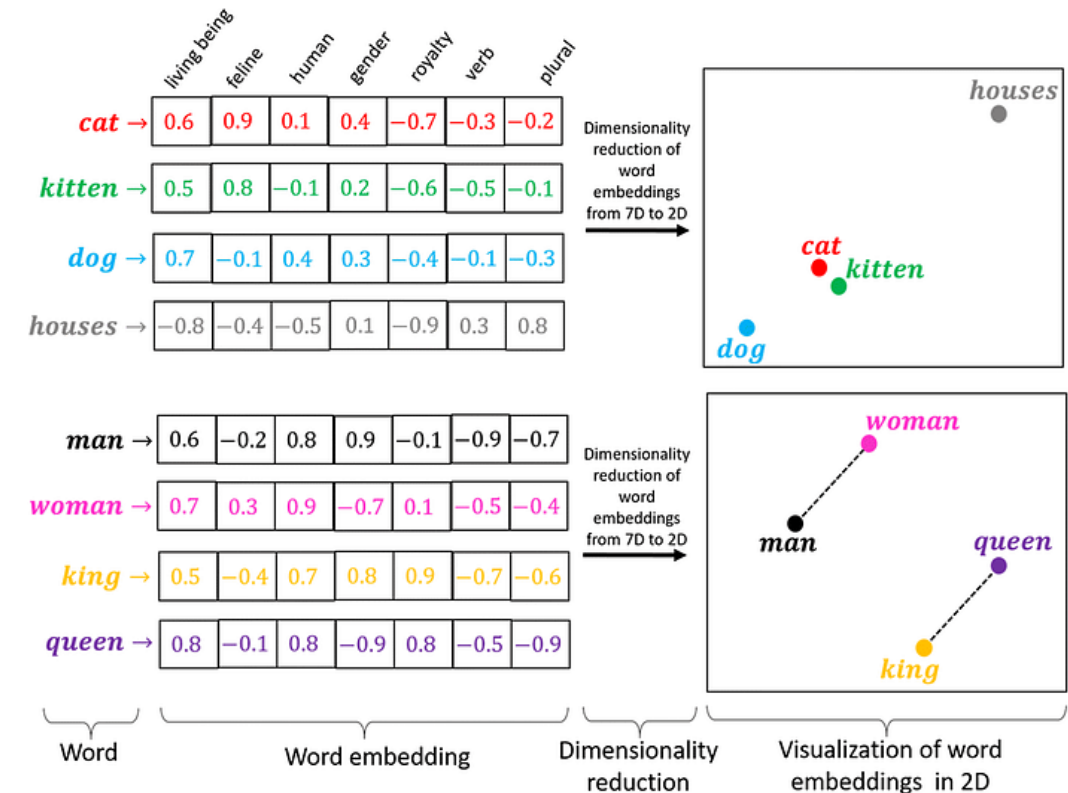


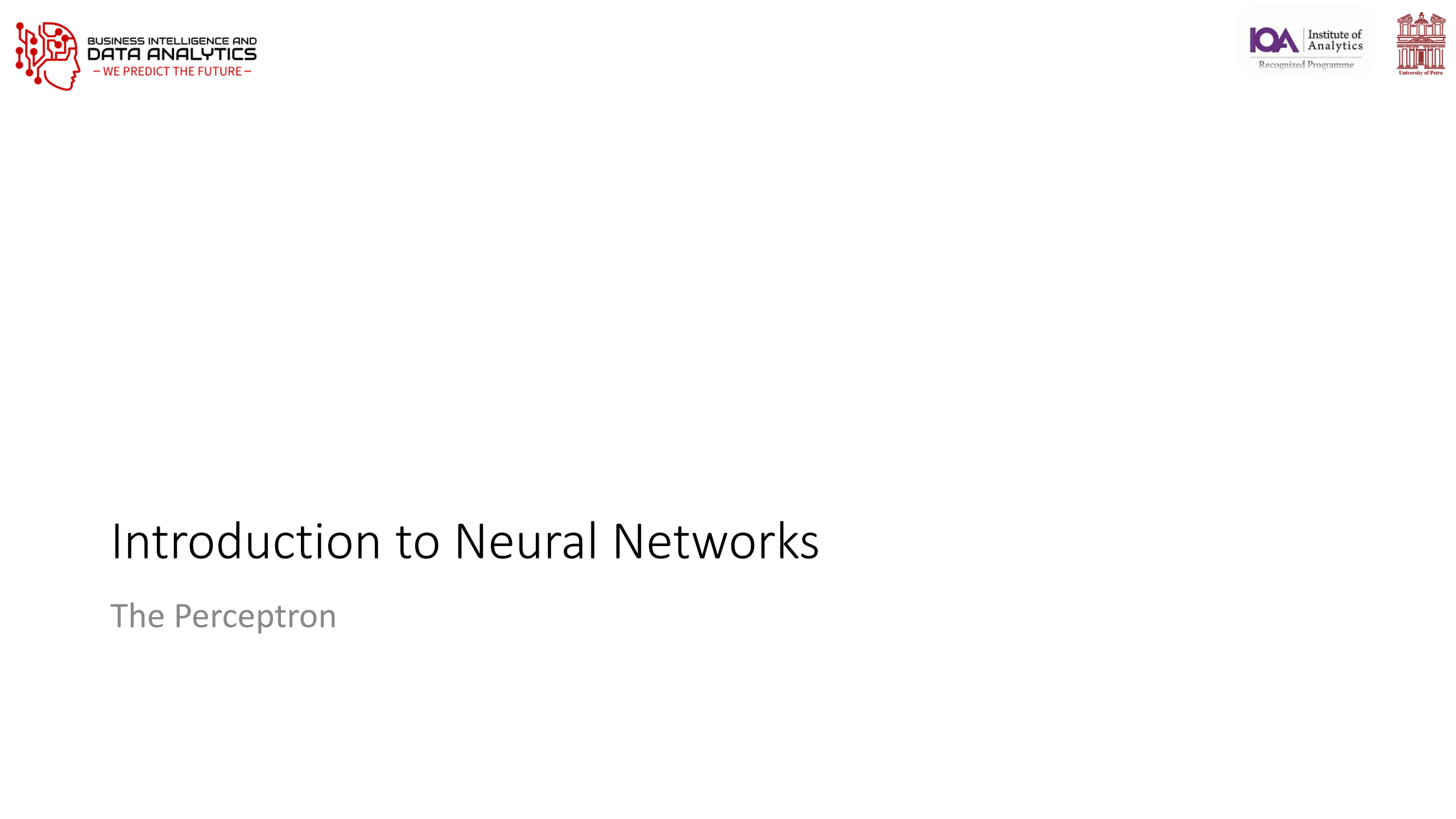
307307

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.





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

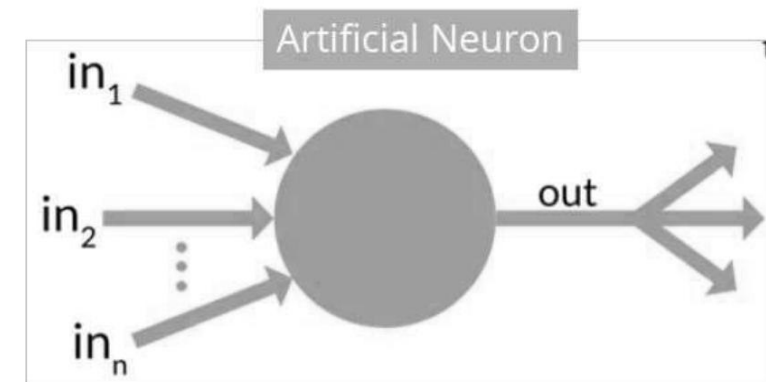
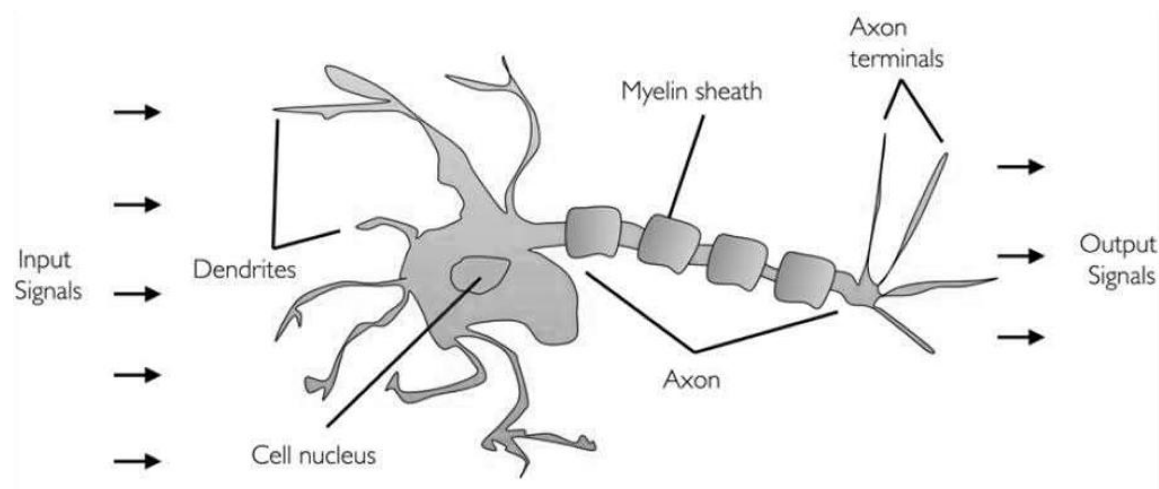
- In 1943, researchers Warren McCulloch and published their first concept of simplified brain cell.
- This was called McCulloch-Pitts (MCP) neuron.
- They described such a nerve cell as a simple logic gate with binary outputs.
- Multiple signals arrive at the dendrites and are then integrated into the 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)

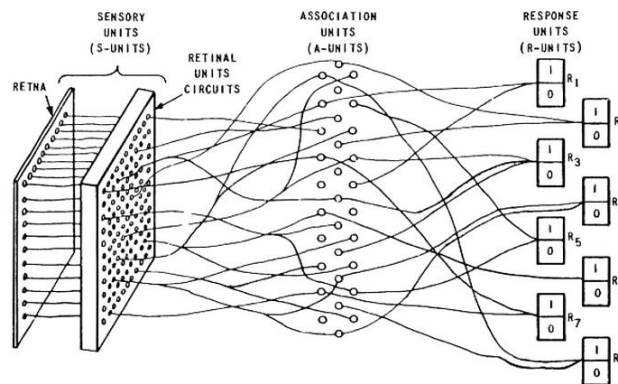


Walter Harry Pitts, Jr.
(1923 – 1969)

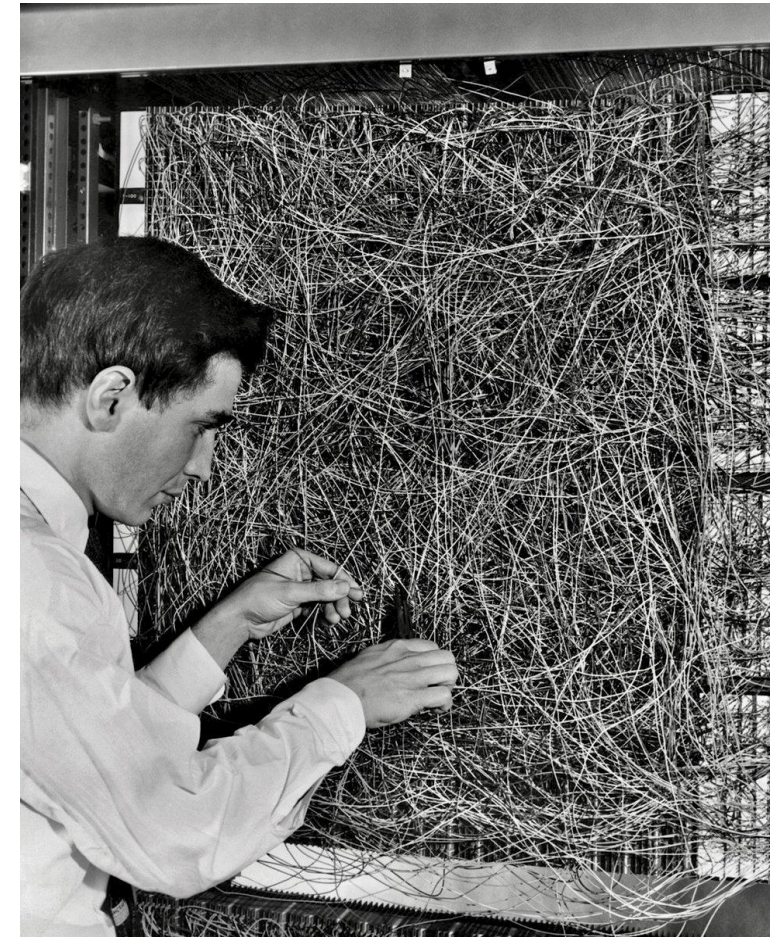


The Perceptron: Building Block of Neural Networks

- In 1953, inspired by McCulloch 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

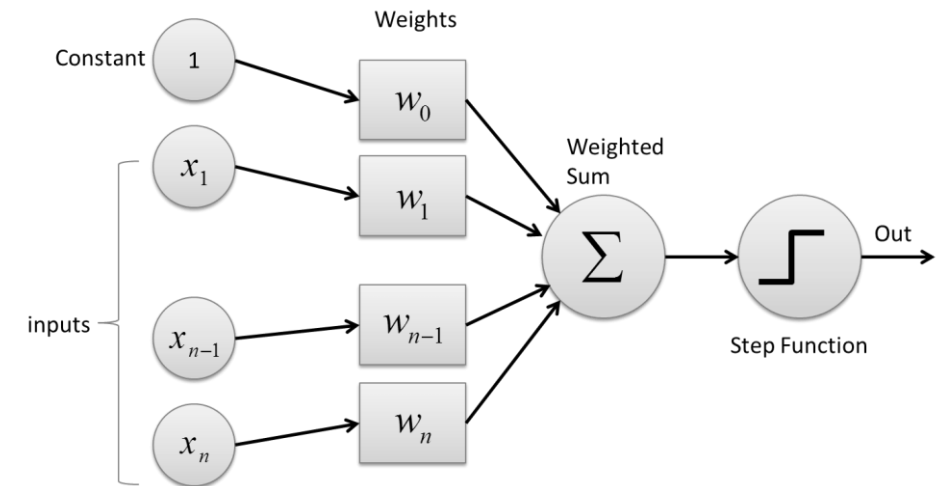


F. Rosenblatt



The Perceptron

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

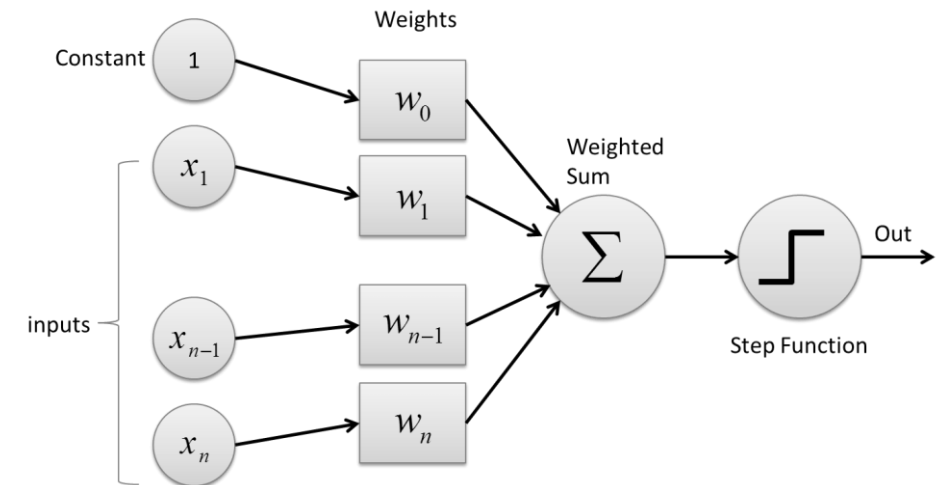


How a Perceptron Works

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

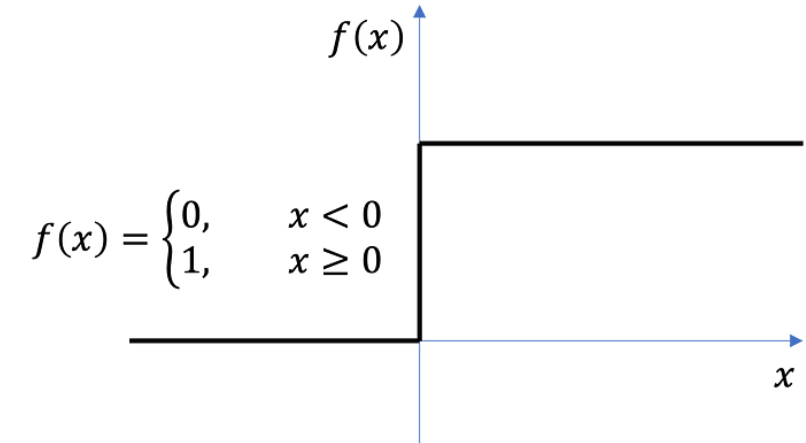
Mathematically:

