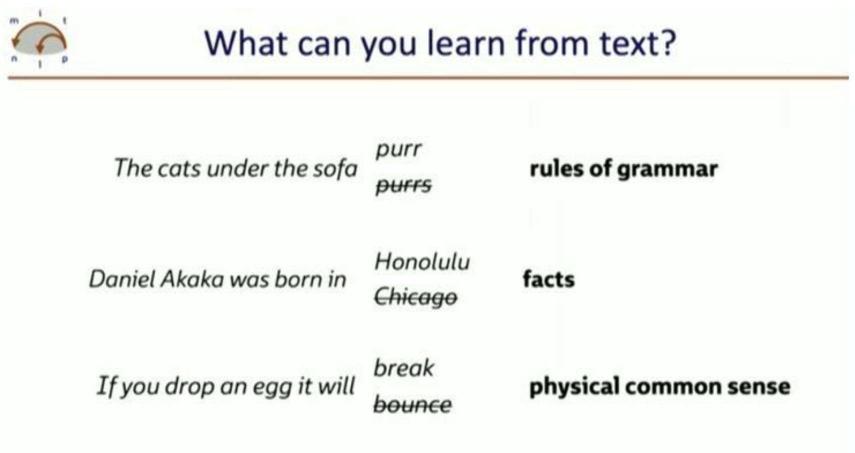
Model
text-davinci-003

Temperature 1

https://platform.openai.com/playground

Tokens and structure



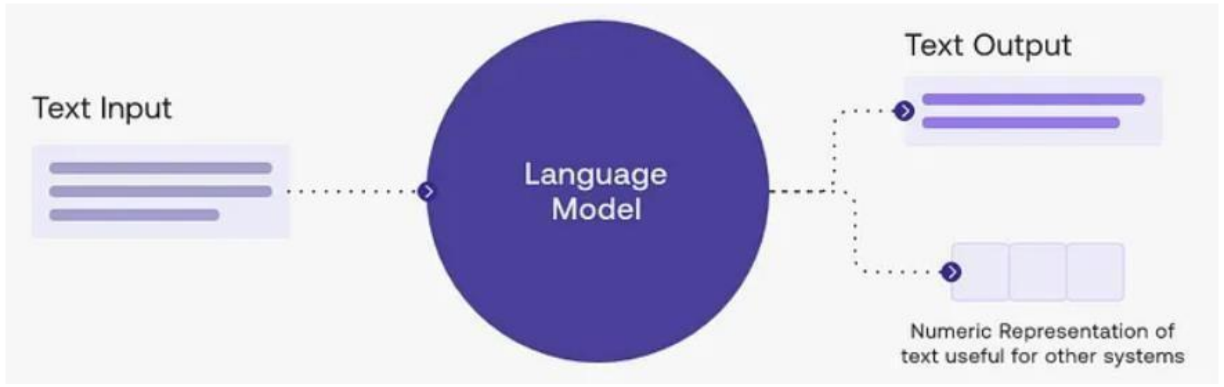

What can you learn from text?

The cats under the sofa	*purr* ~~*purrs*~~	**rules of grammar**
Daniel Akaka was born in	*Honolulu* ~~*Chicago*~~	**facts**
If you drop an egg it will	*break* ~~*bounce*~~	**physical common sense**

Source:
Shanahan M, McDonell K, Reynolds L. Role play with large language models. Nature. 2023 Nov;623(7987):493-498. doi: 10.1038/s41586-023-06647-8 Epub 2023 Nov 8. PMID: 37938776.

