BUSINESS INTELLIGENCE AND DATA ANALYTICS

University of Petra

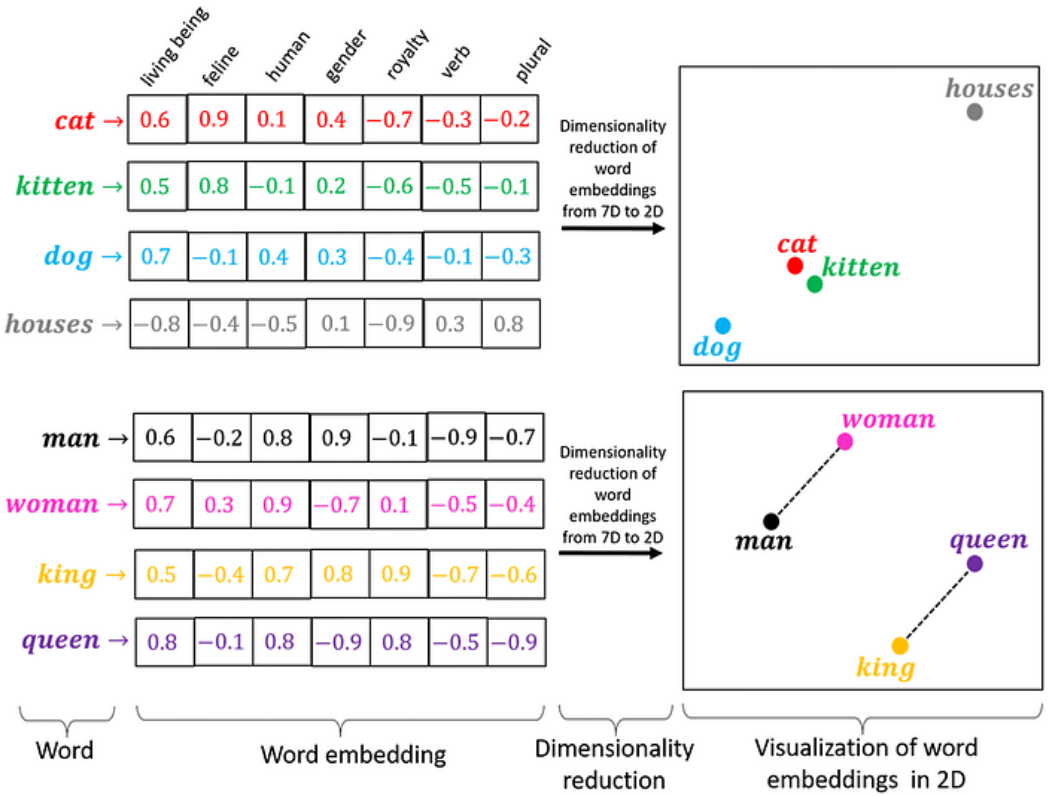
Institute of Analytics Recognized Programme

University of Petra

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.



BUSINESS INTELLIGENCE AND DATA ANALYTICS
– WE PREDICT THE FUTURE –

Institute of Analytics
Recognized Programme

University of Petra

Introduction to Neural Networks

The Perceptron

Outcomes

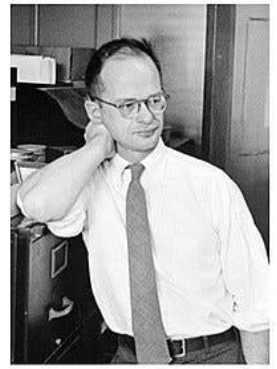
- Fundamentals of neural networks
- Evolution from single perceptrons to MLPs
- Detailed MLP architecture (input, hidden, and output layers)
- Mathematical representations
- Various activation functions (Sigmoid, ReLU, etc.)
- Backpropagation and training methodologies
- Loss functions and optimization 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.

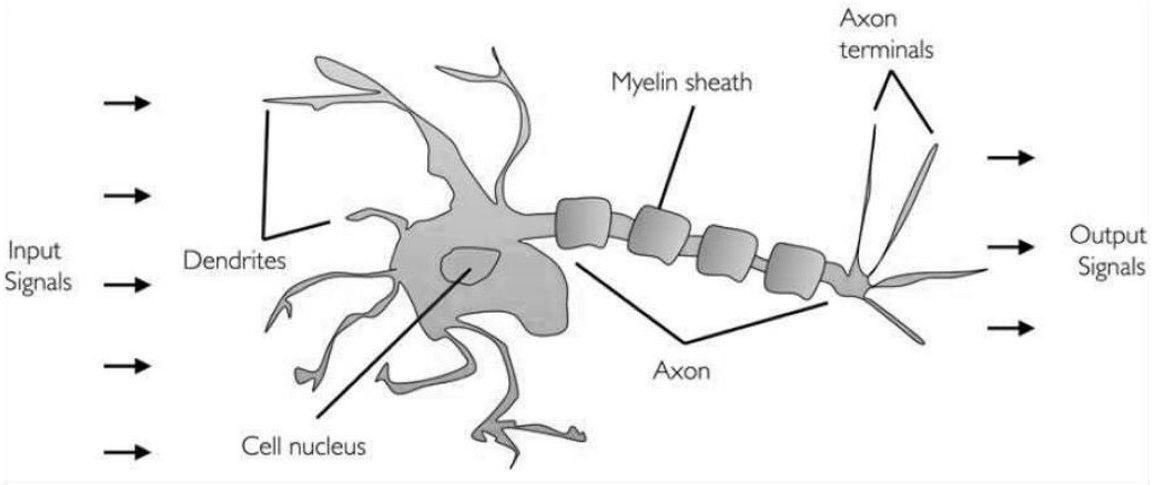
Warren Sturgis McCulloch

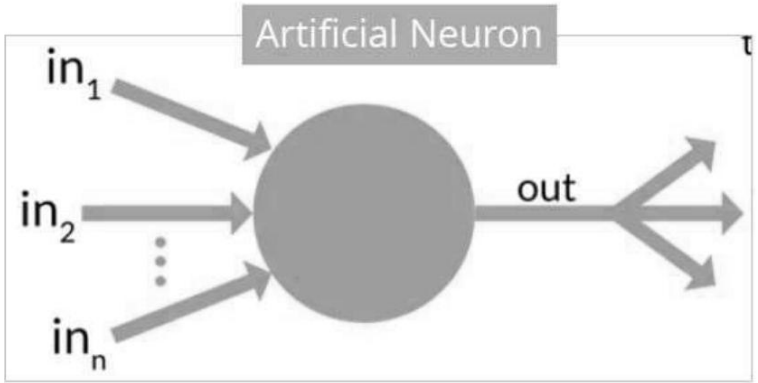
(1898 – 1969)



Walter Harry Pitts, Jr.

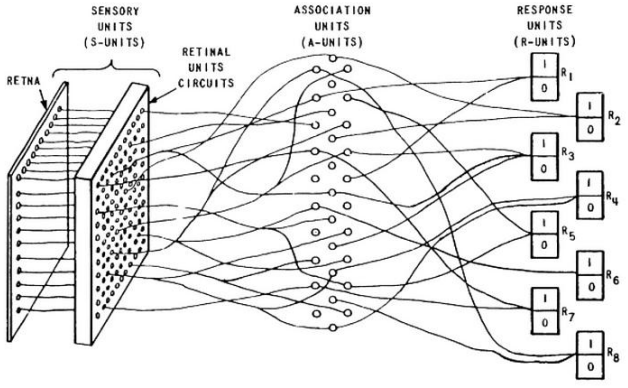
(1923 – 1969)