- $z = w_1x_1 + w_2x_2 + \dots + w_nx_n + b$
- $\text{output} = \text{activation}(z)$



Perceptron Activation Function

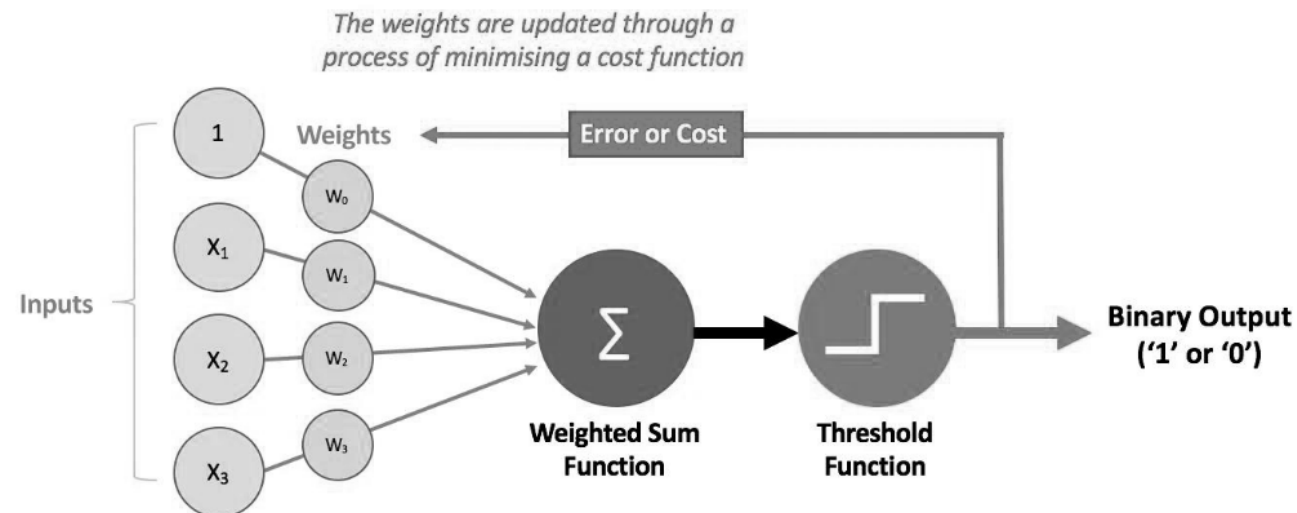
- **Step Function:**
 - Output: 1 if $z \geq 0$, 0 if $z < 0$
 - Used in original perceptrons
 - Not differentiable at 0



How Perceptron Learn (The Cost Function)

For each training example:

1. Calculate predicted output y_{pred}
2. Calculate error: $\text{error} = y_{\text{true}} - y_{\text{pred}}$
3. Update weights: $w_{\text{new}} = w_{\text{old}} + \text{learning_rate} * \text{error} * x$
4. Update bias: $b_{\text{new}} = b_{\text{old}} + \text{learning_rate} * \text{error}$



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 = $y - \text{output} = 0 - 1 = -1$
- 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 = $y - \text{output} = 0 - 1 = -1$
- 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 = $y - \text{output} = 0 - 1 = -1$
- 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 = $y - \text{output} = 1 - 0 = 1$
- 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 = $y - \text{output} = 0 - 1 = -1$
- 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 = $y - \text{output} = 0 - 0 = 0$
- 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

```
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: $0.2 \times (\text{input1}) + 0.2 \times (\text{input2}) - 0.2$

For the four input combinations:

- [0,0]: $0.2 \times 0 + 0.2 \times 0 - 0.2 = -0.2 < 0 \rightarrow \text{output } 0$
- [0,1]: $0.2 \times 0 + 0.2 \times 1 - 0.2 = 0 \rightarrow \text{output } 0$
- [1,0]: $0.2 \times 1 + 0.2 \times 0 - 0.2 = 0 \rightarrow \text{output } 0$
- [1,1]: $0.2 \times 1 + 0.2 \times 1 - 0.2 = 0.2 > 0 \rightarrow \text{output } 1$

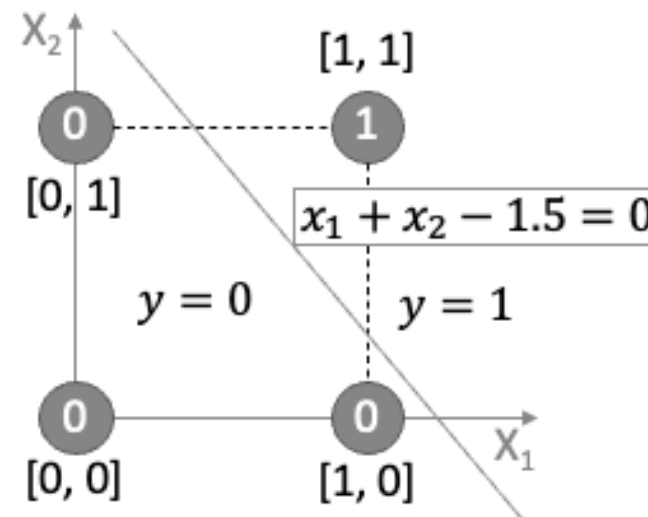
This perceptron implements the AND gate logic.

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

Decision Boundary

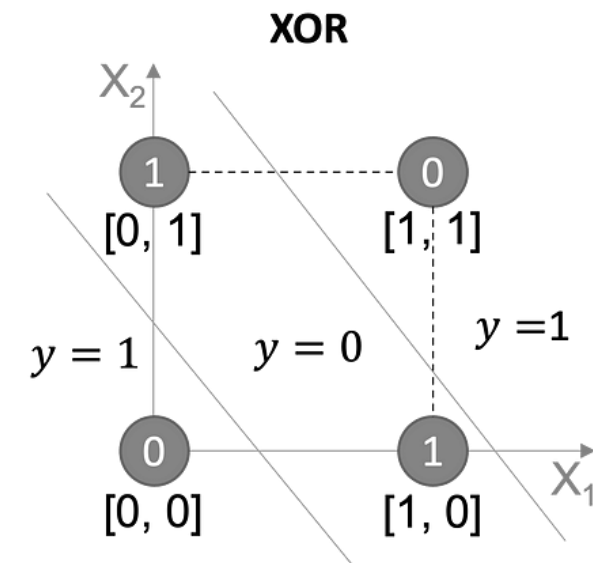
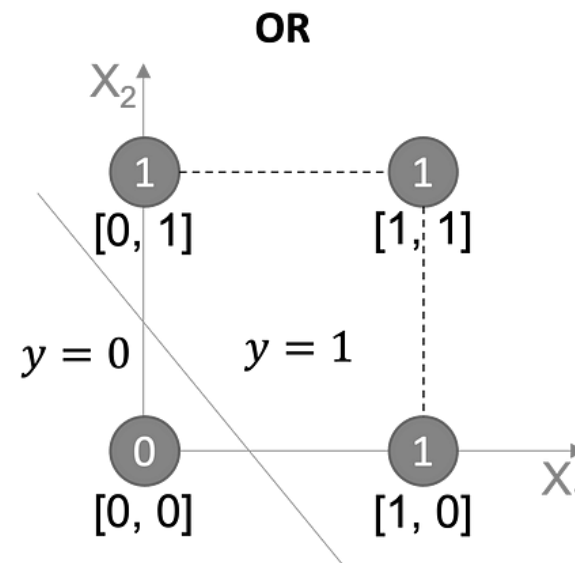
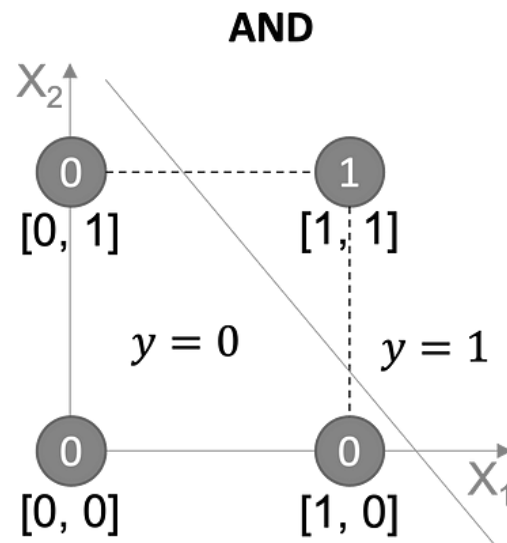
- The perceptron learns a decision boundary: $w_1x_1 + w_2x_2 + b = 0$
- Points above the line are classified as 1
- Points below the line are classified as 0
- For AND gate, only the point (1,1) should be above the line

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 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."

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

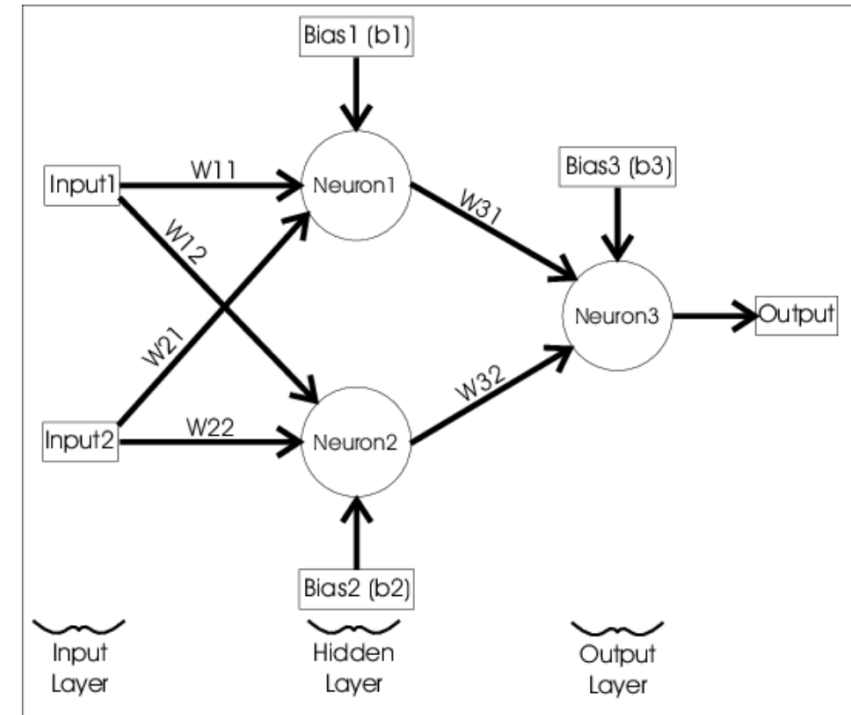
- Receives the raw input features
- One neuron per input feature
- No computation occurs here; inputs are simply passed forward

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

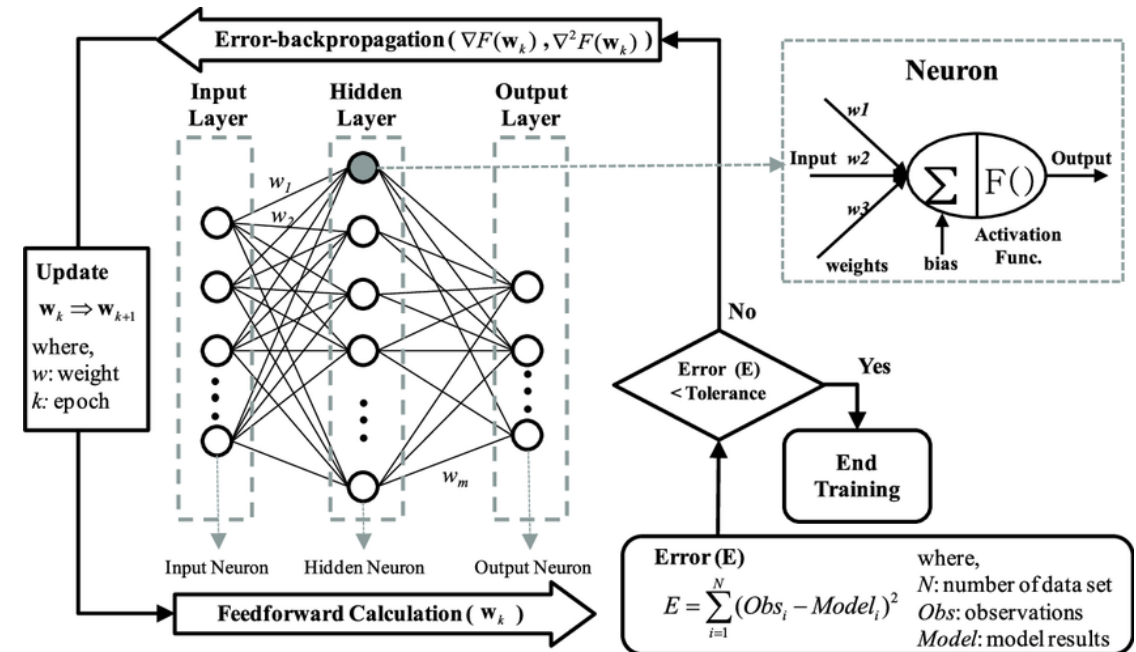


The Neural Network Model to solve the XOR Logic (from: <https://stopsmokingaids.me/>)

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).

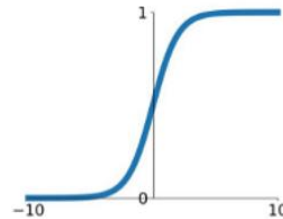


Activation Functions other than Step Function

- Neural Networks use activation functions other than the simple step function in the Perceptron.
- Activation Function helps the neural network use important information while suppressing irrelevant data points (i.e., allows local “gating” of information).

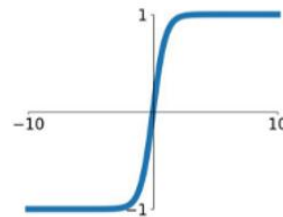
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



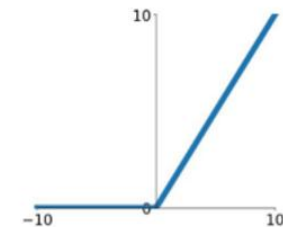
tanh

$$\tanh(x)$$



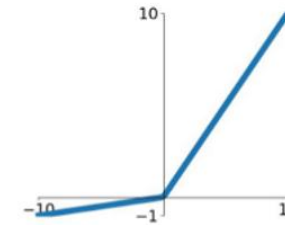
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

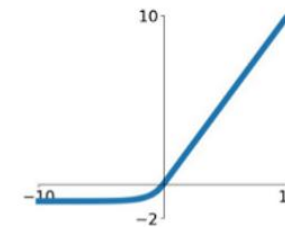


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

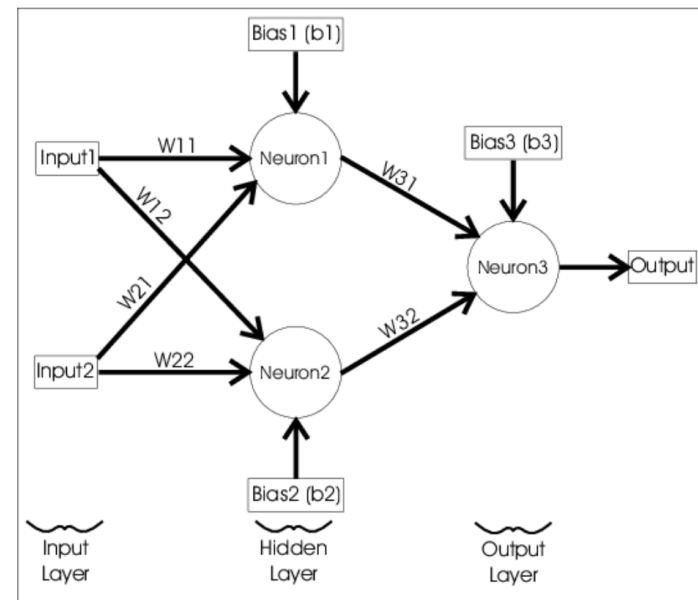


Multi-Layer Perceptron