Tokens versus parameters

- Large language model (LLM) context
- Token is a basic unit of meaning e.g., word, punctuation mark
- Parameters = numerical values that define model behaviour
- Adjusted during training to optimize model's ability to generate relevant and coherent text



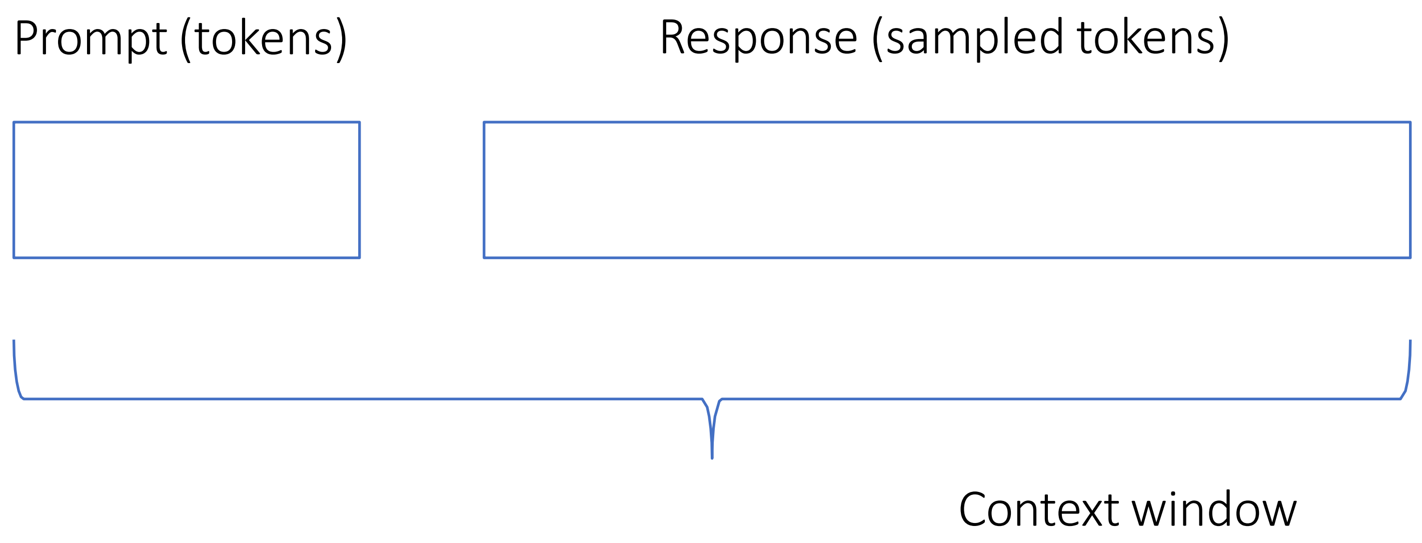
Source: medium.com
https://medium.com/@hmohamedhussain2004/what-is-an-llm-a0086882e585

Large language model

Takes an input and produces a token as an output

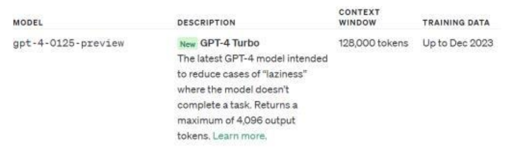
Why are tokens important?

- Context window in large language models
- Length of text model can process and respond to in given instance
- Constraints length of prompt and response