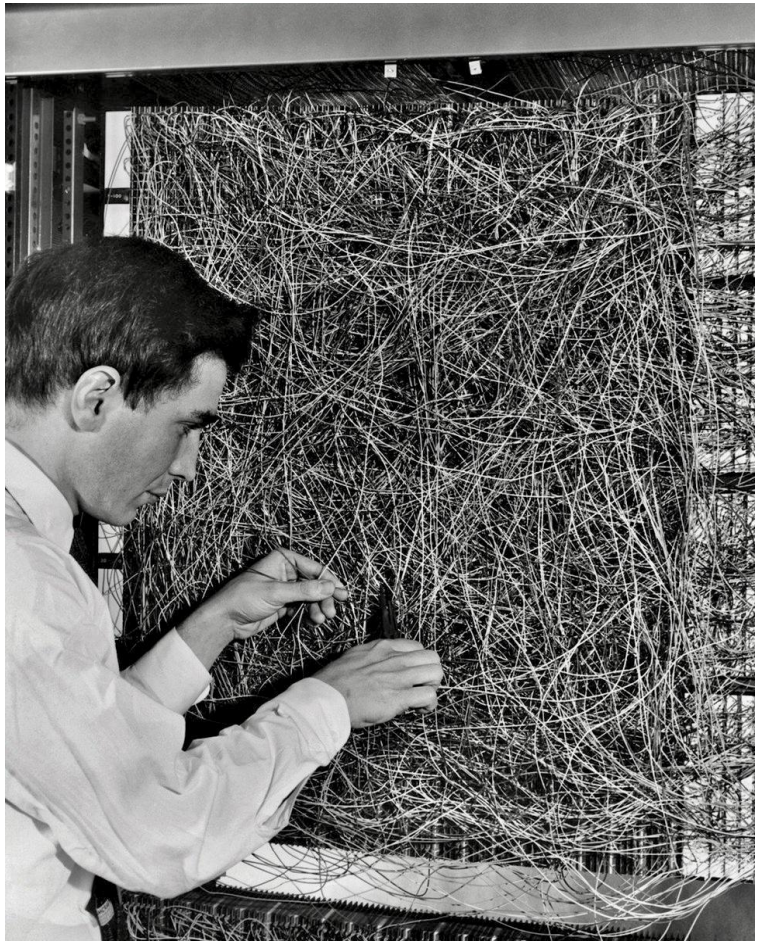


The Perceptron: Building Block of Neural Networks

- 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

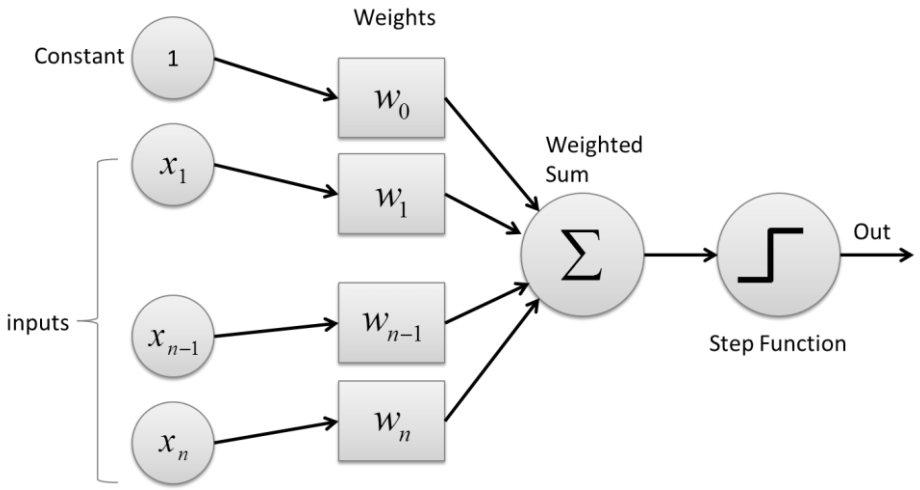


F. Rosenblatt



The Perceptron

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

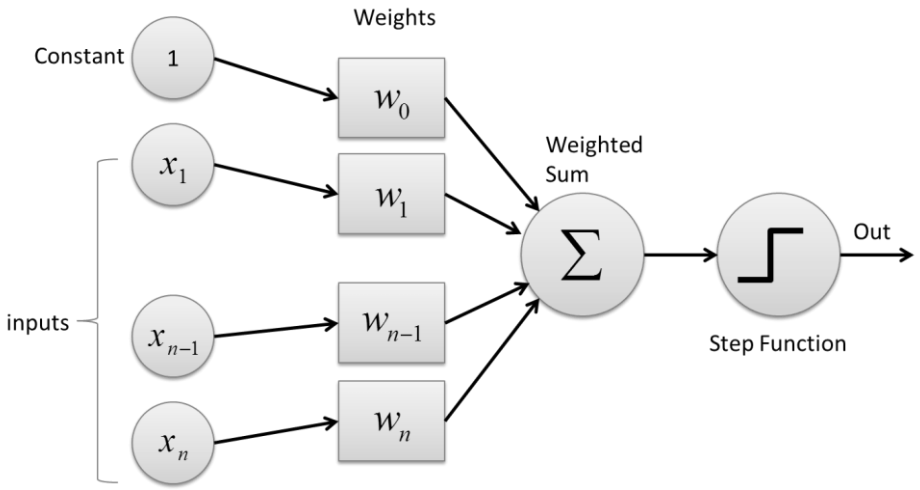


How a Perceptron Works

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

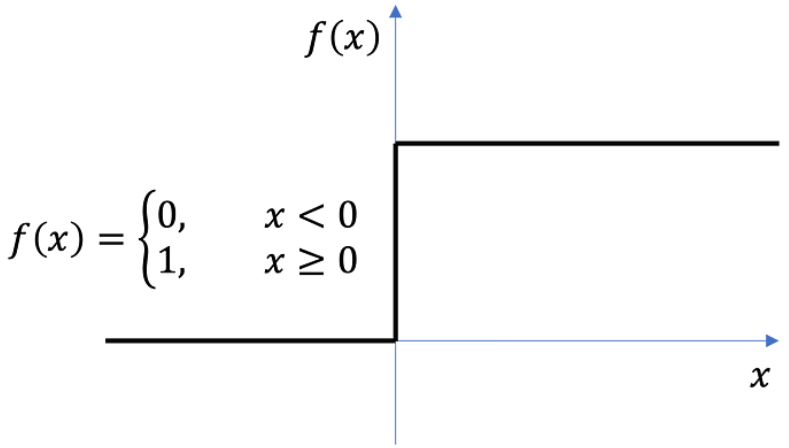
Mathematically:

- $z = w_1x_1 + w_2x_2 + ... + w_nx_n + b$
- 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:
  - $w_1 = w_1 + \eta * error * x_1 = 0.3 + 0.1 * (-1) * 0 = 0.3$
  - $w_2 = w_2 + \eta * error * x_2 = -0.1 + 0.1 * (-1) * 0 = -0.1$
  - $b = b + \eta * error = 0.2 + 0.1 * (-1) = 0.1$

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:
  - $w_1 = w_1 + \eta * error * x_1 = 0.3 + 0.1 * (-1) * 0 = 0.3$
  - $w_2 = w_2 + \eta * error * x_2 = -0.1 + 0.1 * (-1) * 1 = -0.2$
  - $b = b + \eta * error = 0.1 + 0.1 * (-1) = 0$

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:
  - $w_1 = w_1 + \eta * error * x_1 = 0.3 + 0.1 * (-1) * 1 = 0.2$
  - $w_2 = w_2 + \eta * error * x_2 = -0.2 + 0.1 * (-1) * 0 = -0.2$
  - $b = b + \eta * error = 0 + 0.1 * (-1) = -0.1$

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:
  - $w_1 = w_1 + \eta * error * x_1 = 0.2 + 0.1 * 1 * 1 = 0.3$
  - $w_2 = w_2 + \eta * error * x_2 = -0.2 + 0.1 * 1 * 1 = -0.1$
  - $b = b + \eta * error = -0.1 + 0.1 * 1 = 0$

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:
  - $w_1 = w_1 + \eta * error * x_1 = 0.3 + 0.1 * (-1) * 0 = 0.3$
  - $w_2 = w_2 + \eta * error * x_2 = -0.1 + 0.1 * (-1) * 0 = -0.1$
  - $b = b + \eta * error = 0 + 0.1 * (-1) = -0.1$

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
import numpy as np
    class Perceptron:
     def __init__(self, learning_rate=0.01, n_iterations=1000):
         self.learning_rate = learning_rate
         self.n_iterations = n_iterations
         self.weights = None
         self.bias = None

     def fit(self, X, y):
         # Initialize parameters
         n_samples, n_features = X.shape
         self.weights = np.zeros(n_features)
         self.bias = 0

         # Learning
         for _ in range(self.n_iterations):
             for idx, x_i in enumerate(X):
                 linear_output = np.dot(x_i, self.weights) + self.bias
                 y_predicted = 1 if linear_output >= 0 else 0

                 # Perceptron update rule
                 update = self.learning_rate * (y[idx] - y_predicted)
                 self.weights += update * x_i
                 self.bias += update

     def predict(self, X):
         linear_output = np.dot(X, self.weights) + self.bias
         return np.where(linear_output >= 0, 1, 0)
```

```python
import numpy as np
from matplotlib import pyplot as plt

# 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 = Perceptron(learning_rate=0.1,
n_iterations=100)
perceptron.fit(X, y)

# Display results
print("Weights:", perceptron.weights)
print("Bias:", perceptron.bias)
print("Predictions:", perceptron.predict(X))
```

Open in Colab

We can use Python and Sklearn to implement the steps above quickly

```
import numpy as np from sklearn.linear_model
import Perceptron import matplotlib.pyplot as plt

# 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 the perceptron
perceptron = Perceptron(max_iter=100, tol=1e-3,
random_state=42)
perceptron.fit(X, y)

# Test the perceptron
print("Weights:", perceptron.coef_)
print("Bias:", perceptron.intercept_)
print("Predictions:", perceptron.predict(X))
```

```
Weights: [[2. 2.]]
Bias: [-3.]
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: [[2. 2.]]** - The perceptron learned to assign a weight of 2.0 to both inputs
- **Bias: [-3.]** - The bias is -3.0
- **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: 2×(input1) + 2×(input2) - 3

For the four input combinations:

- [0,0]: 2×0 + 2×0 - 3 = -3 < 0 → output 0
- [0,1]: 2×0 + 2×1 - 3 = -1 < 0 → output 0
- [1,0]: 2×1 + 2×0 - 3 = -1 < 0 → output 0
- [1,1]: 2×1 + 2×1 - 3 = 1 > 0 → output 1

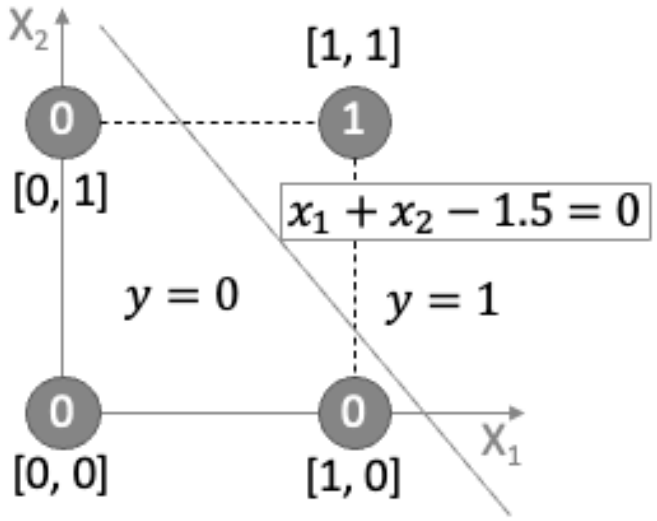
This perceptron implements the AND gate logic.

The decision boundary is the line $2x_1 + 2x_2 - 3 = 0$, which separates the point (1,1) from the other three points.