```
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):      Weights (hidden to output):
[[ 2.7144501  3.27401218]      [[-4.37775211]
 [-2.73418453 -3.17014048]]      [ 4.46553876]]
```

```
Bias hidden:                    Bias output:
[ 1.21994174 -1.63451199]      [3.61855675]
```



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

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.
- And much later, it inspired the **distributional hypothesis** in computational linguistics, especially by scholars like Zellig Harris and later computational models (Word2Vec, etc.)



...government debt problems turning into **banking** crises as happened in 2009...
...saying that Europe needs unified **banking** regulation to replace the hodgepodge...
...India has just given its **banking** system a shot in the arm...

These **context words** will represent **banking**

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

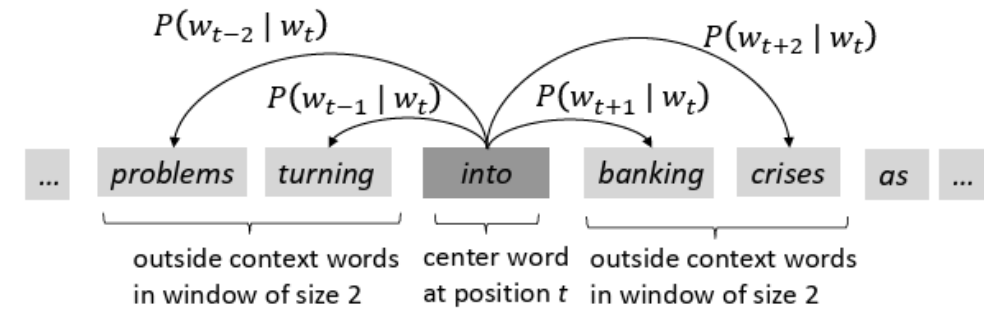
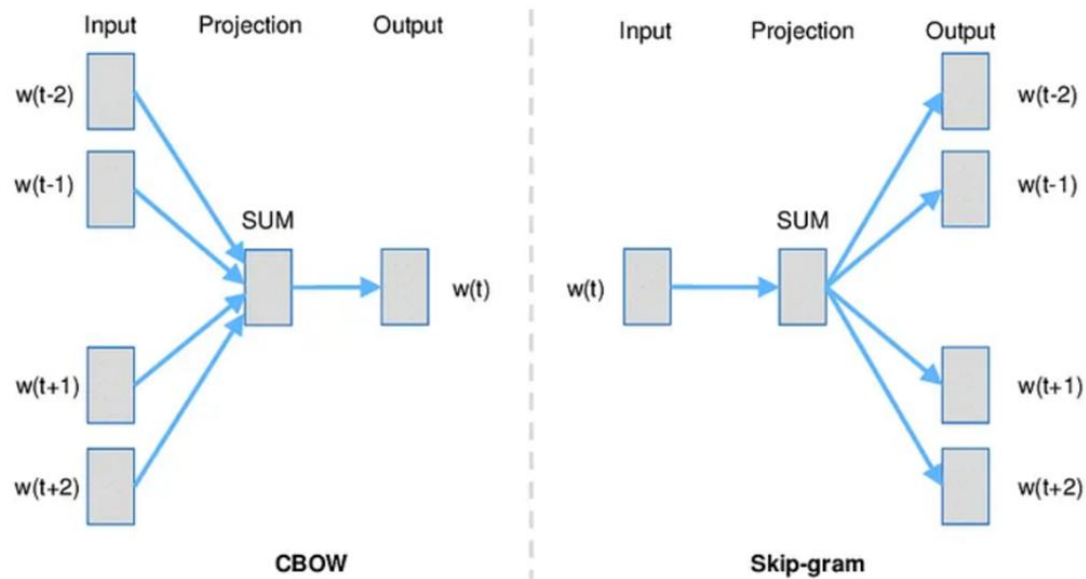
- **Continuous Bag of Words (CBOW):**
 - Predicts target word from context words
 - Faster training, better for frequent words
- **Skip-gram:**
 - Predicts context words from target word
 - Better for rare words, captures more semantic information

Characteristics

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



CBow and Skip-Gram Models

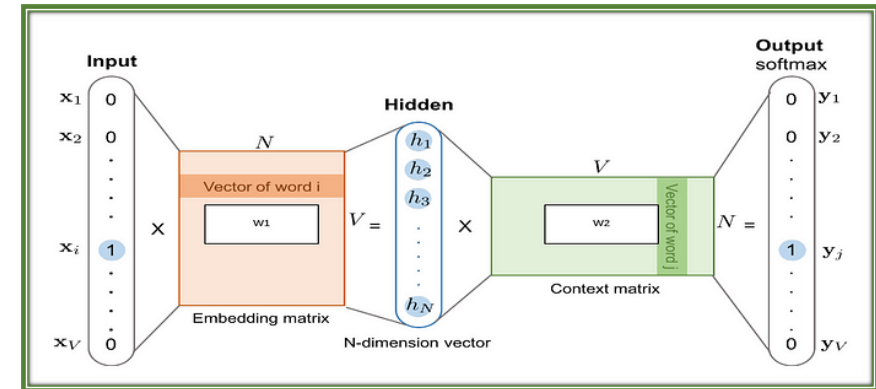


For each position $t=1, \dots, T$, predict context words within a window of fixed size m , given center word w_t .

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.



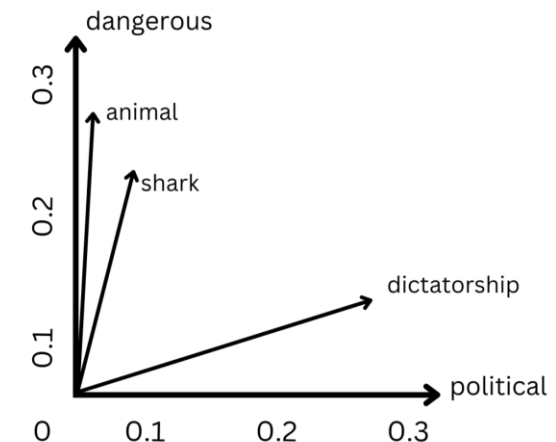
<https://python.plainenglish.io/understanding-word-embeddings-tf-idf-word2vec-glove-fasttext-996a59c1a8d3>

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" \rightarrow [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



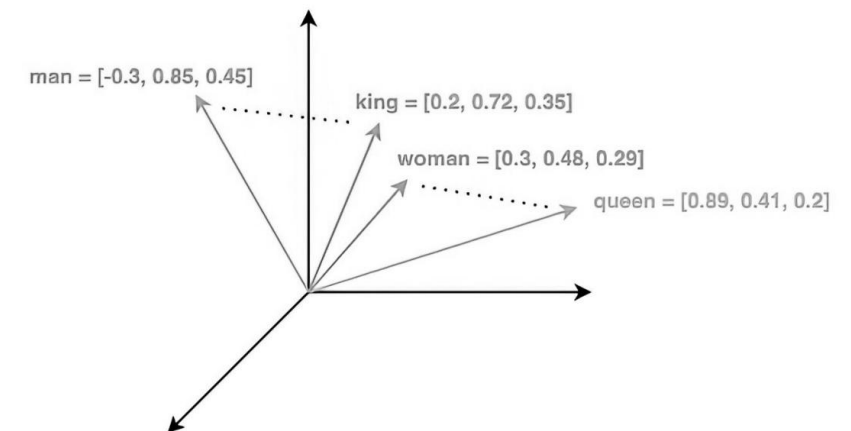
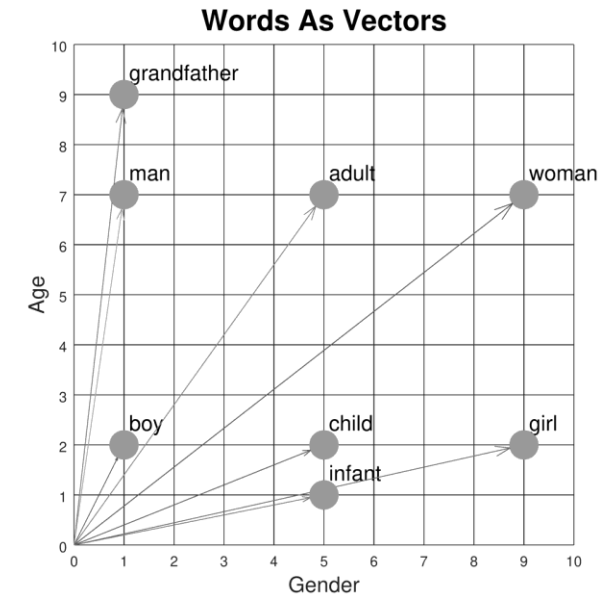
Vector Spaces and Word Embeddings

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" \approx "queen"

Properties of Vector Space

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



Glove (Pennington, Socher, Manning 2014)

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

- Key Innovation**

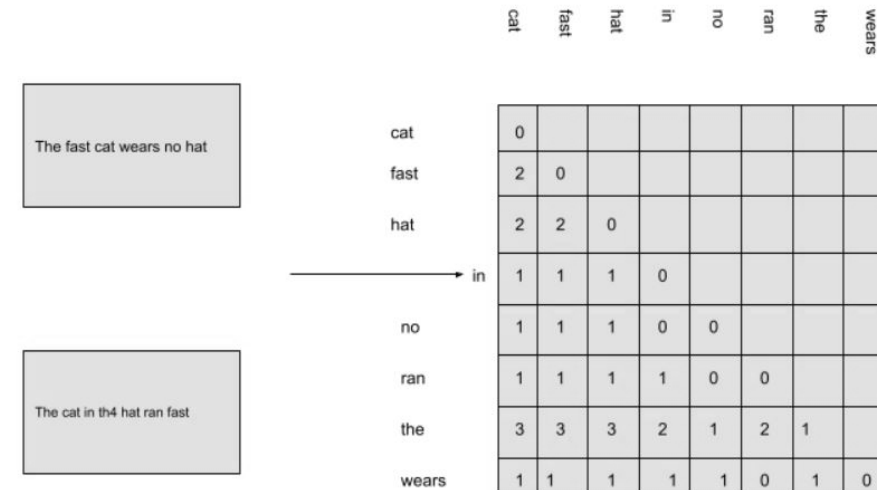
Bridges the gap between count-based methods and prediction-based methods by using global co-occurrence statistics to learn word vectors

- Approach**

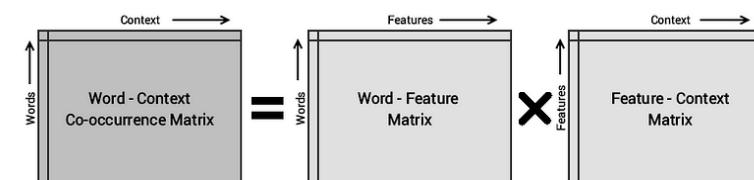
- Builds a word-word co-occurrence matrix over a large corpus
- Learns embeddings by factorizing the matrix using a weighted least squares objective

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 \approx queen
- Trained on massive corpora (Wikipedia, Common Crawl)
- Limitations: Ignores context variability—still one vector per word regardless of usage**



	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

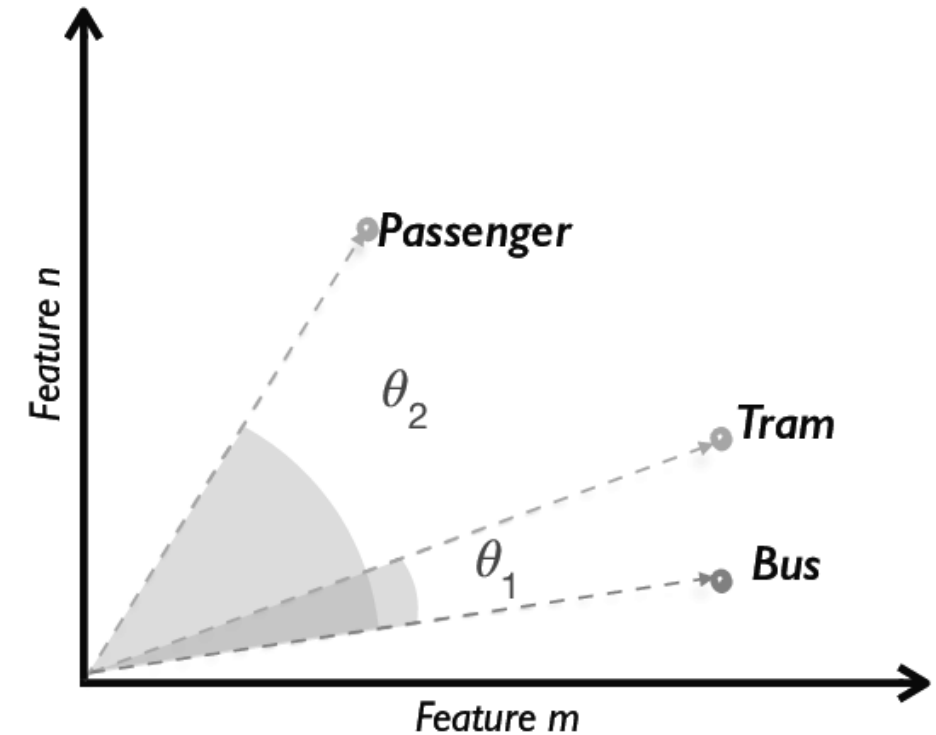
Most common metric used to compare word vectors:

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

- 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.



Experimenting with Fake Embeddings

```
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"))
```

Learn Embeddings From Scratch

```
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),

Use Pre-Trained Embeddings

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.

```
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
```


Word Embeddings Example using Spacy Library

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.