gpt-4 8000 tokens
gpt-4-32k 32000 tokens

gpt-4 6000 words
gpt-4-32k 24000 words

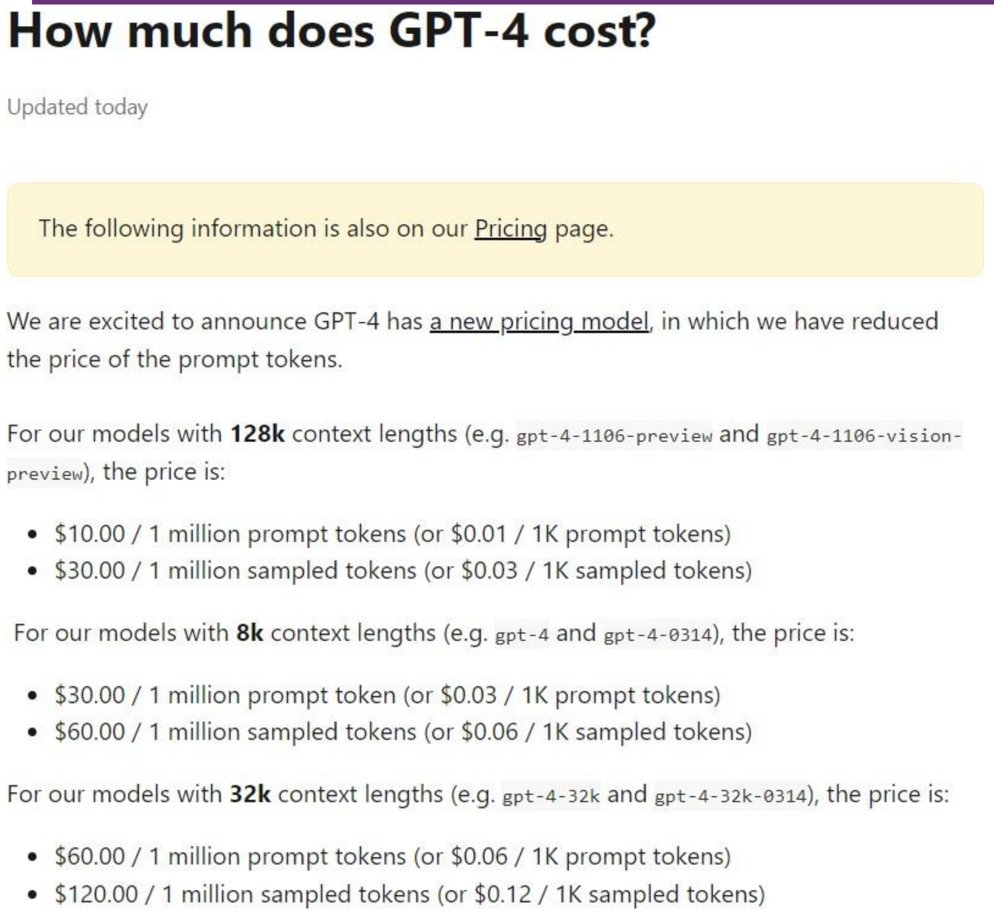


MODEL	DESCRIPTION	CONTEXT WINDOW	TRAINING DATA
gpt-4-0125-preview	New **GPT-4 Turbo** The latest GPT-4 model intended to reduce cases of "laziness" where the model doesn't complete a task. Returns a maximum of 4,096 output tokens. Learn more.	128,000 tokens	Up to Dec 2023

ChatGPT-4 currently has a cap related to message frequency – 128,000 tokens

https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo

Cost is based on tokens used



How much does GPT-4 cost?

Updated today

The following information is also on our Pricing page.

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 **128k** context lengths (e.g. `gpt-4-1106-preview` and `gpt-4-1106-vision-preview`), the price is:

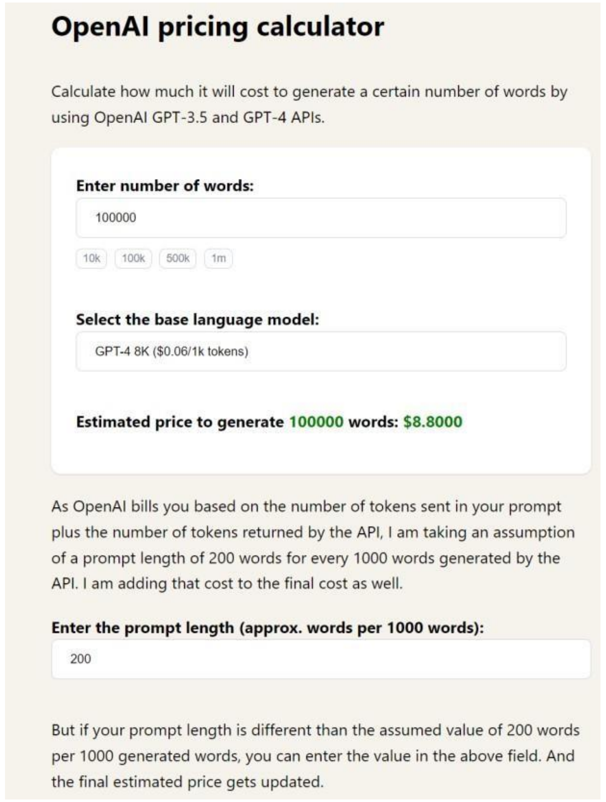
- $10.00 / 1 million prompt tokens (or $0.01 / 1K prompt tokens)
- $30.00 / 1 million sampled tokens (or $0.03 / 1K sampled tokens)