Decision Boundary

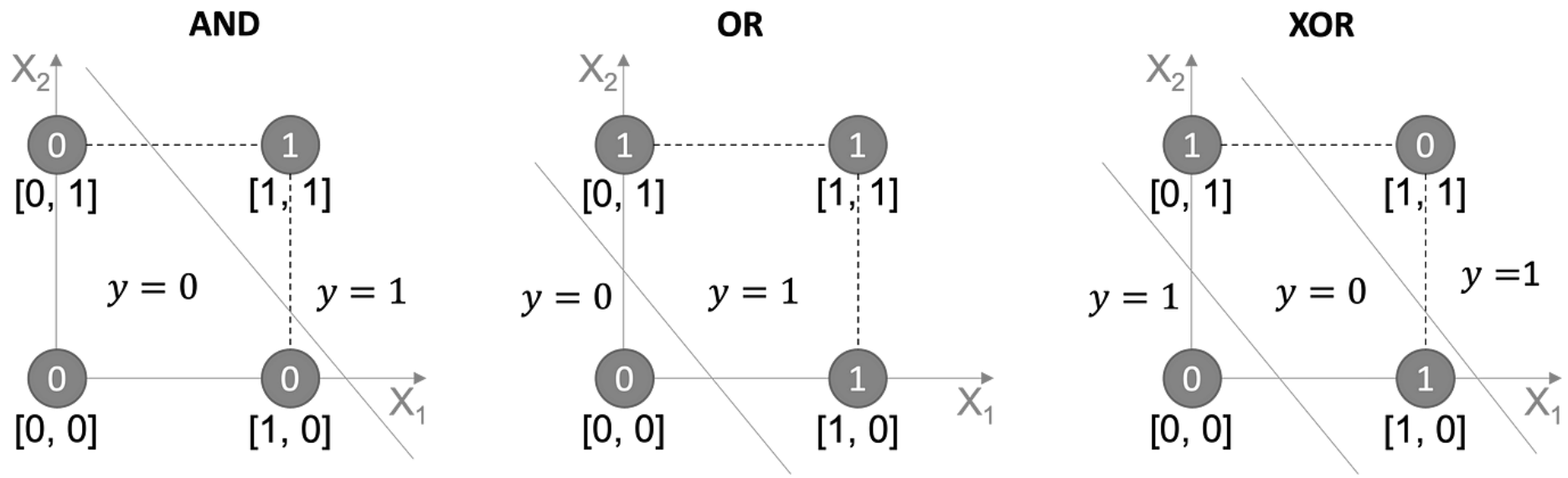
- The perceptron learns a decision boundary: $w_1x_1 + w_2x_2 + b = 0$
- Points above the line are classified as 1
- Points below the line are classified as 0
- For AND gate, only the point (1,1) should be above the line

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



BUSINESS INTELLIGENCE AND DATA ANALYTICS
– WE PREDICT THE FUTURE –

Institute of Analytics
Recognized Programme

University of Petra

Introduction to Neural Networks

Multi-Layer Neural Network

The Multi-Layer Perceptron (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.“

Enter 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

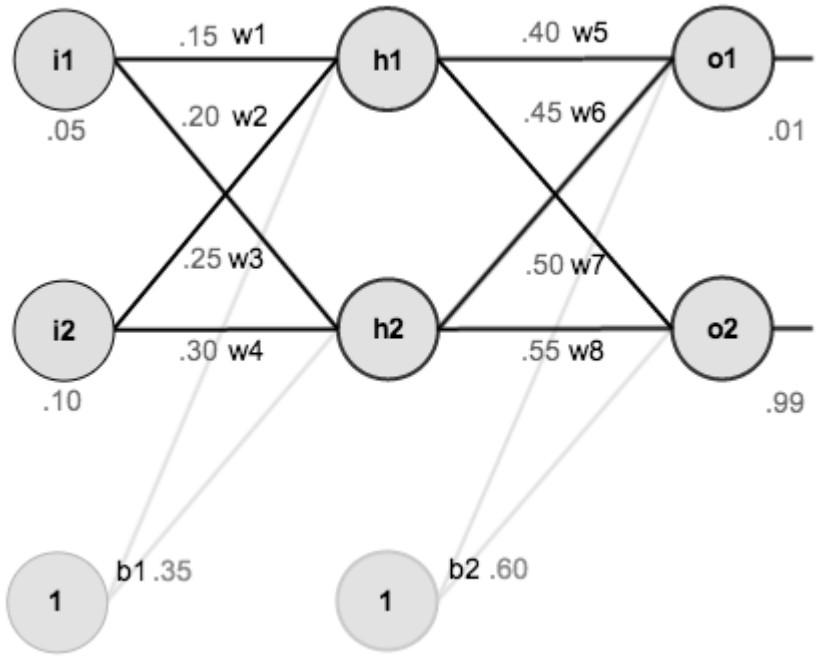
- Receives the raw input features
- One neuron per input feature
- No computation occurs here; inputs are simply passed forward

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

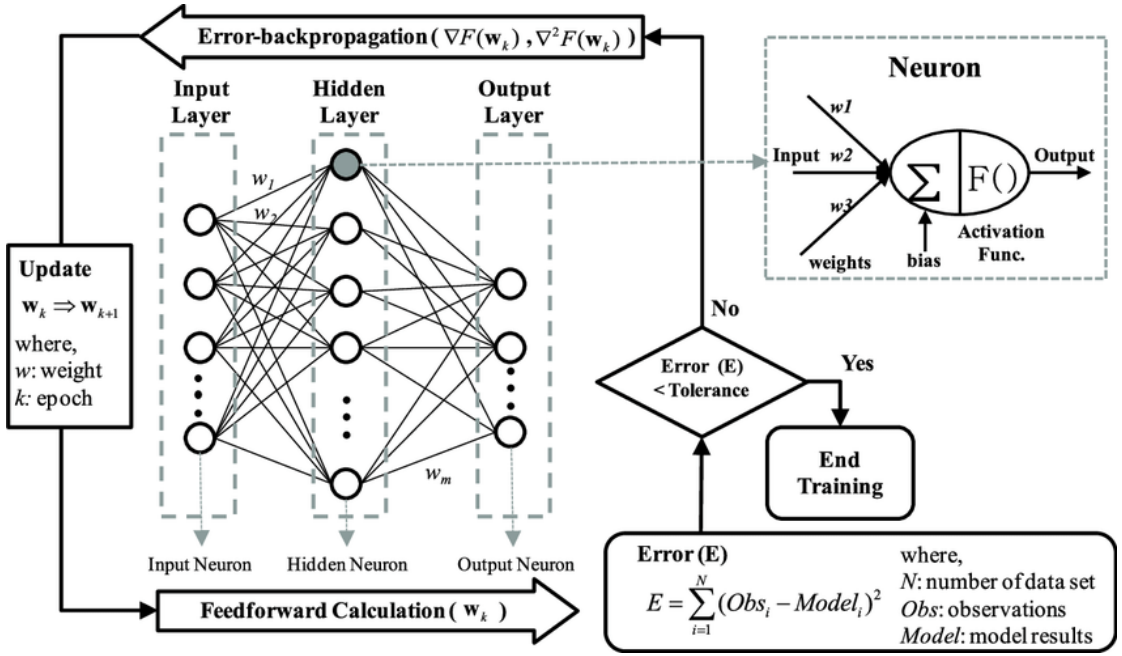
- Produces the final prediction or classification
- Structure depends on the task:
    - Regression: Often a single neuron with linear activation
    - Binary classification: One neuron with sigmoid activation
    - Multi-class classification: Multiple neurons (one per class) with softmax activation



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

Manual Computation for One Forward-Backward Pass

Problem Setup

We are solving a basic 2-2-1 neural network by hand:

- Input vector:

$$\mathbf{x} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad y = 1$$

- Hidden layer weights $W^{[1]} \in \mathbb{R}^{2\times 2}$:

$$W^{[1]} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix}$$

- Hidden layer bias $\mathbf{b}^{[1]} \in \mathbb{R}^2$:

$$\mathbf{b}^{[1]} = \begin{bmatrix} 0.05 \\ 0.05 \end{bmatrix}$$

- Output layer weights $W^{[2]} \in \mathbb{R}^{2\times 1}$:

$$W^{[2]} = \begin{bmatrix} 0.5 \\ 0.6 \end{bmatrix}$$

- Output layer bias $b^{[2]} \in \mathbb{R}$:

$$b^{[2]} = 0.1$$

Forward Propagation

Step 1: Hidden Layer Linear Combination

$$\mathbf{z}^{[1]} = W^{[1]} \cdot \mathbf{x} + \mathbf{b}^{[1]} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 0.05 \\ 0.05 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.4 \end{bmatrix} + \begin{bmatrix} 0.05 \\ 0.05 \end{bmatrix} = \begin{bmatrix} 0.25 \\ 0.45 \end{bmatrix}$$

Step 2: Hidden Layer Activation

Apply sigmoid:

$$\mathbf{a}^{[1]} = \sigma(\mathbf{z}^{[1]}) = \begin{bmatrix} \sigma(0.25) \\ \sigma(0.45) \end{bmatrix} \approx \begin{bmatrix} 0.562 \\ 0.610 \end{bmatrix}$$

Step 3: Output Layer Linear Combination

$$z^{[2]} = (W^{[2]})^\top \cdot \mathbf{a}^{[1]} + b^{[2]} = \begin{bmatrix} 0.5 & 0.6 \end{bmatrix} \begin{bmatrix} 0.562 \\ 0.610 \end{bmatrix} + 0.1 = 0.281 + 0.366 + 0.1 = 0.747$$

Step 4: Output Activation

$$\hat{y} = \sigma(z^{[2]}) \approx \sigma(0.747) \approx 0.678$$

Backward Propagation

Step 5: Output Error

$$\delta^{[2]} = \frac{\partial \mathcal{L}}{\partial z^{[2]}} = (\hat{y} - y) \cdot \sigma'(z^{[2]})$$

First compute:

$$\hat{y} - y = 0.678 - 1 = -0.322$$

Then:

$$\sigma'(z^{[2]}) = \hat{y}(1 - \hat{y}) = 0.678(1 - 0.678) \approx 0.678 \cdot 0.322 \approx 0.218$$

So:

$$\delta^{[2]} = -0.322 \cdot 0.218 \approx -0.070$$

Step 6: Gradients for Output Layer

Gradient w.r.t. weights:

$$\frac{\partial \mathcal{L}}{\partial W^{[2]}} = \delta^{[2]} \cdot \mathbf{a}^{[1]} = -0.070 \cdot \begin{bmatrix} 0.562 \\ 0.610 \end{bmatrix} \approx \begin{bmatrix} -0.0393 \\ -0.0427 \end{bmatrix}$$