- `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))
```

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.

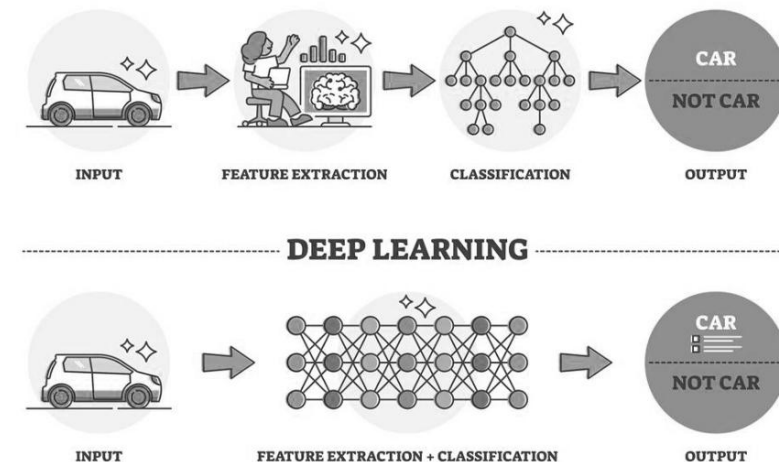
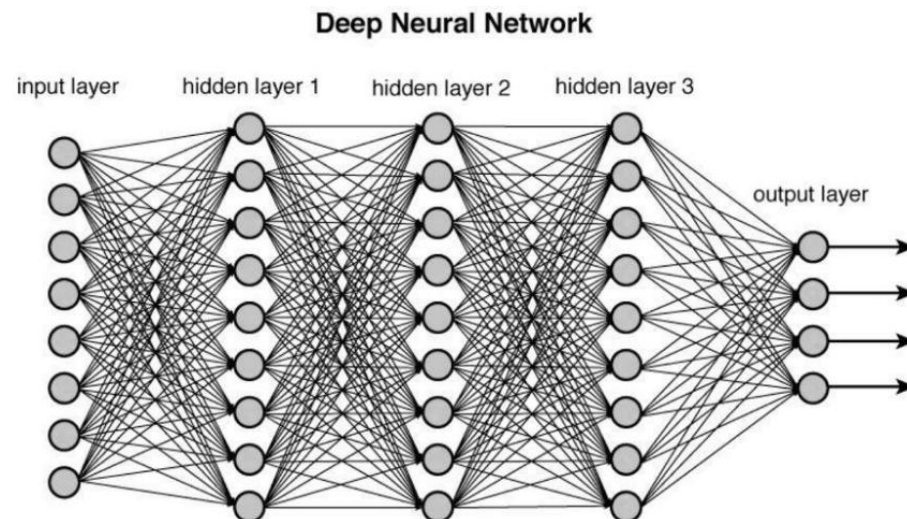
- **6. Information Retrieval**

Enhance search engines by retrieving results based on semantic meaning, not just keyword matches.
Example: Searching for "heart attack" returns documents containing "cardiac arrest."

Introduction to Deep Neural Networks

Introduction to Deep Learning

- Neural Networks have revolutionized artificial intelligence by enabling machines to learn from data in ways that mimic human neural processes.
- Deep neural networks (DNNs) are Neural Networks that are composed of multiple processing layers that can learn representations of data with **multiple levels of abstraction**.
- The power of deep learning comes from its ability to **automatically discover intricate patterns** in raw data through the learning process, without requiring human engineers to manually specify all the knowledge needed by the computer system.



Introduction to Deep Learning

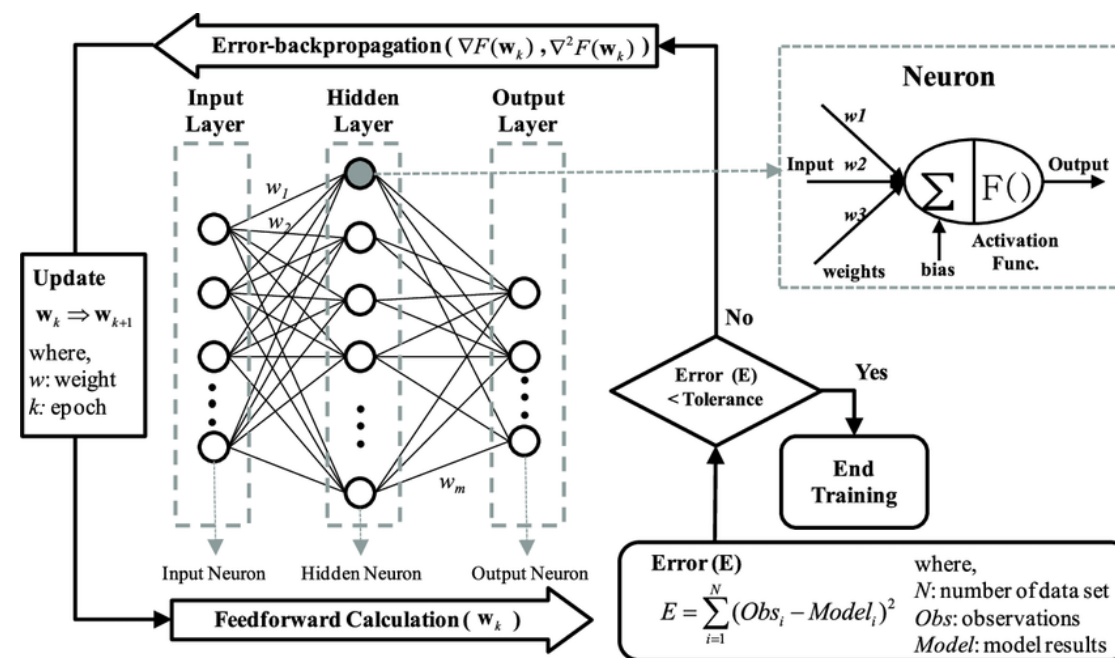
Fundamentals of Neural Networks

At their core, neural networks consist of:

1. **Neurons:** Mathematical functions that take inputs, apply weights, add a bias, and produce an output
2. **Layers:** Collections of neurons that process information in stages
3. **Activation Functions:** Non-linear functions that introduce complexity into the network
4. **Weights and Biases:** Parameters that are adjusted during training

The basic workflow involves:

- Forward propagation: Data flows through the network
- Loss calculation: The network's prediction is compared to the actual value
- Backpropagation: Errors are propagated backward to update weights
- Optimization: Weights are adjusted to minimize errors



Convolutional Neural Networks

CNNs revolutionized **image processing** by introducing **specialized layers** that mimic how the visual cortex processes information.

Key components include:

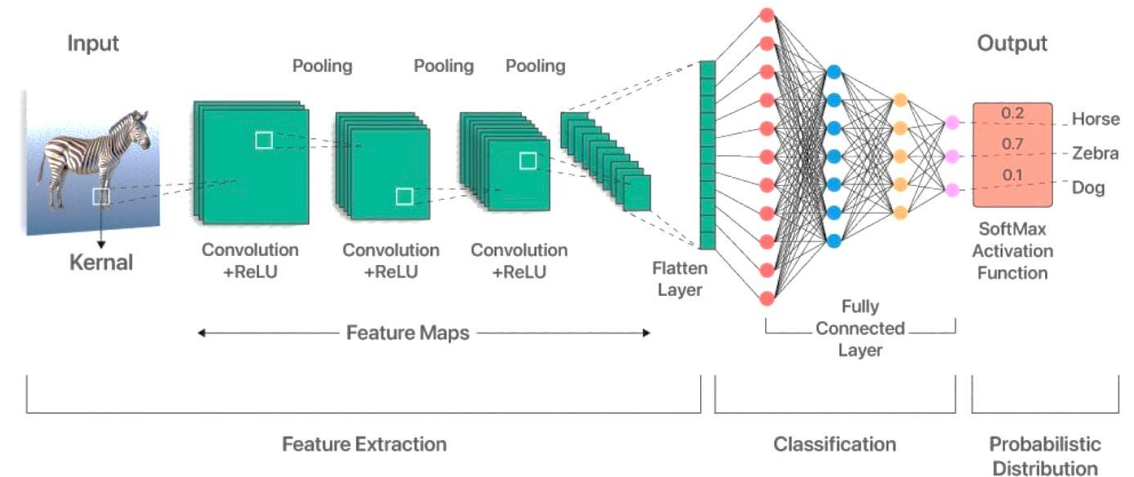
1. **Convolutional Layers:** Apply filters that scan across the input data to detect patterns
2. **Pooling Layers:** Reduce dimensions while preserving important features
3. **Fully Connected Layers:** Connect every neuron to every neuron in adjacent layers

Instead of each neuron connecting to every pixel in an image (which would be computationally expensive), CNNs use:

- **Local connectivity:** Neurons connect only to nearby pixels
- **Parameter sharing:** The same filter is applied across the entire image

Business applications include:

- Product image recognition
- Visual quality control in manufacturing
- Document processing
- Customer behavior analysis in retail



3 ₀	3 ₁	2 ₂	1	0
0 ₂	0 ₂	1 ₀	3	1
3 ₀	1 ₁	2 ₂	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

Recurrent Neural Networks (RNNs)

Unlike traditional neural networks, RNNs process sequences by maintaining a form of memory of previous inputs.

Key characteristics:

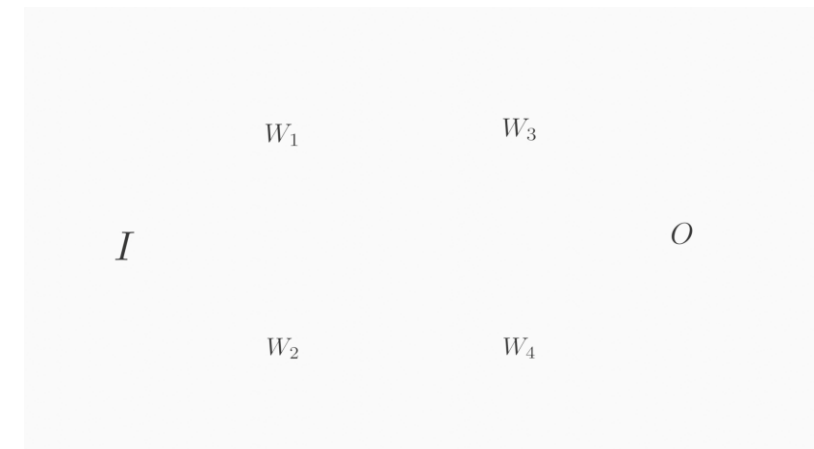
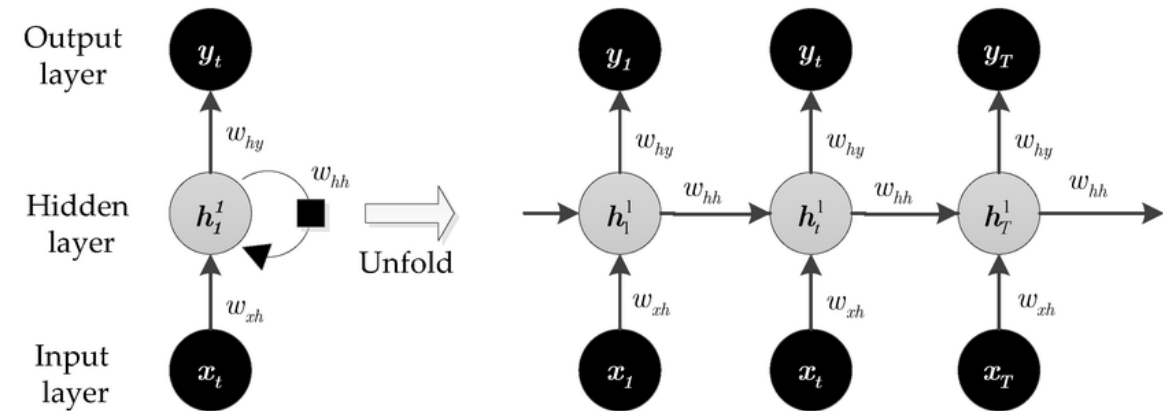
- **Time-dependent processing:** Output depends on both current and previous inputs
- **Shared parameters:** The same weights are applied at each time step
- **Memory:** Internal state acts as a form of short-term memory

However, basic RNNs struggle with long-term dependencies due to:

- **Vanishing gradient problem:** The influence of early inputs fades over time
- **Exploding gradient problem:** Gradients grow uncontrollably during training

Business applications include:

- Language Modeling & Text Generation – Predicting the next word in a sequence (e.g., autocomplete, chatbots).
- Machine Translation – Translating text from one language to another.
- Speech Recognition – Converting spoken language into written text.
- Stock Price Prediction – Predicting future stock or financial data.
- Weather Forecasting – Modeling temporal patterns in weather data.
- Patient Monitoring – Analyzing sequences of medical data (e.g., ECG signals).
- Music Generation – Creating sequences of musical notes.
- Fraud Detection – Detecting unusual sequences in financial transactions.
- Network Intrusion Detection – Monitoring patterns of activity over time.



Transformers

<https://jalammar.github.io/illustrated-transformer/>

Transformers – Architecture and Principles

What is a Transformer?

- A deep learning model based entirely on **self-attention**, with no recurrence or convolutions
- Introduced in the paper “*Attention Is All You Need*” (Vaswani et al., 2017)

Key Innovation: Self-Attention Mechanism

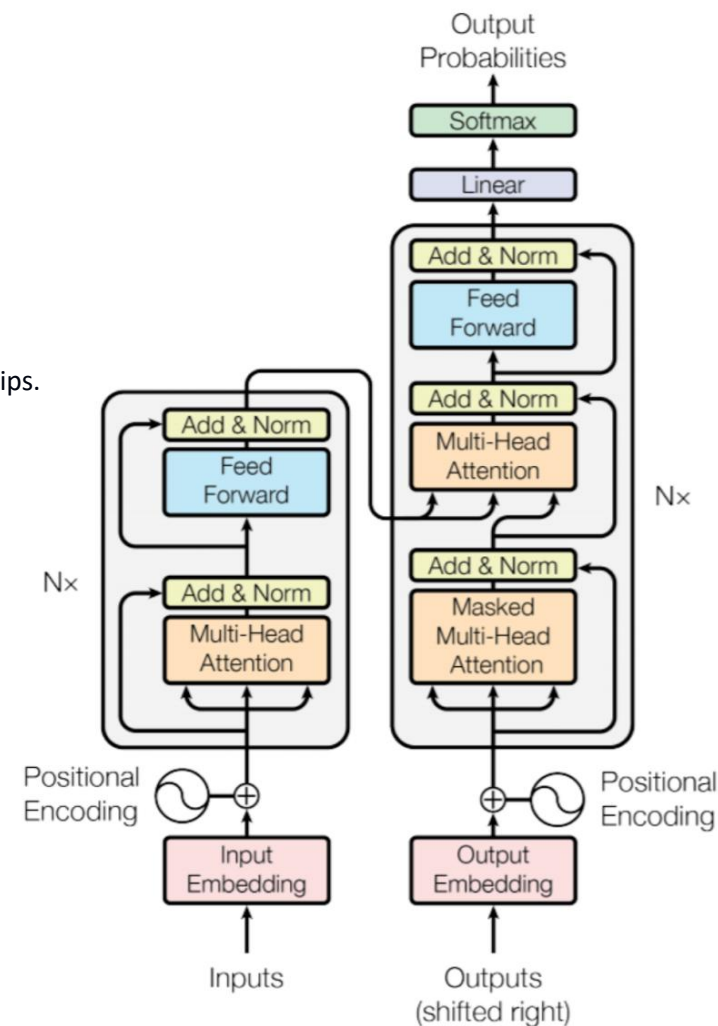
- Self-attention allows each word to gather information from all other words in the sequence.
- The “attention weights” determine how much each word should focus on other words.
- Words that are semantically related tend to have higher attention scores between them.
- This mechanism helps capture long-range dependencies and relationships regardless of word distance.