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

- $30.00 / 1 million prompt token (or $0.03 / 1K prompt tokens)
- $60.00 / 1 million sampled tokens (or $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:

- $60.00 / 1 million prompt tokens (or $0.06 / 1K prompt tokens)
- $120.00 / 1 million sampled tokens (or $0.12 / 1K sampled tokens)



OpenAI pricing calculator

Calculate how much it will cost to generate a certain number of words by using OpenAI GPT-3.5 and GPT-4 APIs.

Enter number of words:

100000

Select the base language model:

GPT-4 8K ($0.06/1k tokens)

Estimated price to generate 100000 words: $8.8000

As OpenAI bills you based on the number of tokens sent in your prompt plus the number of tokens returned by the API, I am taking an assumption of a prompt length of 200 words for every 1000 words generated by the API. I am adding that cost to the final cost as well.

Enter the prompt length (approx. words per 1000 words):

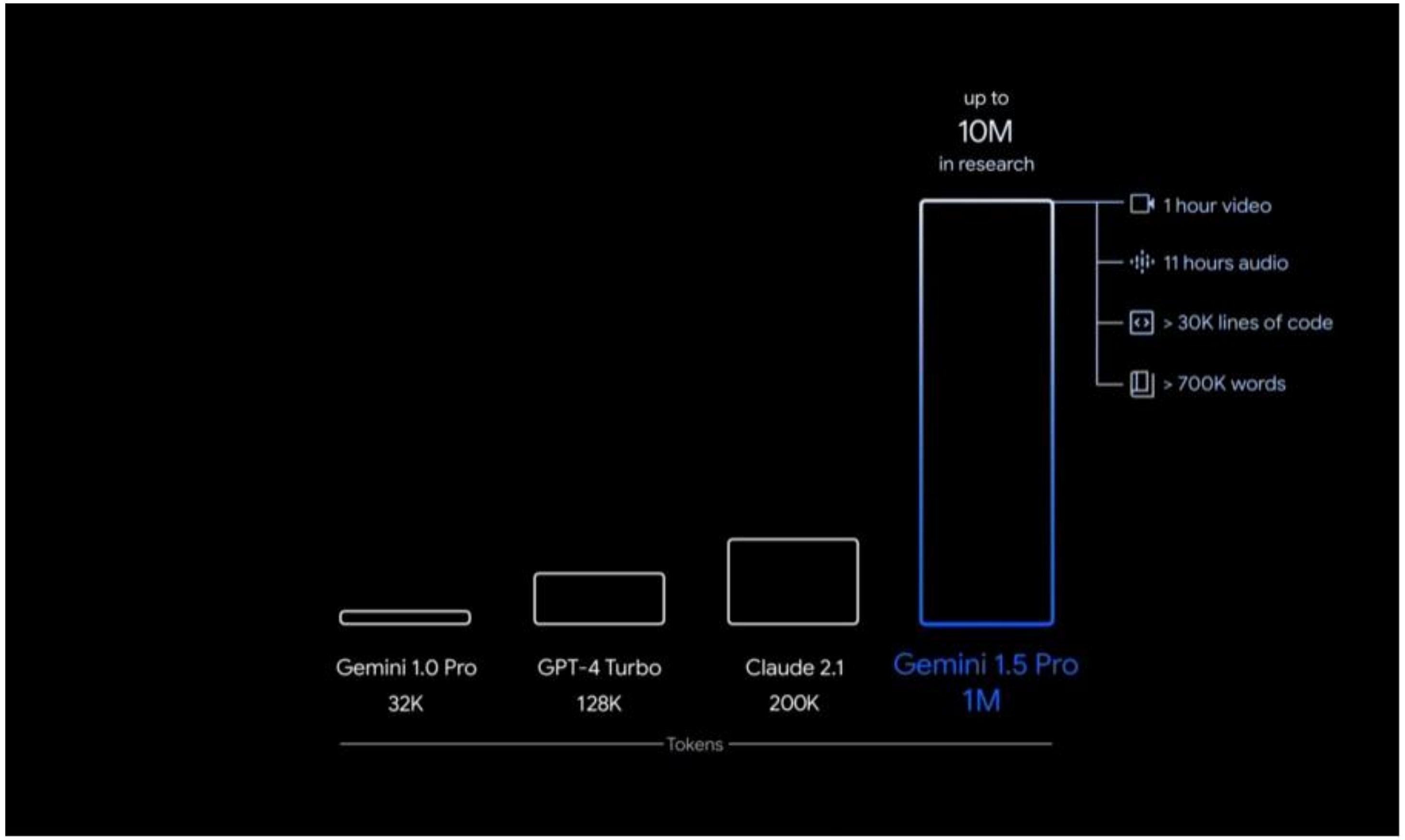
200