Gradient w.r.t. bias:

$$\frac{\partial \mathcal{L}}{\partial b^{[2]}} = \delta^{[2]} = -0.070$$

Step 7: Backpropagate to Hidden Layer

$$\delta^{[1]} = \sigma'(\mathbf{z}^{[1]}) \circ \left(W^{[2]} \cdot \delta^{[2]} \right)$$

Compute:

$$\sigma'(0.25) = 0.562(1 - 0.562) \approx 0.246$$
$$\sigma'(0.45) = 0.610(1 - 0.610) \approx 0.238$$

$$W^{[2]} \cdot \delta^{[2]} = \begin{bmatrix} 0.5 \\ 0.6 \end{bmatrix} \cdot (-0.070) = \begin{bmatrix} -0.035 \\ -0.042 \end{bmatrix}$$

Then:

$$\delta^{[1]} = \begin{bmatrix} 0.246 \\ 0.238 \end{bmatrix} \circ \begin{bmatrix} -0.035 \\ -0.042 \end{bmatrix} \approx \begin{bmatrix} -0.0086 \\ -0.0100 \end{bmatrix}$$

Backward Propagation

Step 8: Gradients for Hidden Layer

Only $x_2 = 1$, so we update the **second row** of $W^{[1]}$:

$$\frac{\partial \mathcal{L}}{\partial W^{[1]}_{2,:}} = \delta^{[1]} \cdot x_2 = \begin{bmatrix} -0.0086 \\ -0.0100 \end{bmatrix} \cdot 1 = \begin{bmatrix} -0.0086 \\ -0.0100 \end{bmatrix}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[1]}} = \delta^{[1]} = \begin{bmatrix} -0.0086 \\ -0.0100 \end{bmatrix}$$

Gradient Descent Updates (η = 0.5)

Update Output Layer:

$$W^{[2]} \leftarrow W^{[2]} - \eta \cdot \frac{\partial \mathcal{L}}{\partial W^{[2]}} = \begin{bmatrix} 0.5 \\ 0.6 \end{bmatrix} - 0.5 \cdot \begin{bmatrix} -0.0393 \\ -0.0427 \end{bmatrix} \approx \begin{bmatrix} 0.5196 \\ 0.6214 \end{bmatrix}$$

$$b^{[2]} \leftarrow 0.1 - 0.5 \cdot (-0.070) = 0.135$$

Update Hidden Layer (only second row):

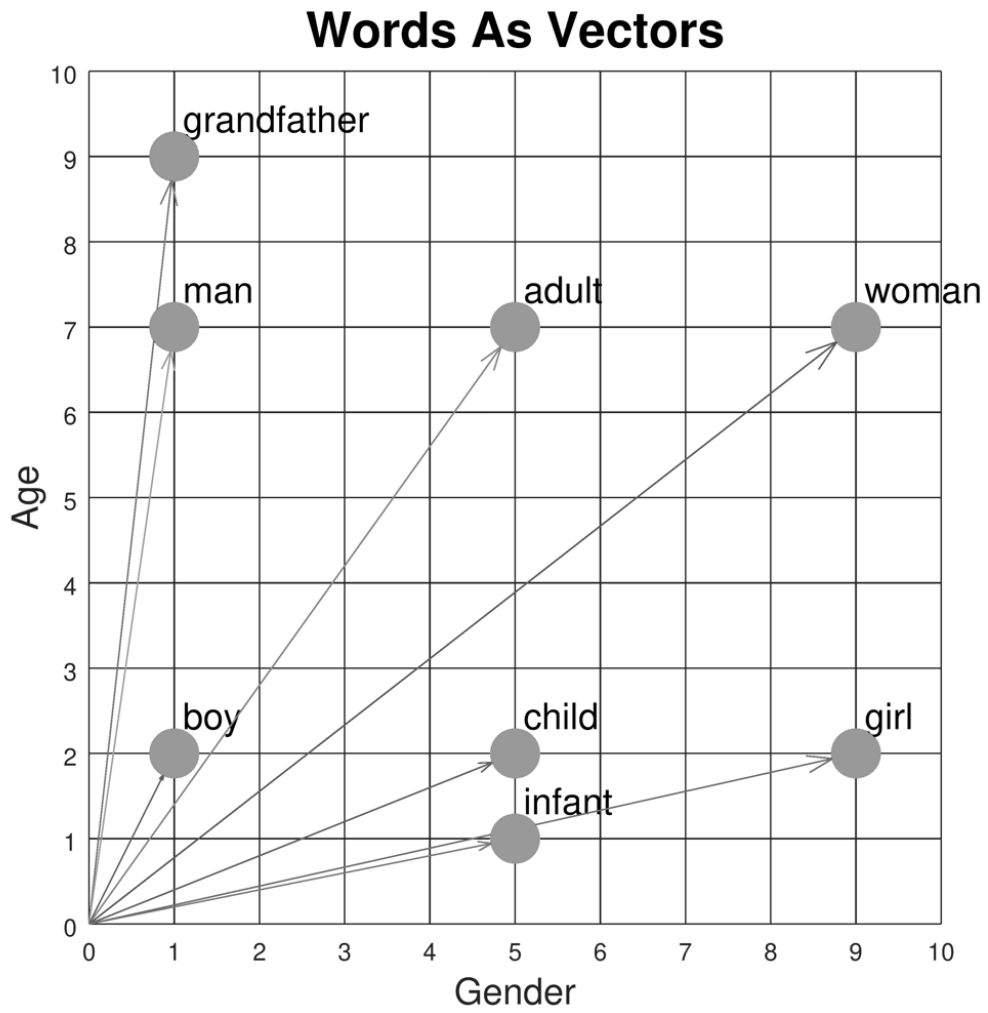
$$W^{[1]}_{2,:} \leftarrow W^{[1]}_{2,:} - 0.5 \cdot \begin{bmatrix} -0.0086 & -0.0100 \end{bmatrix} = \begin{bmatrix} 0.3 & 0.4 \end{bmatrix} + \begin{bmatrix} 0.0043 & 0.0050 \end{bmatrix} = \begin{bmatrix} 0.3043 & 0.4050 \end{bmatrix}$$

$$\mathbf{b}^{[1]} \leftarrow \mathbf{b}^{[1]} - 0.5 \cdot \begin{bmatrix} -0.0086 \\ -0.0100 \end{bmatrix} = \begin{bmatrix} 0.05 + 0.0043 \\ 0.05 + 0.0050 \end{bmatrix} = \begin{bmatrix} 0.0543 \\ 0.0550 \end{bmatrix}$$

Introduction to Word Embeddings

Introduction to Word Embeddings

- **Definition**: Word embeddings are vector representations of words in a continuous vector space.
- **Purpose**: Capture semantic and syntactic relationships between words.
- **Key innovation**: Words with similar meanings have similar vector representations.
- **Foundation of modern NLP**: Enable machines to understand relationships between words.



The Evolution of Word Representations

Traditional Methods (Pre-2013)

- **One-hot encoding**: Sparse, binary vectors (dimension = vocabulary size)
- **Limitations**:
  - No notion of similarity between words
  - Curse of dimensionality
  - No semantic information captured

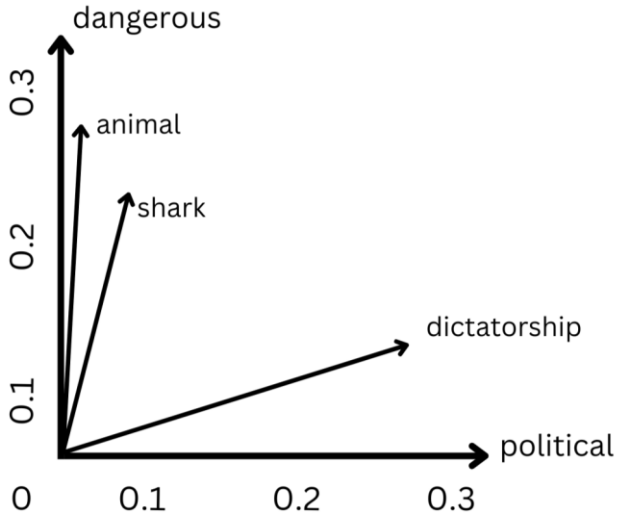
Word	Dimension 1 (cat)	Dimension 2 (dog)	Dimension 3 (fish)	Dimension 4 (bird)
cat	1	0	0	0
dog	0	1	0	0
fish	0	0	1	0
bird	0	0	0	1

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)

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



- Words are represented as dense vectors in a continuous vector space
- Each dimension potentially captures semantic meaning
- Similar words cluster together in the vector space
- Semantic relationships are preserved (e.g., "shark" is closer to "dangerous" than "political")
- Enables meaningful similarity measurements and analogies

Word Similarities

```
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"))
```

Word2Vec (Mikolov et al., 2013)

Key Innovation

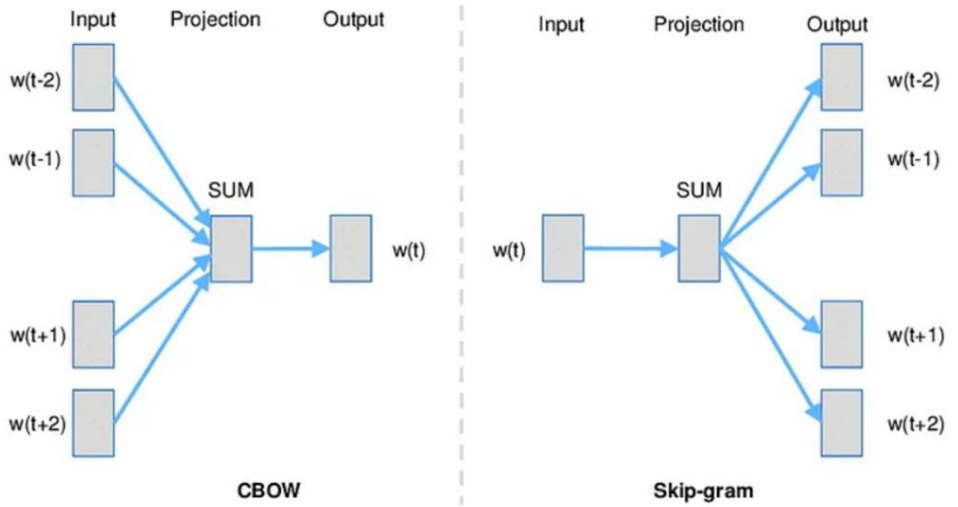
- Transformed NLP by creating dense vector representations through prediction-based models