- Multiple attention heads in parallel (Multi-head Attention) allow the model to focus on different aspects of relationships.

Transformer Components:

1. **Embeddings:** Convert tokens to vector representations
2. **Positional Encoding:** Adds position information
3. **Multi-Head Attention:** Processes relationships from multiple perspectives
4. **Feed-Forward Networks:** Process each position independently
5. **Layer Normalization:** Stabilizes training
6. **Residual Connections:** Helps with gradient flow

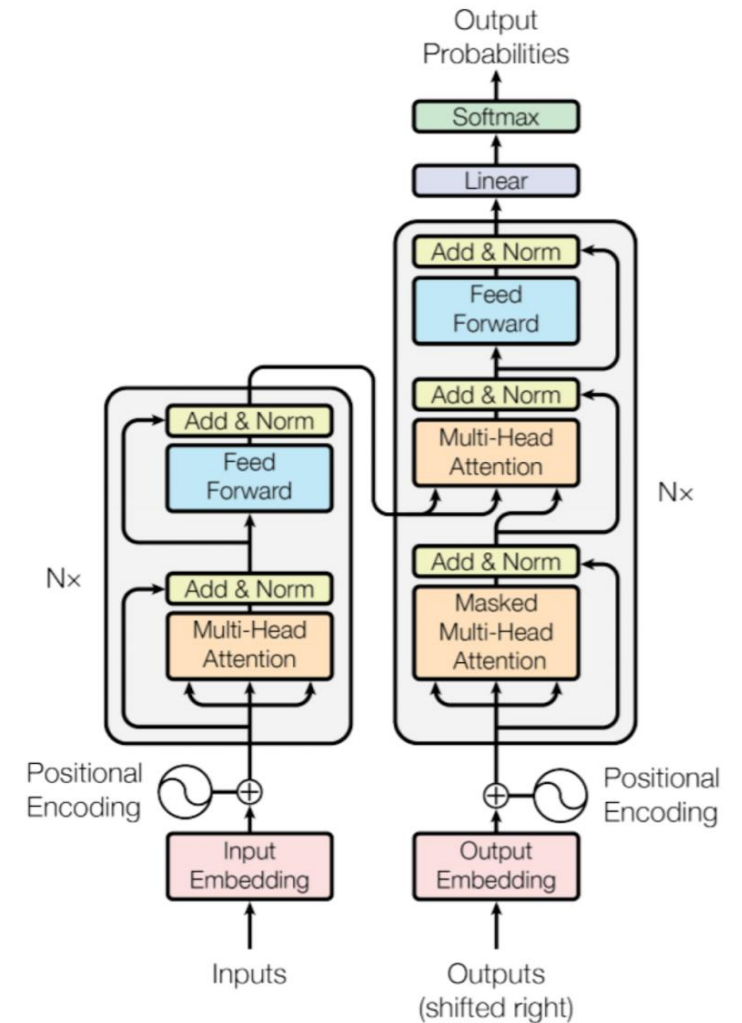
Architecture Variations:

- **Encoder-only** (BERT): Good for understanding (classification, NER)
- **Decoder-only** (GPT): Good for generation
- **Encoder-decoder** (T5): Good for transformation tasks (translation, summarization)



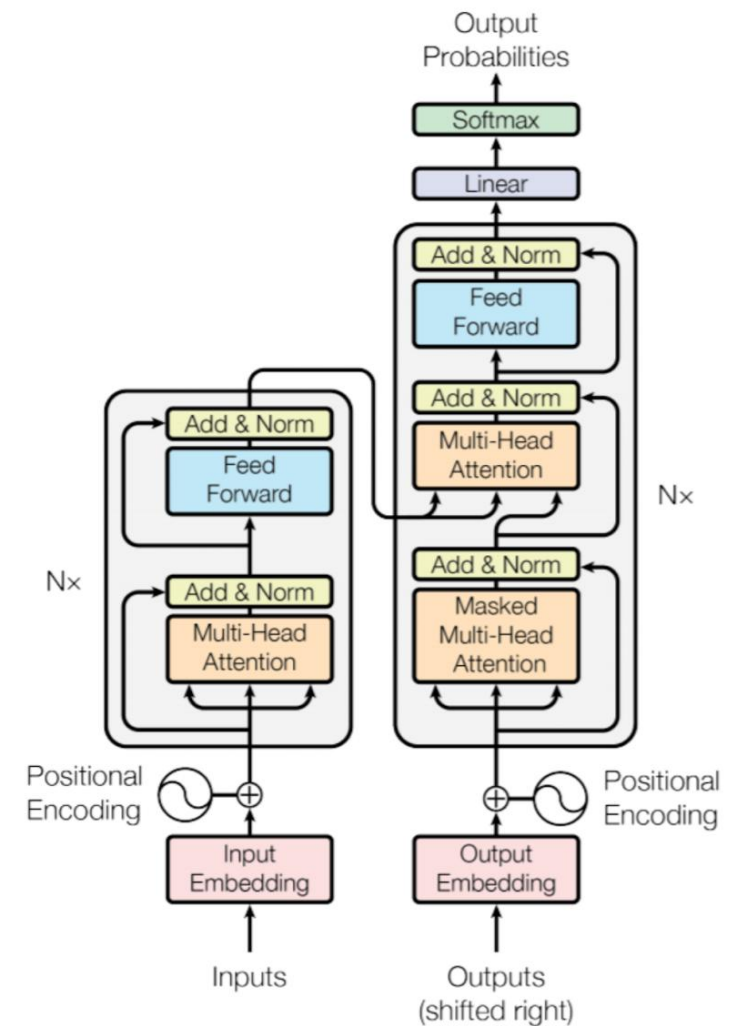
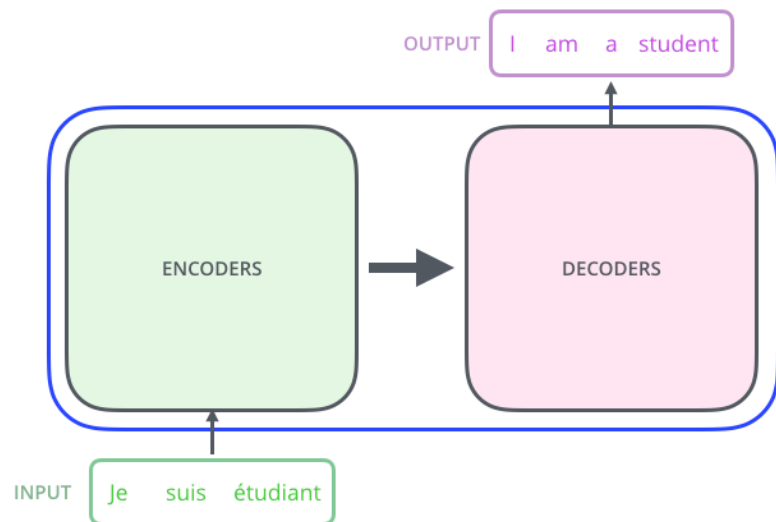
A High-Level Look at the Transformer

In a machine translation application, it would take a sentence in one language and output its translation in another.



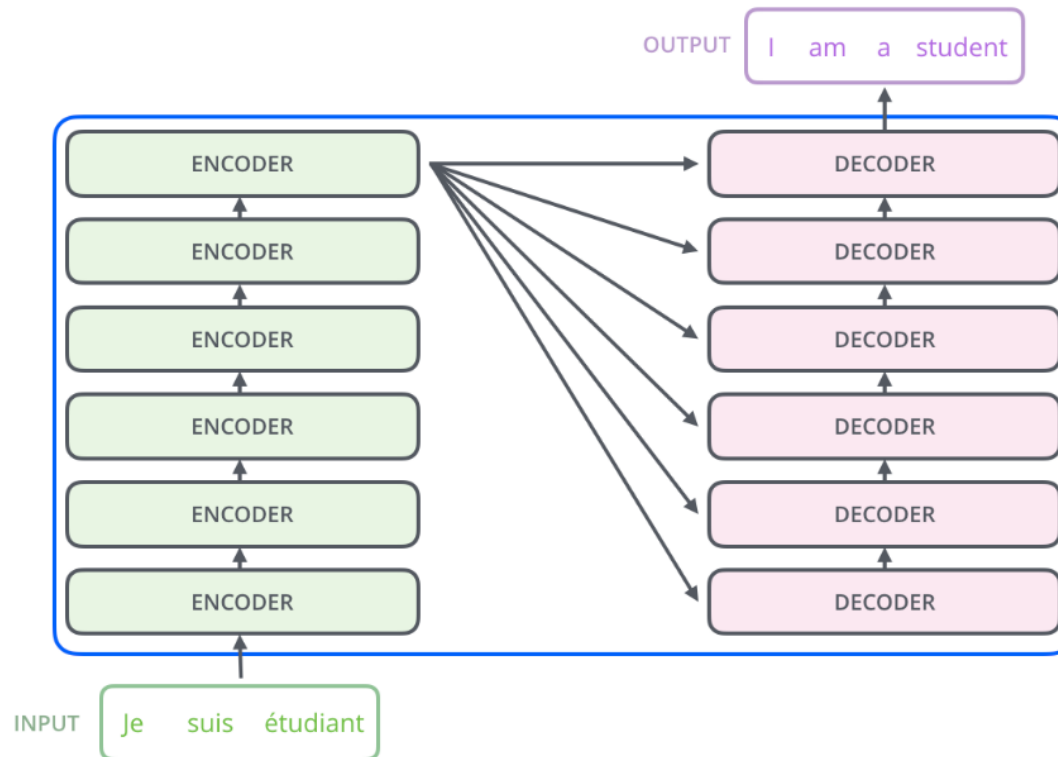
Looking Inside

The transformer is comprised of two main components, the encoders part and the decoders.

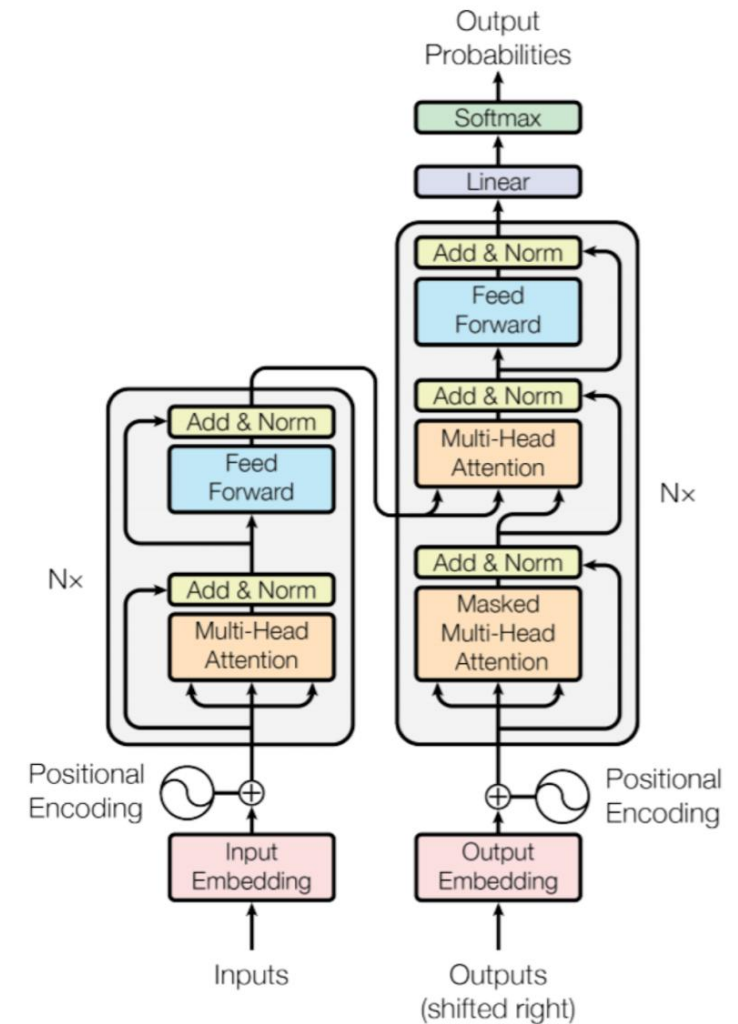


The Transformer

The encoding component is a stack of encoders (the paper stacks six of them on top of each other – there's nothing magical about the number six, one can definitely experiment with other arrangements). The decoding component is a stack of decoders of the same number.

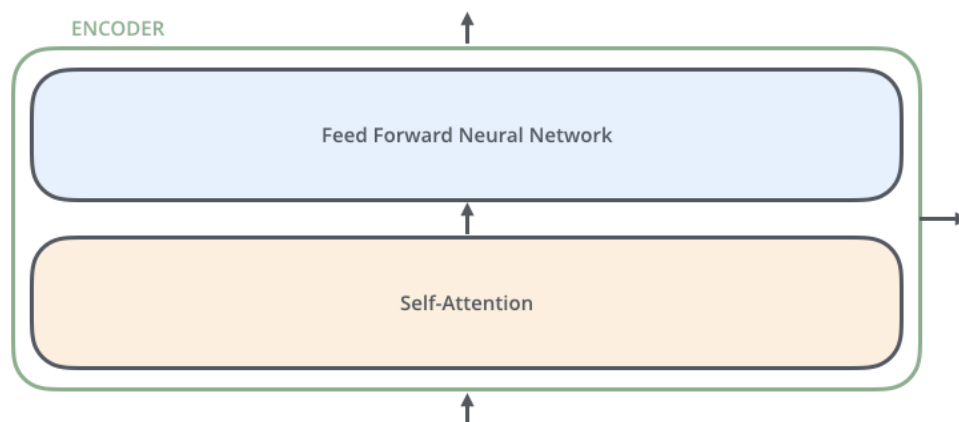


The encoders are all identical in structure (yet they do not share weights). Each one is broken down into two sub-layers:



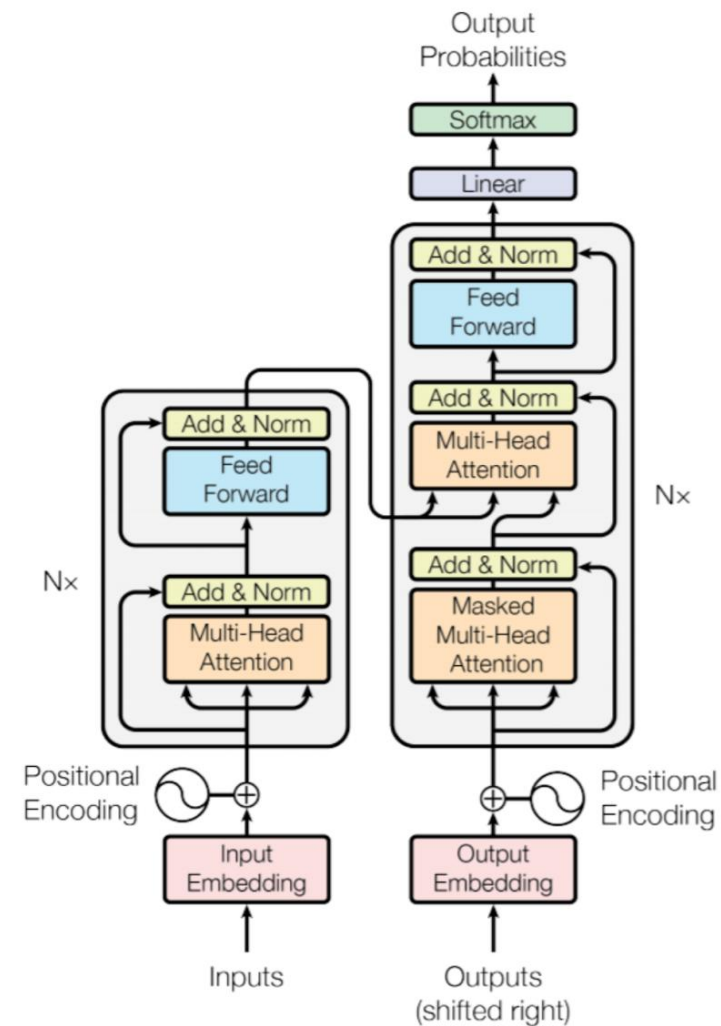
The Transformer

The encoders are all identical in structure (yet they do not share weights). Each one is broken down into two sub-layers:



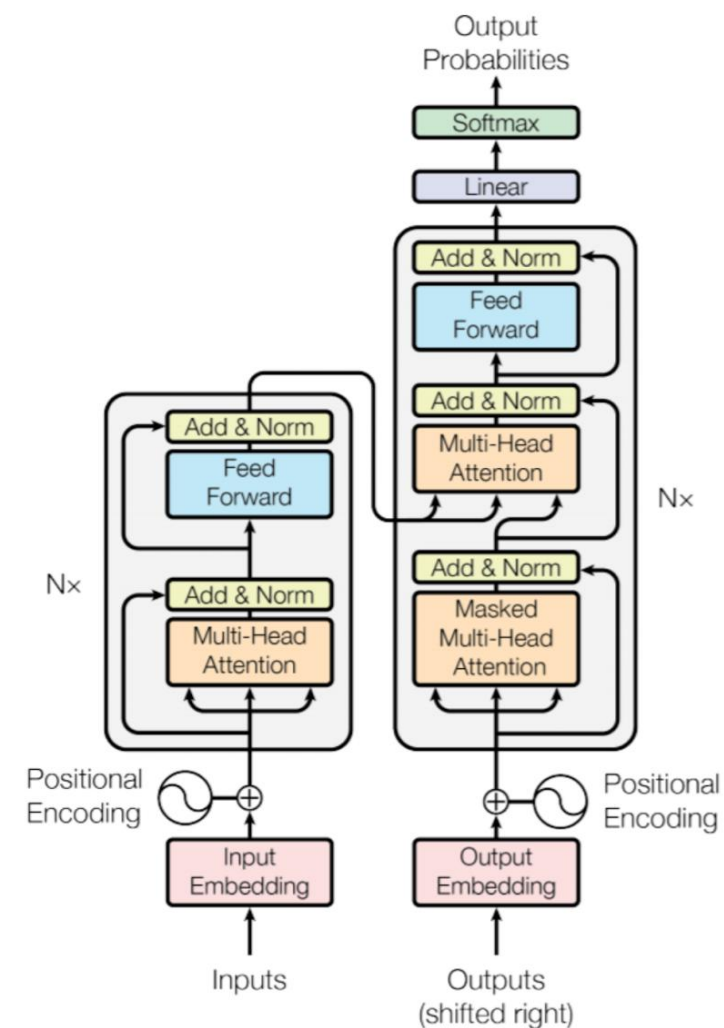
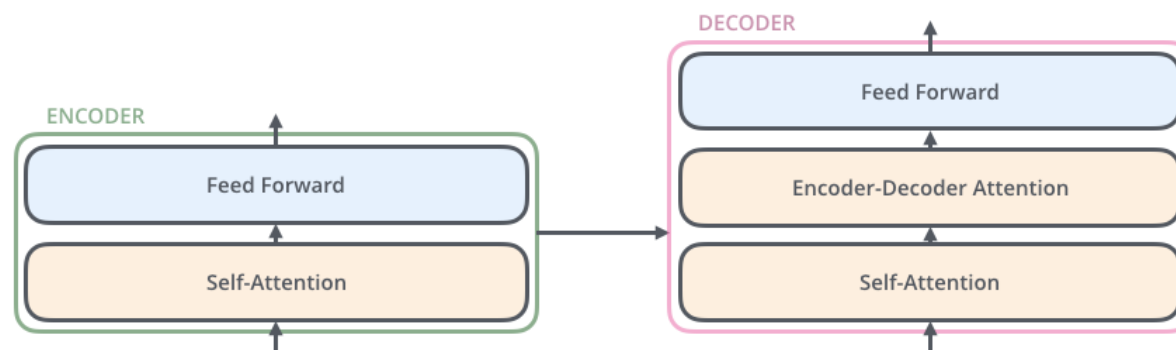
The encoder's inputs first flow through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word. We'll look closer at self-attention later in the post.

The outputs of the self-attention layer are fed to a feed-forward neural network. The exact same feed-forward network is independently applied to each position.



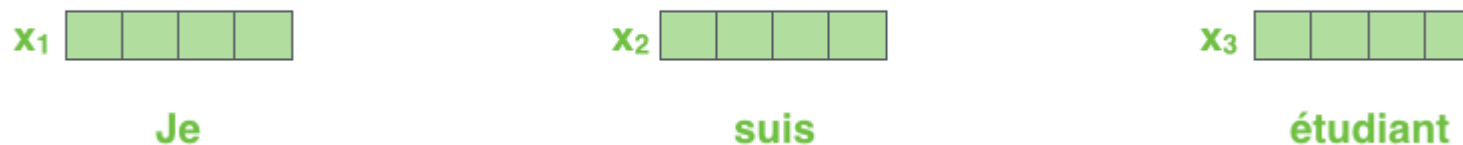
The Transformer

The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence (similar what attention does in [seq2seq models](#)).



Process Flow

- Now that we've seen the major components of the model, let's start to look at the various vectors/tensors and how they flow between these components to turn the input of a trained model into an output.
- As is the case in NLP applications in general, we begin by turning each input word into a vector using an [embedding algorithm](#).

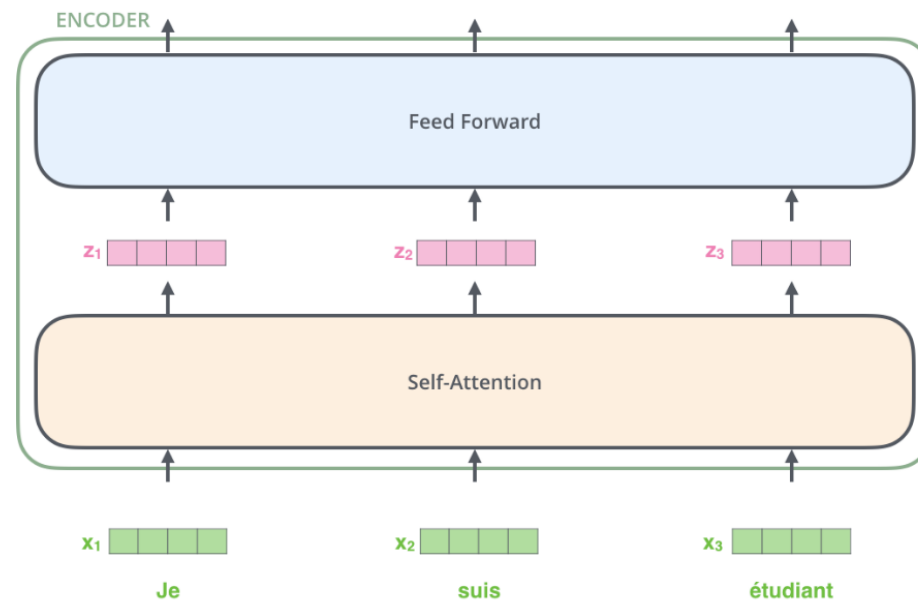


Each word is embedded into a vector of size 512. We'll represent those vectors with these simple boxes.

The embedding only happens in the bottom-most encoder. The abstraction that is common to all the encoders is that they receive a list of vectors each of the size 512 – In the bottom encoder that would be the word embeddings, but in other encoders, it would be the output of the encoder that's directly below. The size of this list is hyperparameter we can set – basically it would be the length of the longest sentence in our training dataset.

Process Flow

After embedding the words in our input sequence, each of them flows through each of the two layers of the encoder.



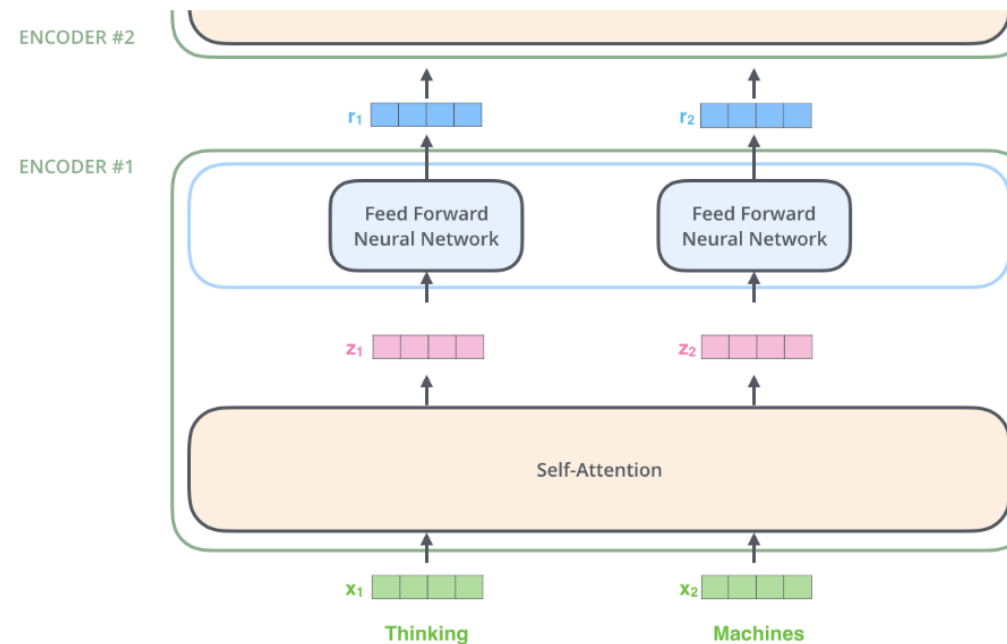
Here we begin to see one key property of the Transformer, which is that the word in each position flows through its own path in the encoder. There are dependencies between these paths in the self-attention layer.

The feed-forward layer does not have those dependencies, however, and thus the various paths can be executed in parallel while flowing through the feed-forward layer.

Next, we'll switch up the example to a shorter sentence and we'll look at what happens in each sub-layer of the encoder.

Process Flow

An encoder receives a list of vectors as input. It processes this list by passing these vectors into a 'self-attention' layer, then into a feed-forward neural network, then sends out the output upwards to the next encoder.



The word at each position passes through a self-attention process. Then, they each pass through a feed-forward neural network -- the exact same network with each vector flowing through it separately.

Attention Mechanism – Basics and Motivation

What is Attention?

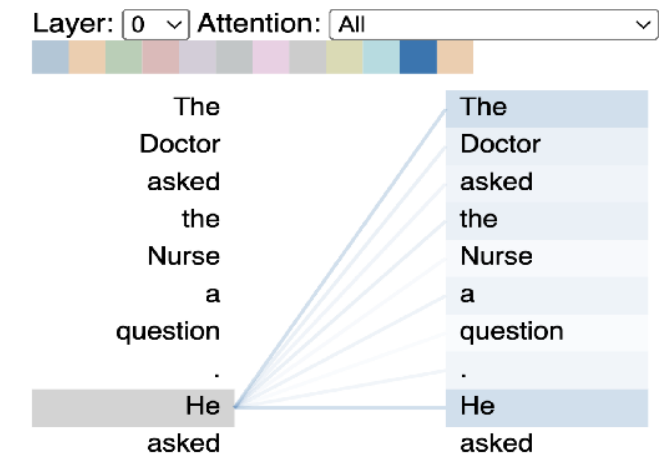
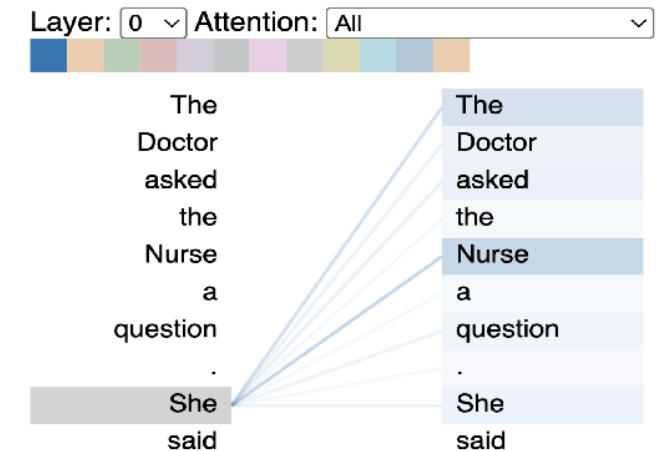
- A mechanism that allows a model to **focus on specific parts** of the input when producing each output
- Inspired by human selective focus in perception

Why Use Attention?

- Overcomes limitations of fixed-size context in RNNs
- Enables better handling of long sequences
- Improves performance in tasks like translation, summarization, and image captioning

Core Idea

- Each output element is computed as a **weighted sum** of all input elements