But if your prompt length is different than the assumed value of 200 words per 1000 generated words, you can enter the value in the above field. And the final estimated price gets updated.

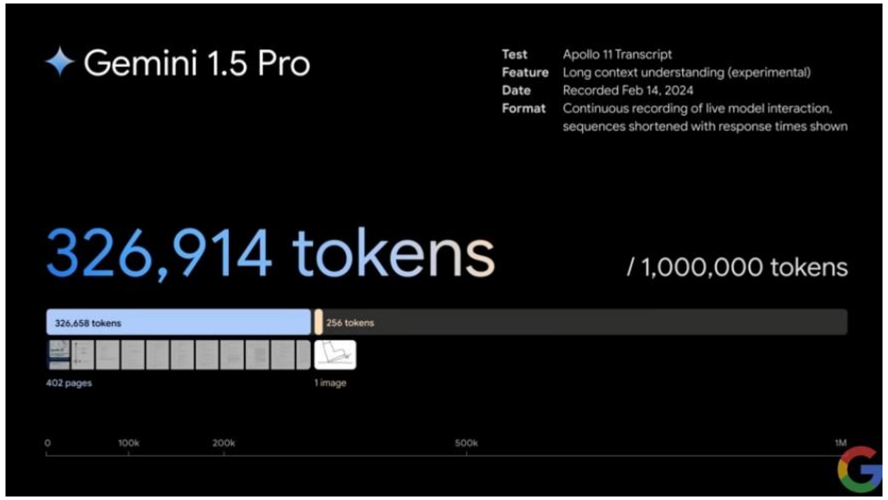
https://help.openai.com/en/articles/7127956-how-much-does-gpt-4-cost

https://invertedstone.com/tools/openai-pricing/

Gemini 1.5 - 1 million multimodal tokens



Context lengths of leading foundation models



Test	Apollo 11 Transcript
Feature	Long context understanding (experimental)
Date	Recorded Feb 14, 2024
Format	Continuous recording of live model interaction, sequences shortened with response times shown

Complex reasoning about vast amounts of information

1.5 Pro can seamlessly analyze, classify and summarize large amounts of content within a given prompt. For example, when given the 402-page transcripts from Apollo 11's mission to the moon, it can reason about conversations, events and details found across the document.

https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/#architecture