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

- 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



Real Embeddings

- pip install spacy
- python -m spacy download en_core_web_md

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

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

```
import gensim
from gensim.models import Word2Vec
import nltk
from nltk.tokenize import word_tokenize
nltk.download('punkt')
 # Sample corpus
sentences = [
   "Large language models are transforming business applications",
   "Natural language processing helps computers understand human language",
   "Word embeddings capture semantic relationships between words",
   "Neural networks learn distributed representations of words",
   "Businesses use language models for various applications",
   "Customer service can be improved with language technology",
   "Modern language models require significant computing resources",
   "Language models can generate human-like text for businesses"]
# Tokenize the sentences
tokenized_sentences = [word_tokenize(sentence.lower()) for sentence in sentences]
# Train Word2Vec model
model = Word2Vec(sentences=tokenized_sentences,
                 vector_size=100,  # Embedding dimension
                 window=5,        # Context window size
                 min_count=1,     # Minimum word frequency
                 workers=4)       # Number of threads
# Save the model
model.save("word2vec.model")

# Find the most similar words to "language"
similar_words = model.wv.most_similar("language", topn=5)
print("Words most similar to 'language':")
for word, similarity in similar_words:
   print(f"{word}: {similarity:.4f}")

# Vector for a specific word
word_vector = model.wv["business"]
print(f"\nVector for 'business' (first 10 dimensions):\n{word_vector[:10]}")

# Word analogies
analogy_result = model.wv.most_similar(positive=["business", "language"],
                                       negative=["models"],
                                       topn=3)
print("\nAnalogy results:")
for word, similarity in analogy_result:
   print(f"{word}: {similarity:.4f}")
```

Practical Implementation: Word2Vec in Python

```
import gensim.downloader as api
from gensim.models import Word2Vec
import numpy as np

# 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
```

Practical Implementation: GloVe in Python

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
# Function to load GloVe vectors
def load_glove_vectors(file_path):
    embeddings_dict = {}
    with open(file_path, 'r', encoding='utf-8') as f:
        for line in f:
            values = line.split()
            word = values[0]
            vector = np.array(values[1:], dtype='float32')
            embeddings_dict[word] = vector
    return embeddings_dict
# Load pre-trained GloVe vectors
glove_vectors = load_glove_vectors('glove.6B.100d.txt')
# Calculate word similarity
def get_similarity(word1, word2, embeddings):
    if word1 in embeddings and word2 in embeddings:
        vec1 = embeddings[word1].reshape(1, -1)
        vec2 = embeddings[word2].reshape(1, -1)
        return cosine_similarity(vec1, vec2)[0][0]
    return None
# Example usage
similarity = get_similarity('king', 'queen', glove_vectors)
print(f"Similarity between 'king' and 'queen': {similarity:.4f}")
```

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()
```

BUSINESS INTELLIGENCE AND DATA ANALYTICS
– WE PREDICT THE FUTURE –

Institute of Analytics
Recognized Programme

University of Petra

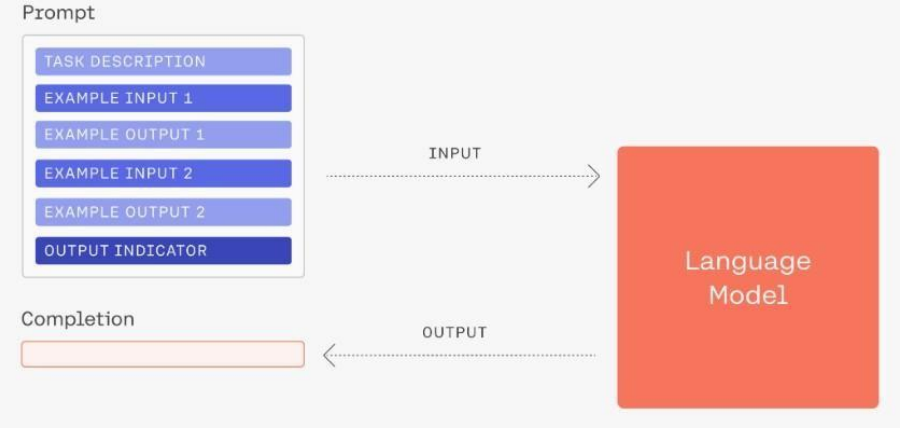
GenAI and Large Language Models

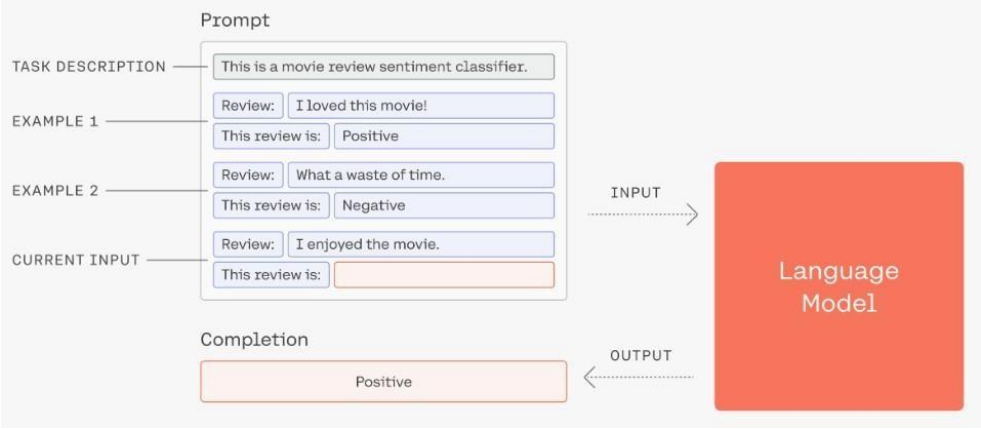
Overview

- What is generative AI
- Large language models
- Tokens
- Tokens versus parameters
- Prompt engineering
- Zero-shot, one-shot and few-shot learning
- Prompt guide
- Generative AI systems examples
- Large language models – getting started
- OpenAI – ChatGPT
- OpenAI tools
- ChatGPT command types
- ChatGPT - plugins
- What about the students?
- Office 365 Copilot – Windows 11 Copilot
- What are our options?
- What about Turnitin?
- Practical use in learning and teaching
- Assessment redesign for generative AI
- Resources and further reading