- The weights (attention scores) determine how much focus is given to each input



How Does Self Attention Work

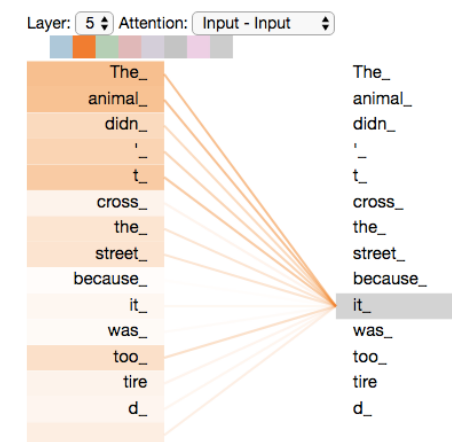


Self-Attention

- Say the following sentence is an input sentence we want to translate:

"The animal didn't cross the street because it was too tired"

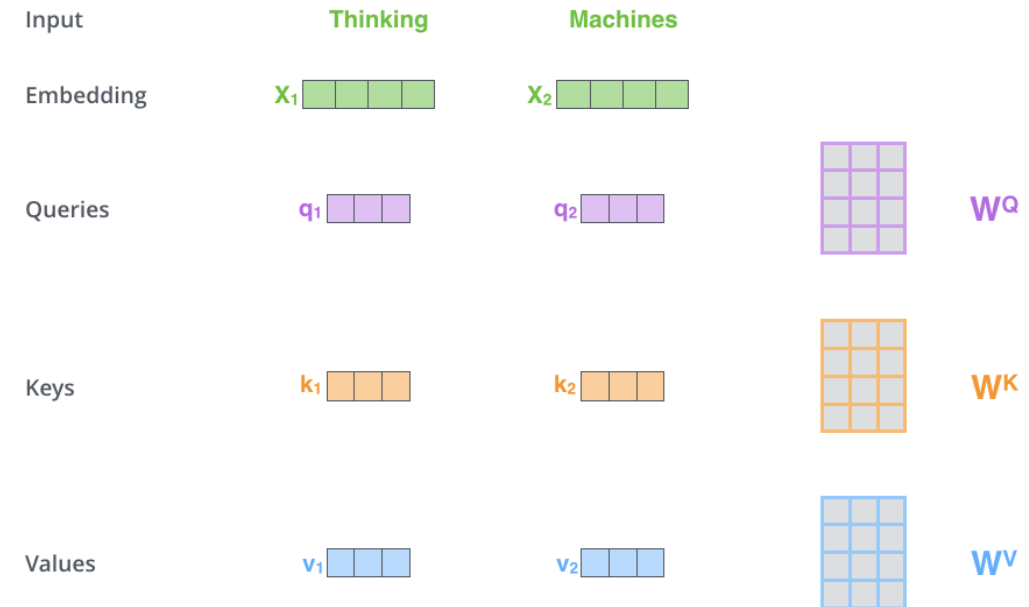
- What does "it" in this sentence refer to? Is it referring to the street or to the animal? It's a simple question to a human, but not as simple to an algorithm.
- When the model is processing the word "it", self-attention allows it to associate "it" with "animal".
- As the model processes each word (each position in the input sequence), self attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word.
- If you're familiar with RNNs, think of how maintaining a hidden state allows an RNN to incorporate its representation of previous words/vectors it has processed with the current one it's processing.
- Self-attention is the method the Transformer uses to bake the "understanding" of other relevant words into the one we're currently processing.



As we are encoding the word "it" in encoder #5 (the top encoder in the stack), part of the attention mechanism was focusing on "The Animal", and baked a part of its representation into the encoding of "it".

Key, Query, Value Vectors

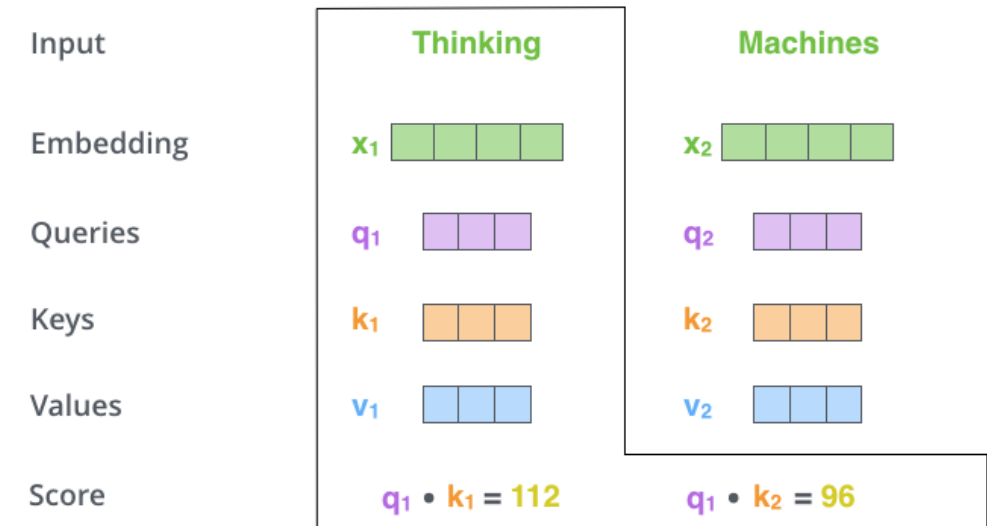
- The first step in calculating self-attention is to create three vectors from each of the encoder's input vectors (in this case, the embedding of each word).
- The “query”, “key”, and “value” vectors are abstractions that are useful for calculating and thinking about attention.
- So, for each word, we create a Query vector, a Key vector, and a Value vector.
- These vectors are created by multiplying the embedding by three matrices that we trained during the training process.
- Notice that these new vectors are smaller in dimension than the embedding vector.
- Their dimensionality is 64, while the embedding and encoder input/output vectors have dimensionality of 512.
- They don't HAVE to be smaller, this is an architecture choice to make the computation of multiheaded attention (mostly) constant.



Multiplying x_1 by the W^Q weight matrix produces q_1 , the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

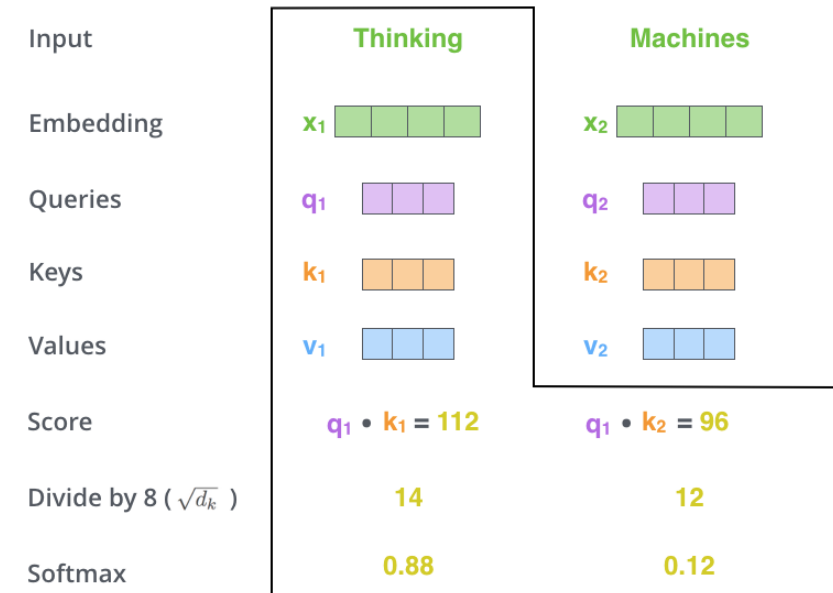
Key, Query, Value Vectors

- The **second step** in calculating self-attention is to calculate a score.
- Say we're calculating the self-attention for the first word in this example, "Thinking".
- We need to score each word of the input sentence against this word.
- The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.
- The score is calculated by taking the dot product of the **query vector** with the **key vector** of the respective word we're scoring.
- So if we're processing the self-attention for the word in position #1, the first score would be the dot product of **q1** and **k1**.
- The second score would be the dot product of **q1** and **k2**.



Key, Query, Value Vectors

- The **third and fourth steps** are to divide the scores by 8 (the square root of the dimension of the key vectors used in the paper – 64).
- This leads to having more stable gradients.
- There could be other possible values here, but this is the default), then pass the result through a softmax operation.
- Softmax normalizes the scores so they're all positive and add up to 1.
- This softmax score determines how much each word will be expressed at this position.
- Clearly the word at this position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word.



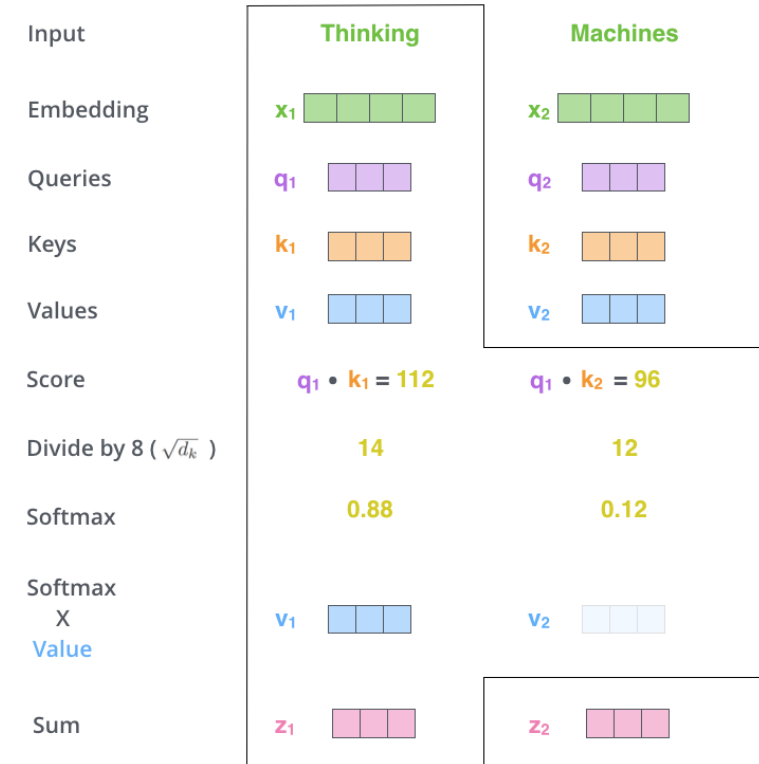
Key, Query, Value Vectors

- **The fifth step** is to multiply each value vector by the softmax score (in preparation to sum them up).
- The intuition here is to keep intact the values of the word(s) we want to focus on and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).
- **The sixth step** is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).

That concludes the self-attention calculation. The resulting vector is one we can send along to the feed-forward neural network.

In the actual implementation, however, this calculation is done in matrix form for faster processing.

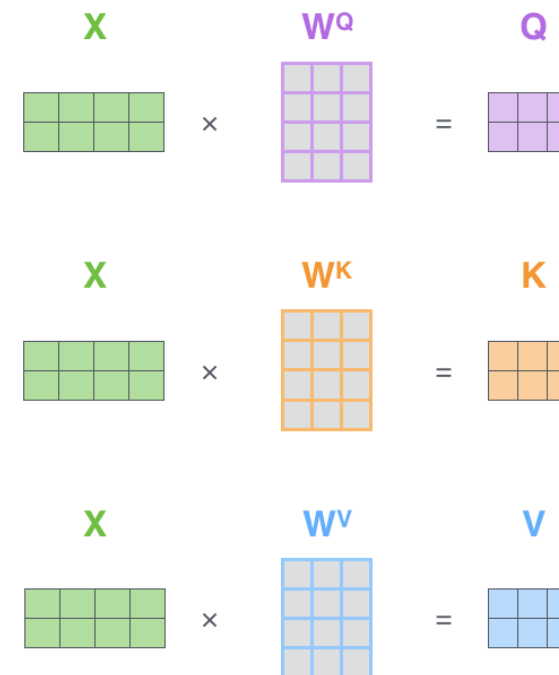
So let's look at that now that we've seen the intuition of the calculation on the word level.



Matrix Calculation of Self-Attention

The first step is to calculate the Query, Key, and Value matrices.

We do that by packing our embeddings into a matrix X , and multiplying it by the weight matrices we've trained (W^Q , W^K , W^V).



Every row in the X matrix corresponds to a word in the input sentence. We again see the difference in size of the embedding vector (512, or 4 boxes in the figure), and the $q/k/v$ vectors (64, or 3 boxes in the figure)

Matrix Calculation of Self-Attention

Finally, since we're dealing with matrices, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.