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

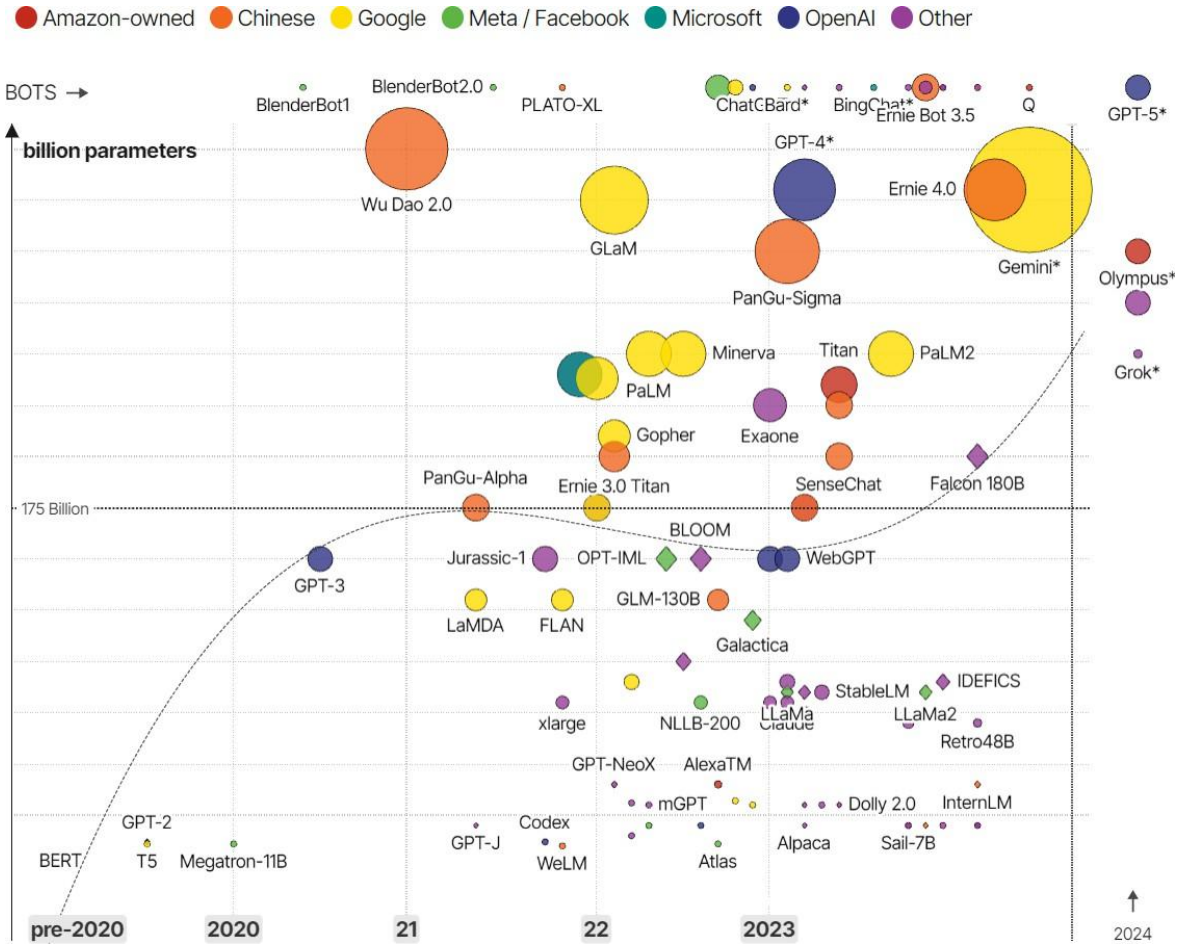
Large language models

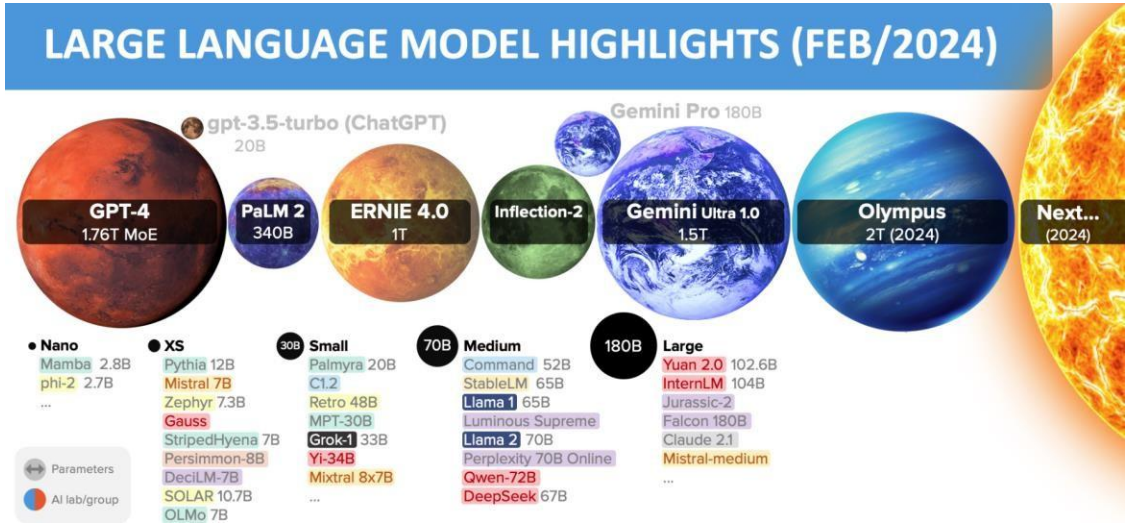
- Type of machine learning model/algorithm
- Performs variety of natural language processing (NLP) tasks
- Learn, understand, and process human language efficiently
- E.g., generate/classify text, answer questions conversationally
- Uses hundreds of billions parameters

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)





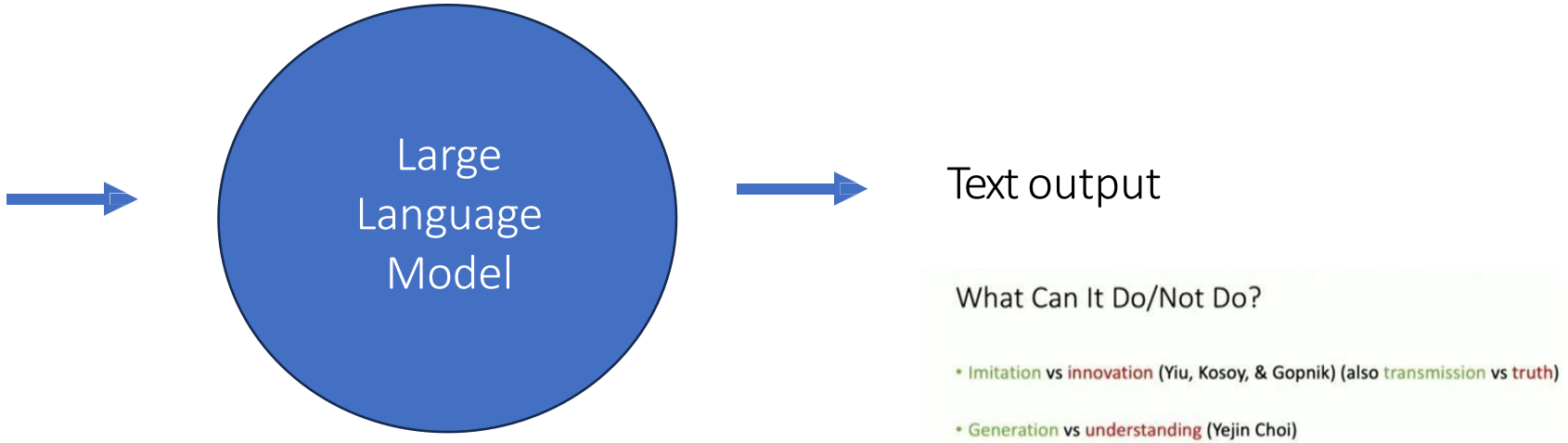
Source: https://lifearchitect.ai/models/

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

Large language models training

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

- Wikipedia
- Common crawl
- Other sources



What Can It Do/Not Do?

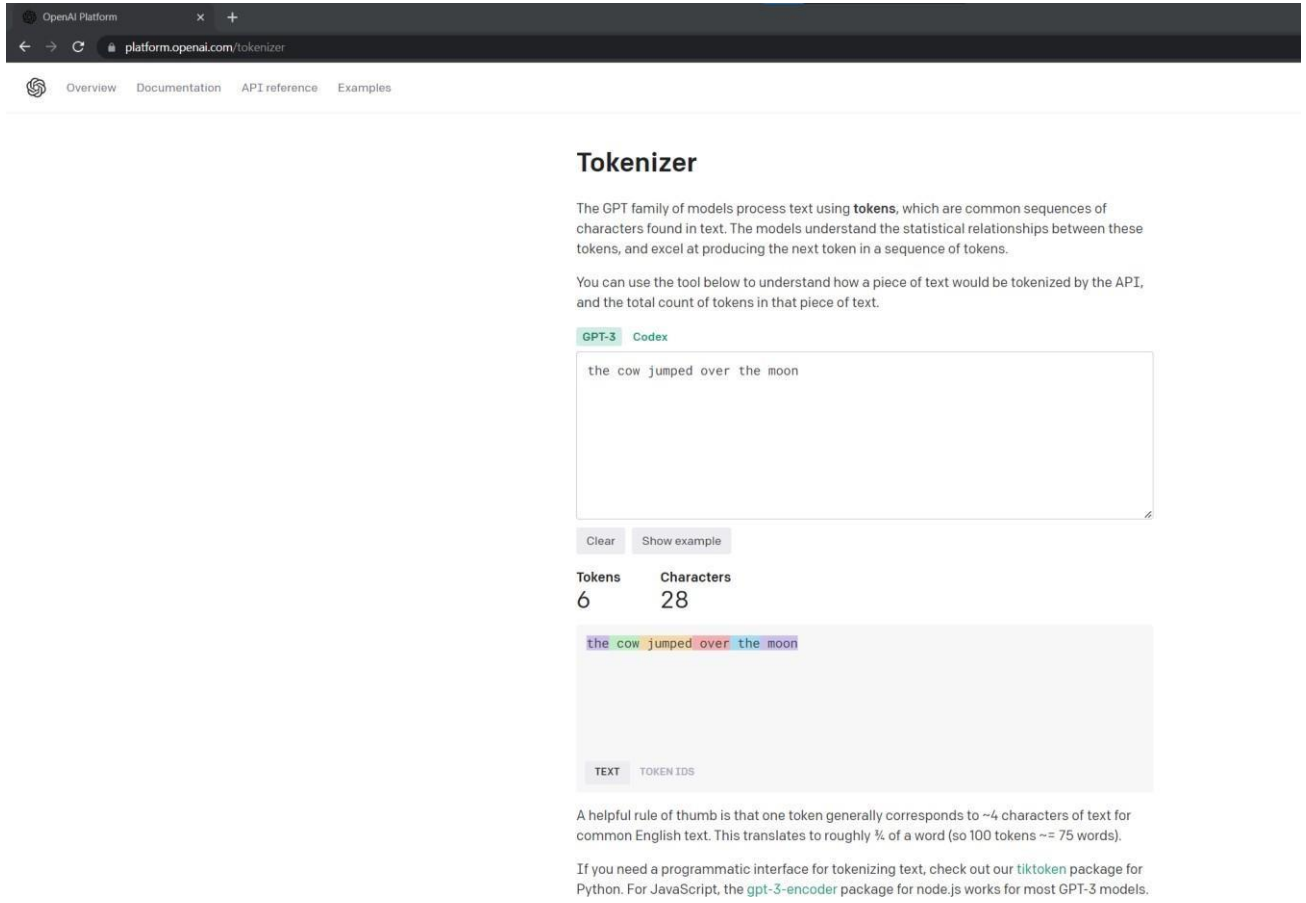
- Imitation vs innovation (Yiu, Kosoy, & Gopnik) (also transmission vs truth)
- Generation vs understanding (Yejin Choi)
- Memorization vs reasoning (Melanie Mitchell)
- Reacting vs planning (Yann LeCun)

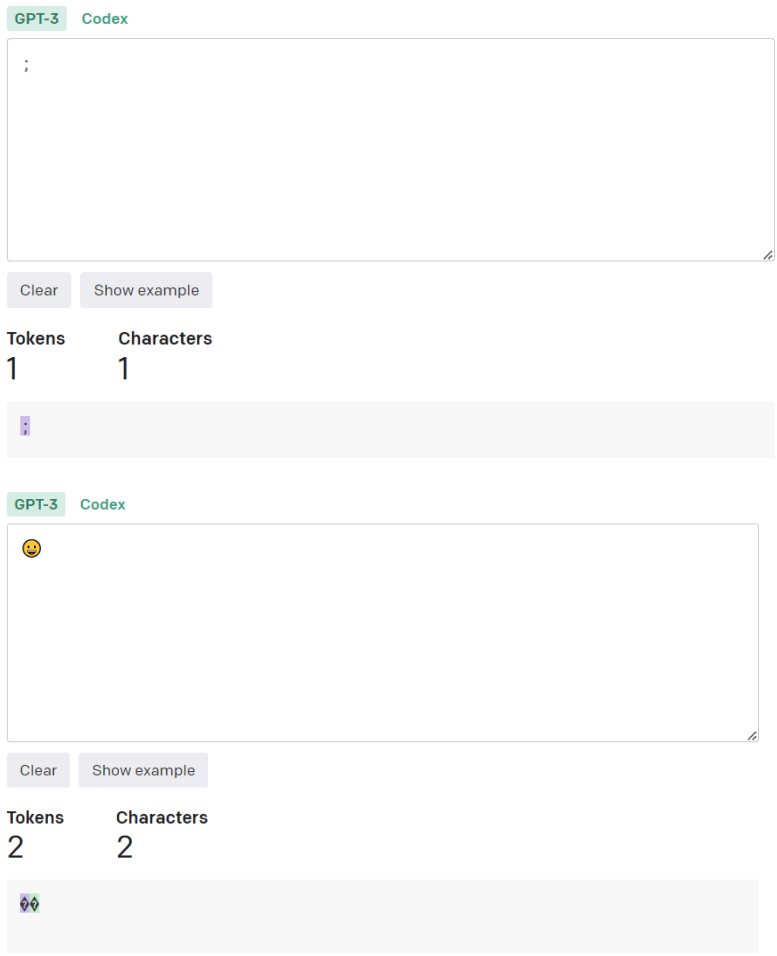
Source: informationisbeautiful.net
Generative AI +: Education
https://www.youtube.com/@mit

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

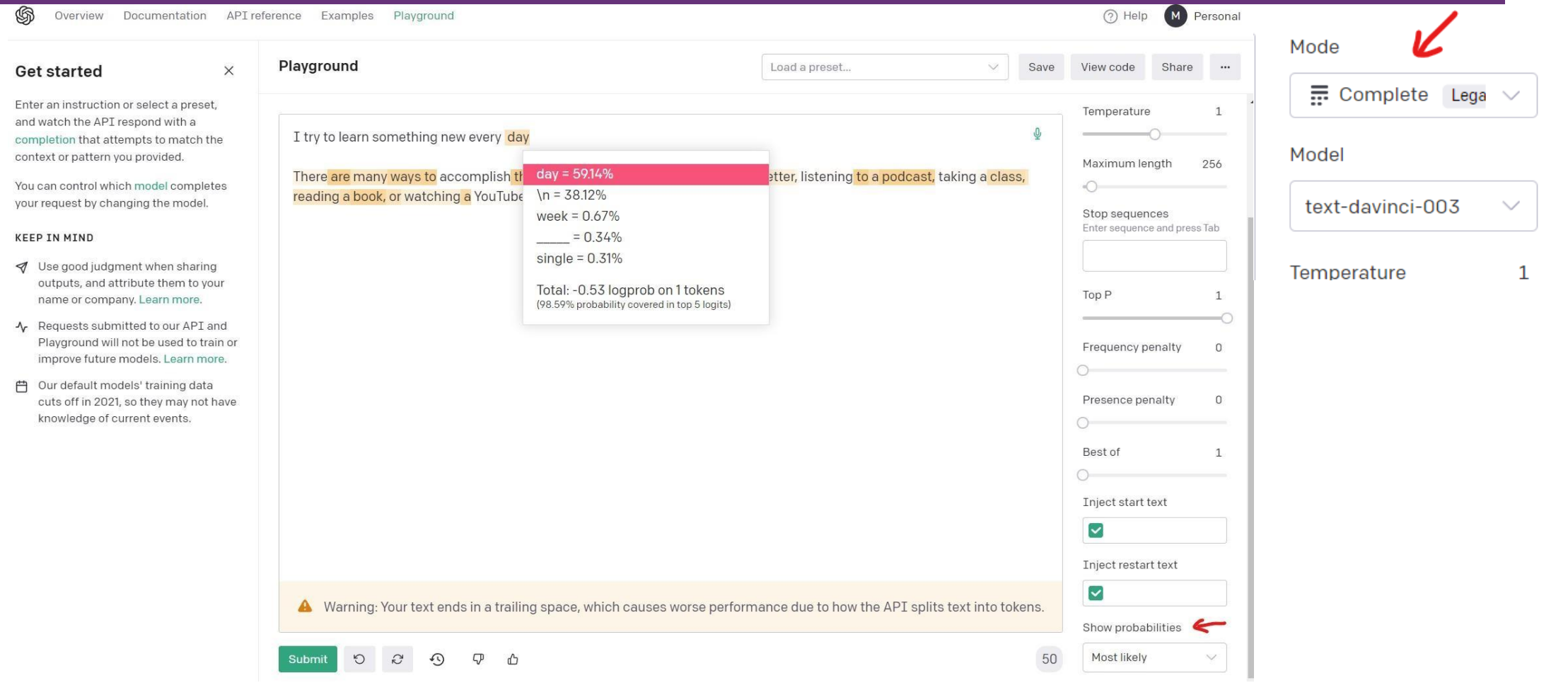
Tokens





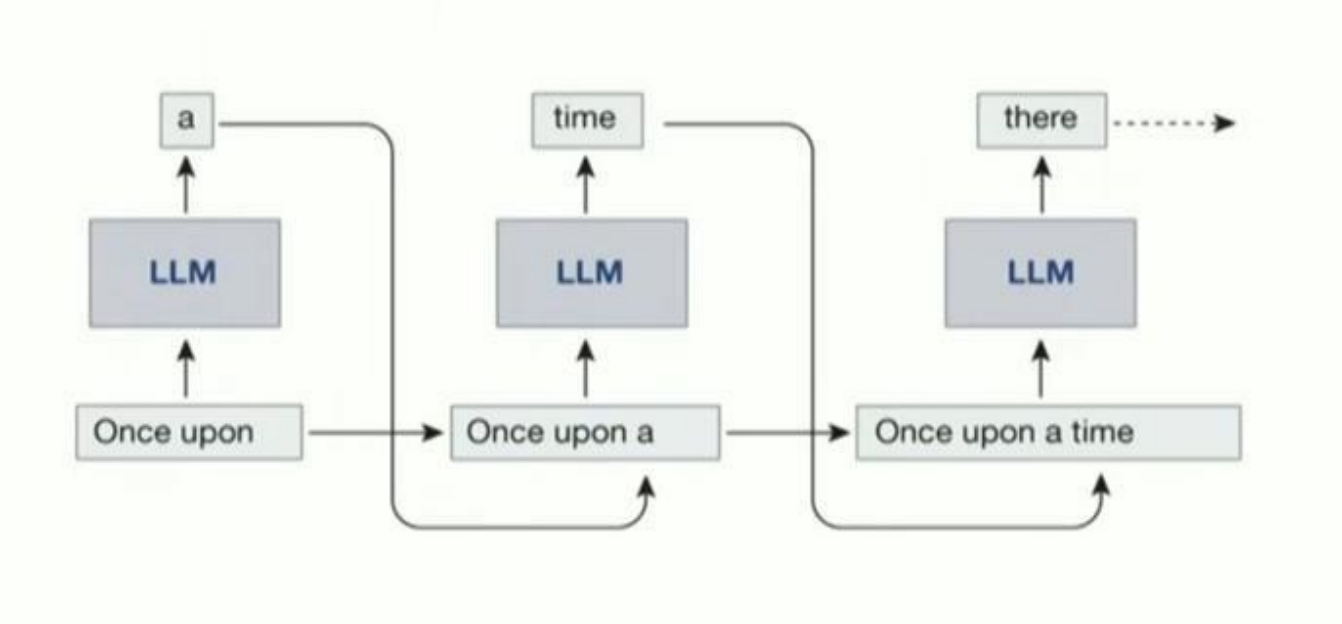
https://platform.openai.com/tokenizer
https://lunary.ai/openai-tokenizer

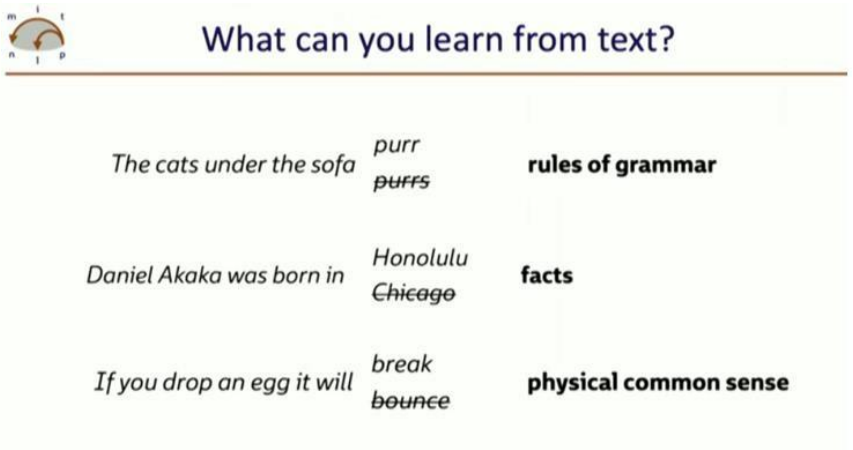
Tokens



https://platform.openai.com/playground

Tokens and structure

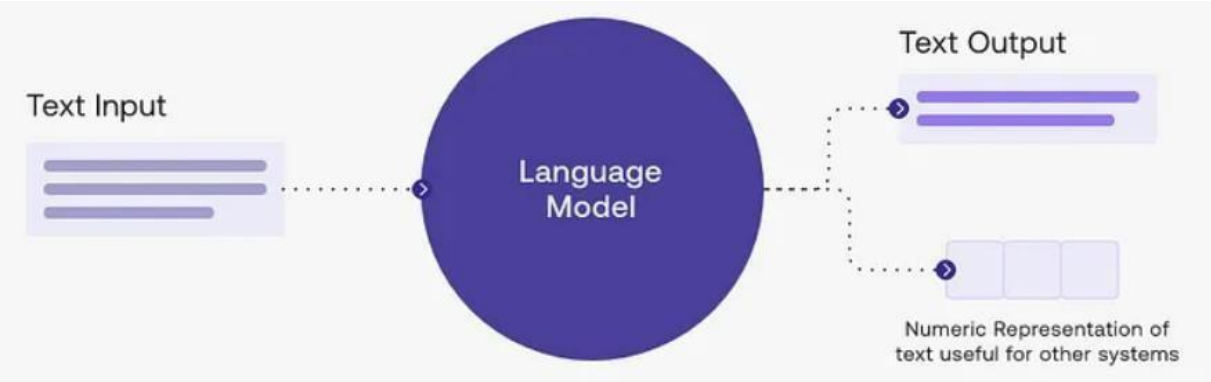




Source:
Shanahan M, McDonell K, Reynolds L. Role play with large language models. Nature. 2023 Nov;623(7987):493-498. doi: 10.1038/s41586-023-06647-8 Epub 2023 Nov 8. PMID: 37938776.