$$\text{softmax}\left(\frac{\begin{matrix} \text{Q} \\ \begin{matrix} \text{3x3 grid} \end{matrix} \end{matrix} \times \begin{matrix} \text{K}^T \\ \begin{matrix} \text{3x3 grid} \end{matrix} \end{matrix}}{\sqrt{d_k}}\right) \begin{matrix} \text{V} \\ \begin{matrix} \text{3x3 grid} \end{matrix} \end{matrix}$$
$$= \begin{matrix} \text{Z} \\ \begin{matrix} \text{3x3 grid} \end{matrix} \end{matrix}$$

The self-attention calculation in matrix form

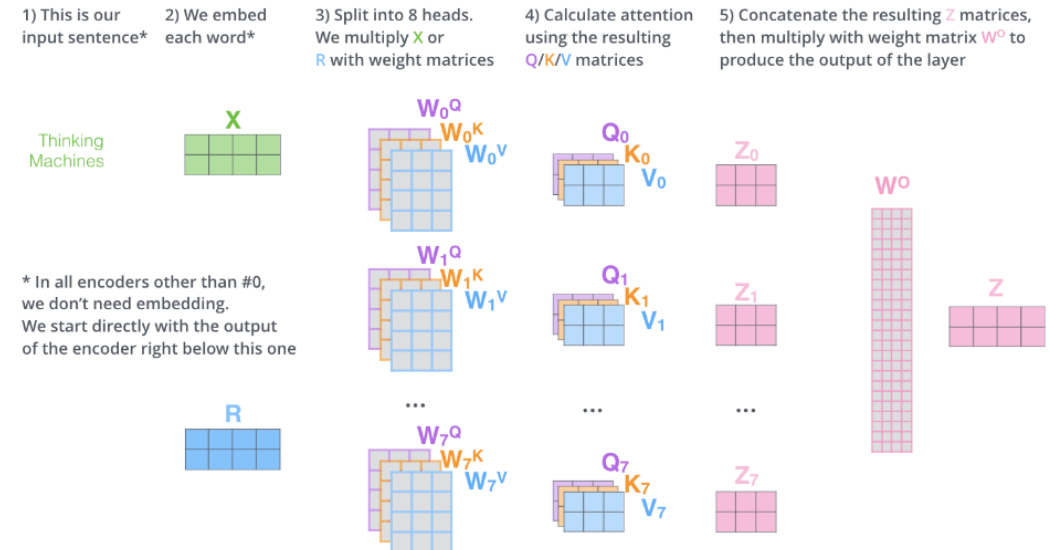


Multi-Head Attention

The paper further refined the self-attention layer by adding a mechanism called “multi-headed” attention. This improves the performance of the attention layer in two ways:

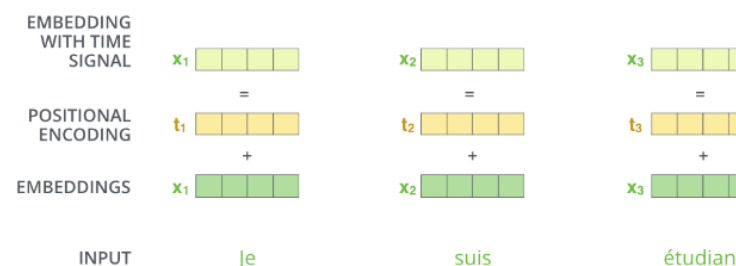
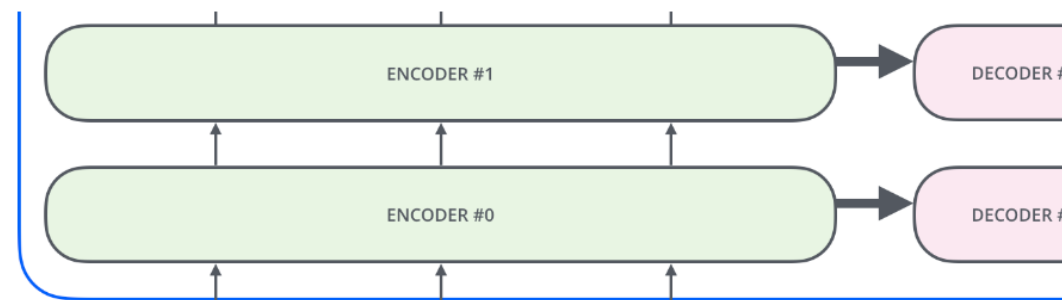
1. It expands the model’s ability to focus on different positions. For example, z_1 contains a little bit of every other encoding, but it could be dominated by the actual word itself.

2. It gives the attention layer multiple “representation subspaces



Positional Encoding

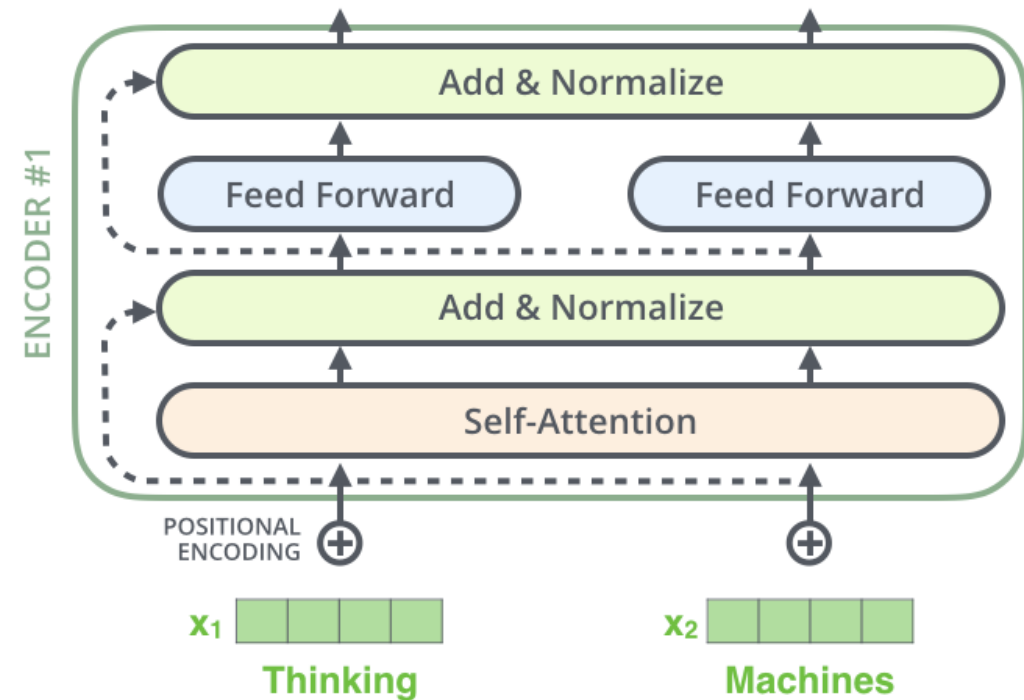
- One thing that's missing from the model as we have described it so far is a way to account for the order of the words in the input sequence.
- To address this, the transformer adds a vector to each input embedding.
- These vectors follow a specific pattern that the model learns, which helps it determine the position of each word, or the distance between different words in the sequence
- The intuition here is that adding these values to the embeddings provides meaningful distances between the embedding vectors.



To give the model a sense of the order of the words, we add positional encoding vectors -- the values of which follow a specific pattern.

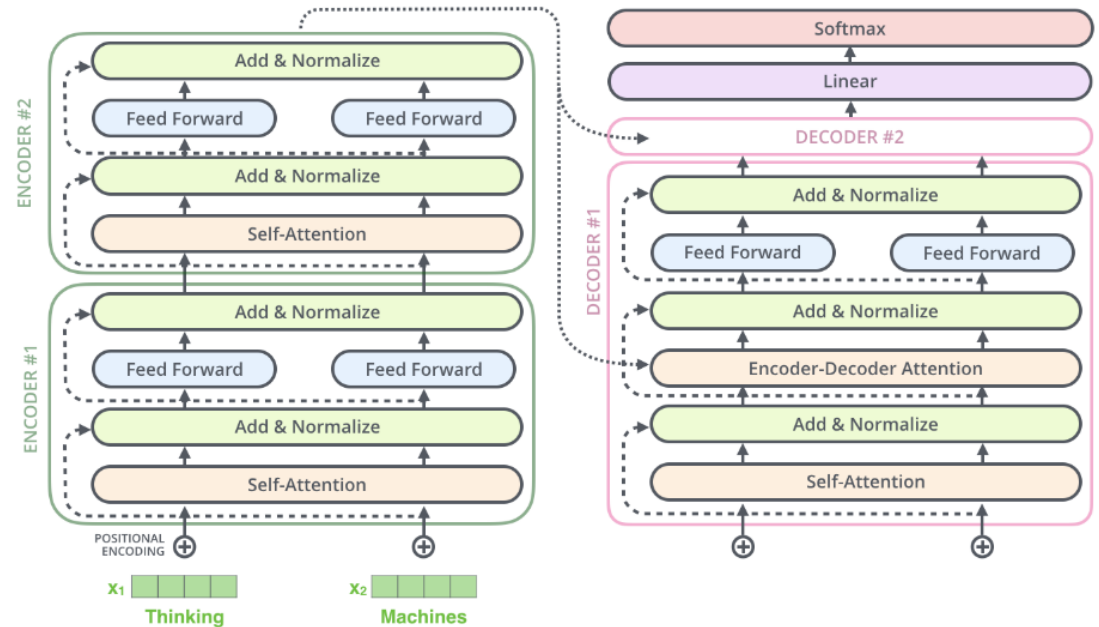
The Residual Connections

One detail in the architecture of the encoder that we need to mention before moving on, is that each sub-layer (self-attention, ffn) in each encoder has a residual connection around it, and is followed by a [layer-normalization](#) step.



The Residual Connections

The residual connections goes for the sub-layers of the decoder as well.



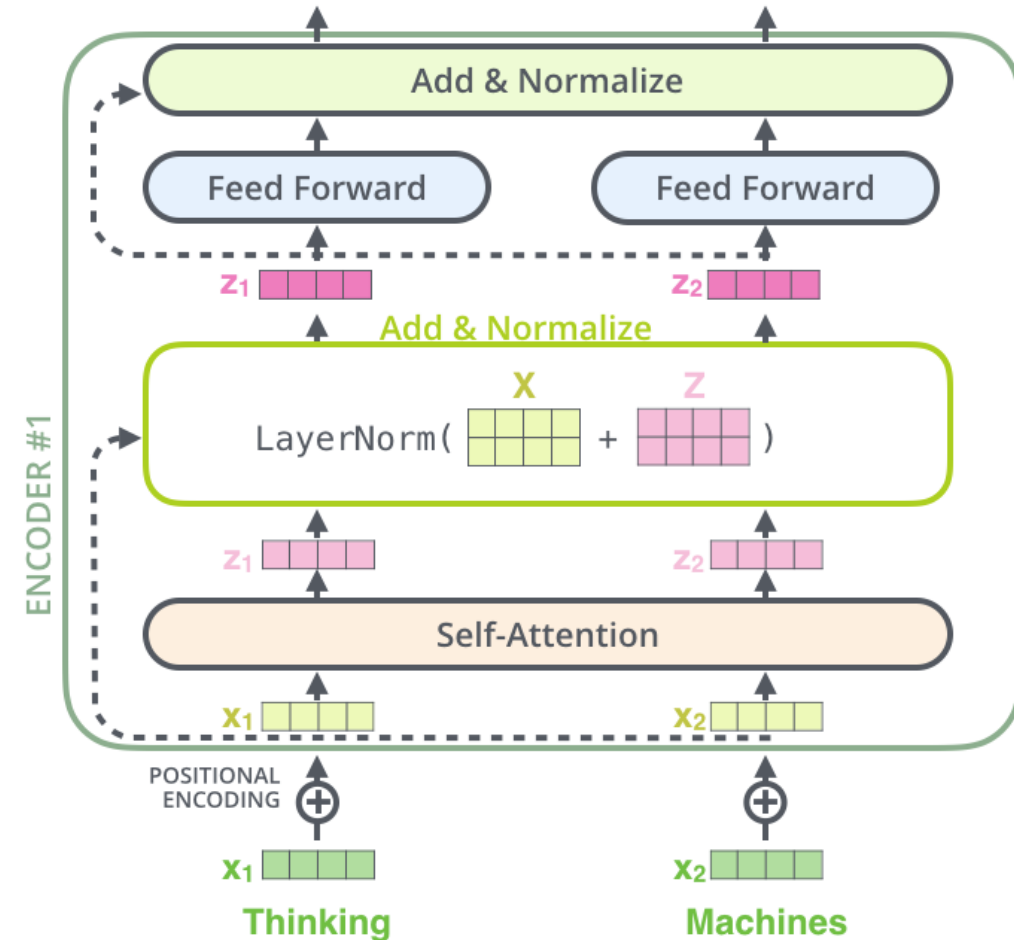
Layer Normalization

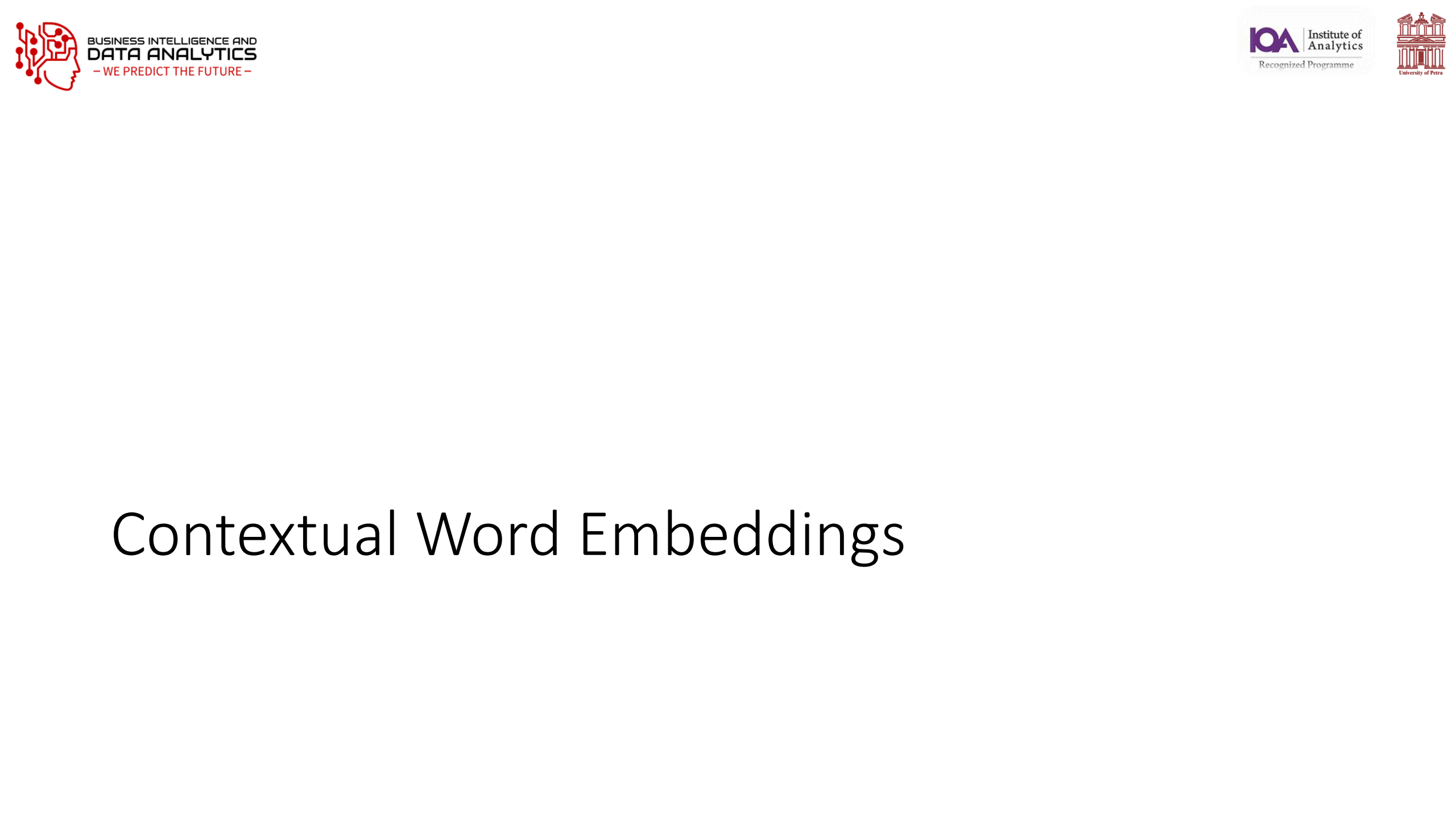
Layer normalization in Transformers stabilizes and accelerates training by **normalizing the inputs across the features for each token** (rather than across the batch like batch norm).

In brief:

It ensures that the inputs to each layer (like self-attention or feedforward) have **zero mean and unit variance**, which helps:

- Prevent internal covariate shift,
- Improve gradient flow,
- Enable faster convergence.
- In Transformers, it's usually applied either **before or after** each sub-layer (attention or feedforward), often with a residual connection.





Contextual Word Embeddings

Contextual Word Embeddings (BERT and GPT)

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

- **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**

BERT – Overview and Architecture

What is BERT?

- A **pre-trained language model** based on the **Transformer encoder**
- Developed by Google in 2018
- Reads text **bidirectionally**, enabling deep contextual understanding

Key Ideas

- Uses only the **encoder** stack of the Transformer
- Pre-trained on large text corpora, then fine-tuned on specific tasks

Pretraining Objectives

- **Masked Language Modeling (MLM)**: Predict randomly masked words in a sentence
- **Next Sentence Prediction (NSP)**: Predict if one sentence follows another

Applications

- Sentiment Analysis
- Question Answering
- Named Entity Recognition
- Text Classification
- Semantic Search

GPT – Overview and Architecture

What is GPT?

- A family of **Transformer-based language models** developed by OpenAI
- Uses only the **decoder stack** of the original Transformer architecture
- Trained with **causal (autoregressive) language modeling** to predict the next token

Architecture Highlights

- Input: Token embeddings + positional encodings
- Multiple stacked **Transformer decoder blocks**
- Each block includes:
 - Masked self-attention (prevents future token access)
 - Feedforward layer
 - Residual connections + layer normalization
- Final layer: Linear + Softmax for token prediction

Training Objective

- Predict the next token in a sequence

Strengths, Variants, and Applications

Strengths of GPT

- **Unidirectional context:** Efficient for generation
- **Scalable:** Performance improves significantly with size
- **Few-shot and zero-shot learning:** Can generalize without fine-tuning

GPT Variants

- **GPT-1:** Introduced the pretrain-then-finetune paradigm
- **GPT-2:** Scaled up model size, trained on web-scale data
- **GPT-3:** 175B parameters, enabled in-context learning
- **GPT-4:** Multimodal, stronger reasoning and generalization

Applications

- Text generation (e.g., chat, storytelling, code)
- Summarization
- Translation
- Question answering
- Semantic search and reasoning tasks

Comparison to BERT

- GPT: Autoregressive, great for **generation**
- BERT: Bidirectional, great for **understanding**

Practical Implementation: BERT in Python