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



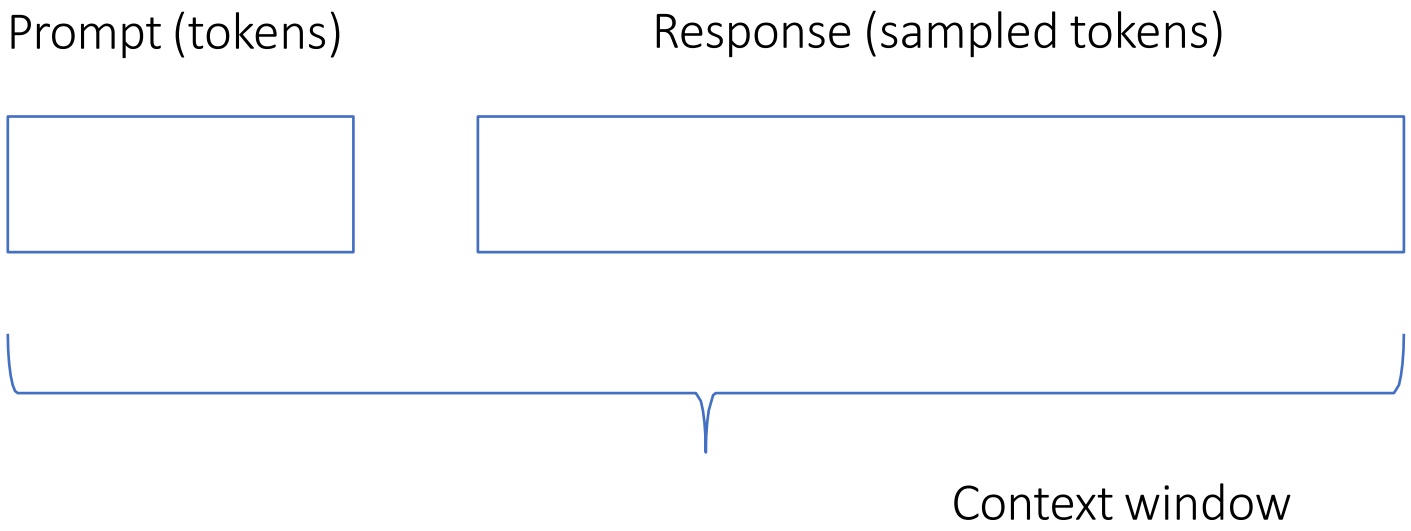
Source: medium.com
https://medium.com/@hmohamedhussain2004/what-is-an-llm-a0086882e585

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)

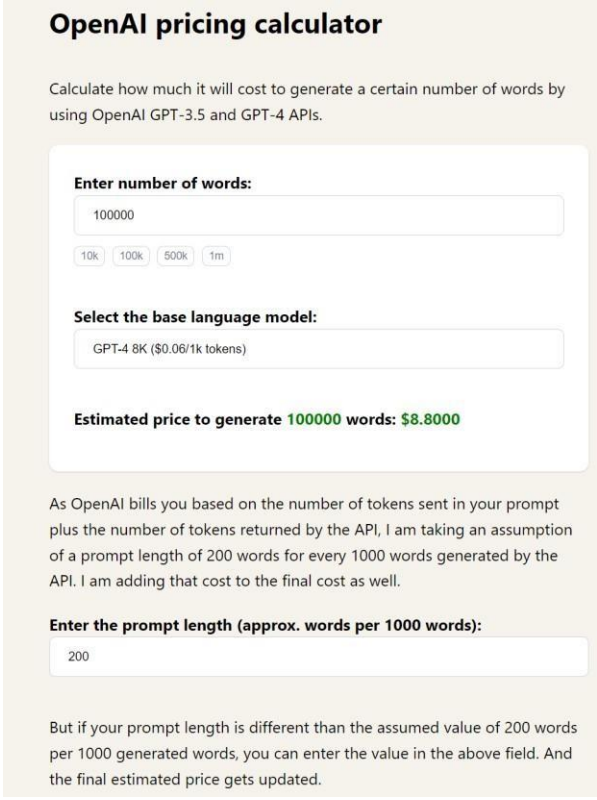
For our models with **8k** context lengths (e.g. `gpt-4` and `gpt-4-0314`), the price is:

- $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:

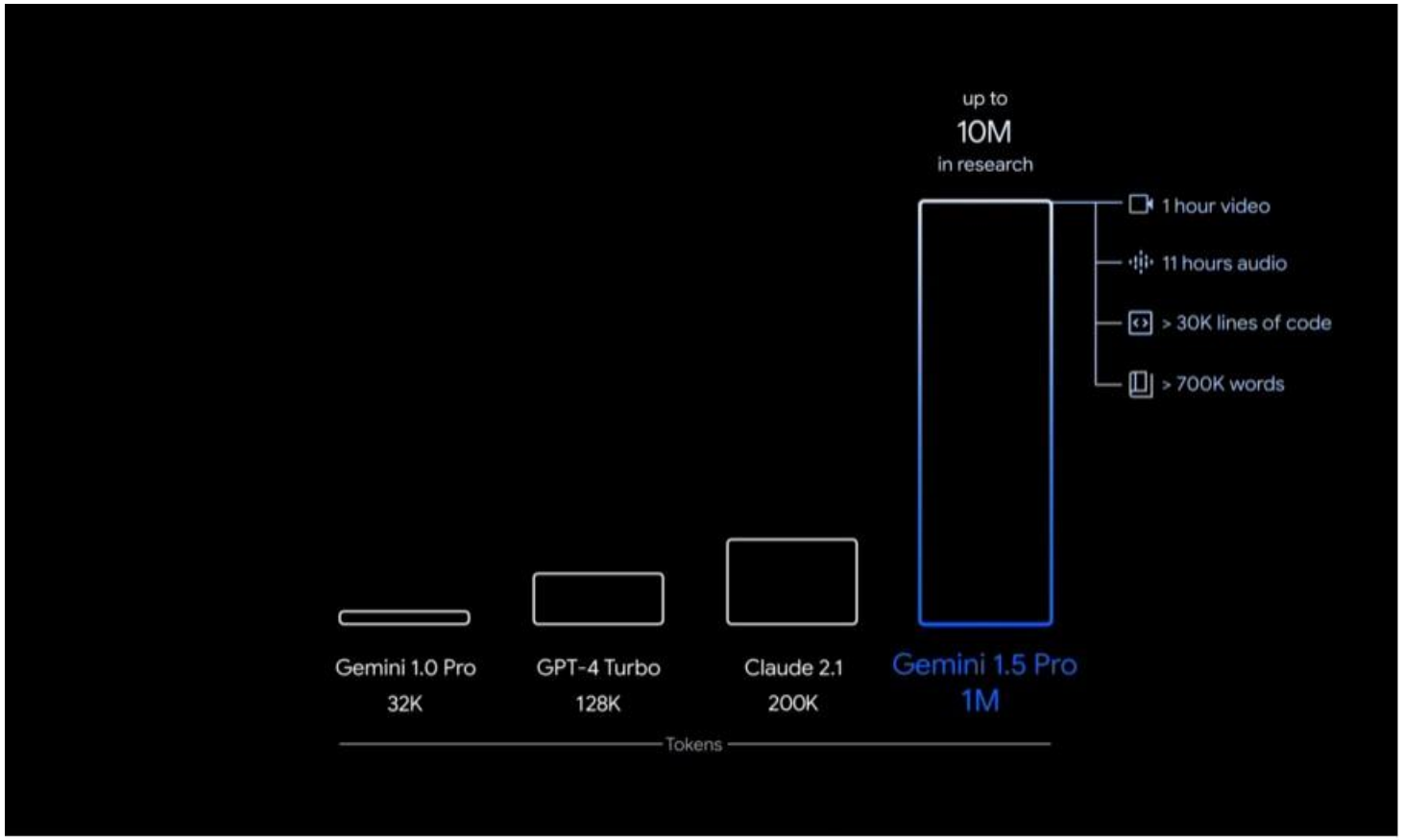
- $60.00 / 1 million prompt tokens (or $0.06 / 1K prompt tokens)
- $120.00 / 1 million sampled tokens (or $0.12 / 1K sampled tokens)

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

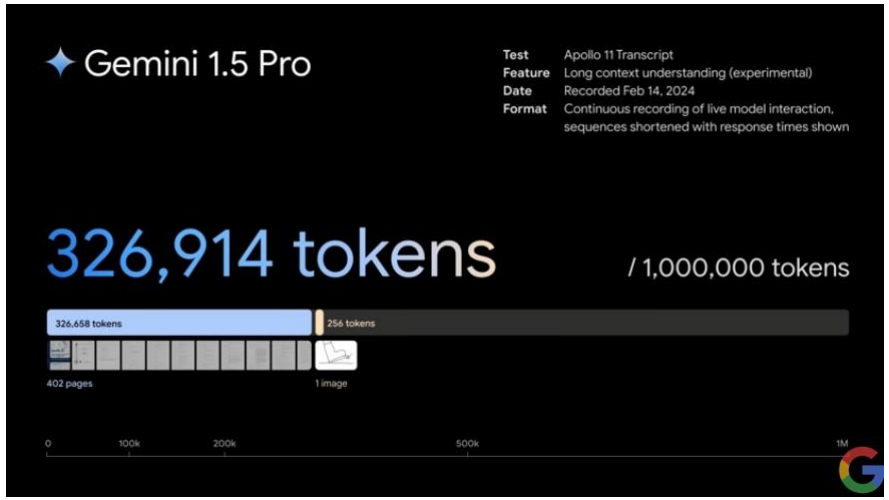


https://invertedstone.com/tools/openai-pricing/

Gemini 1.5 - 1 million multimodal tokens



Context lengths of leading foundation models



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