```
from transformers import BertTokenizer, BertModel
import torch

# Load pretrained BERT
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

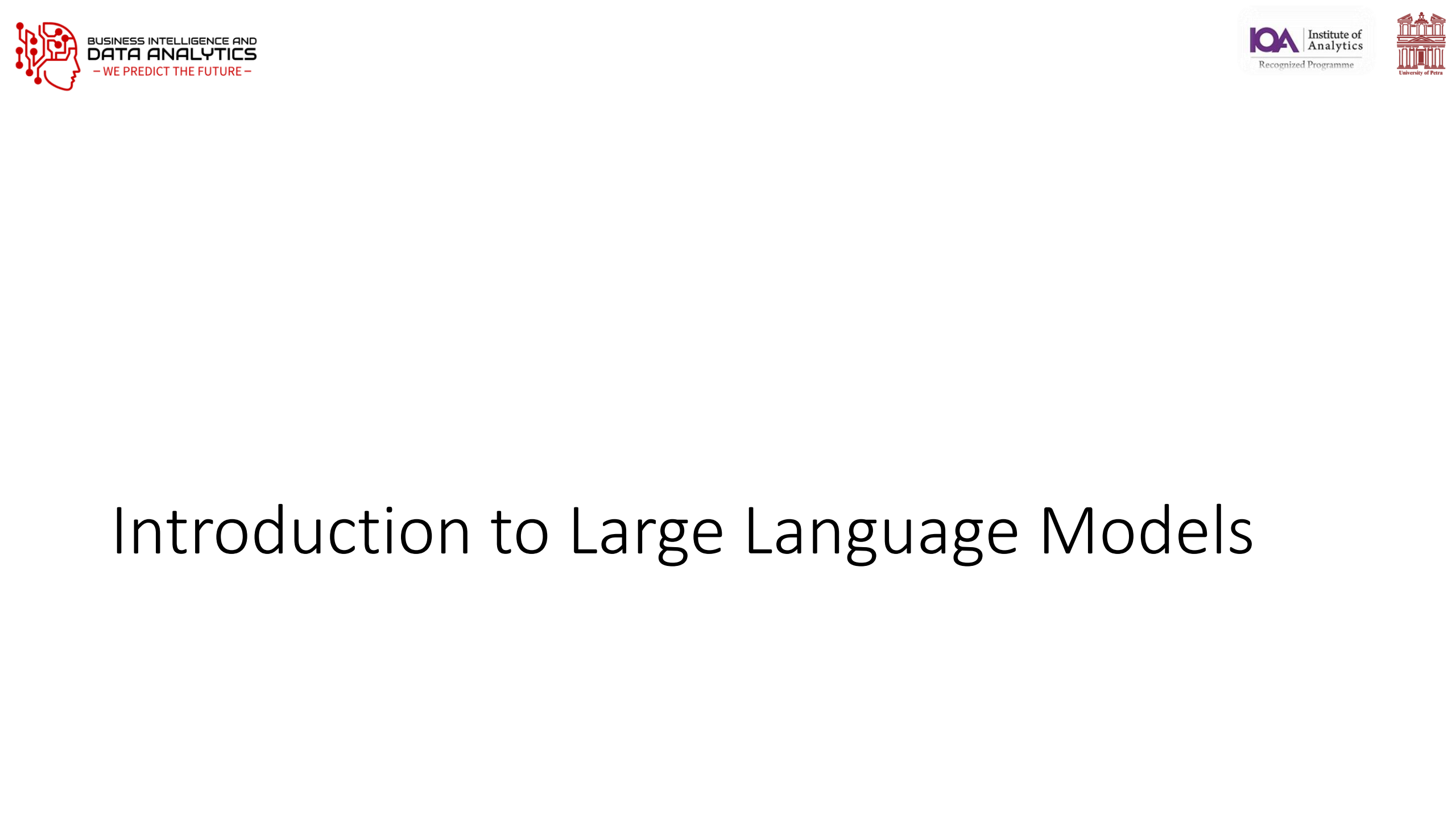
# Sentence
sentence = "He went to the bank to deposit money."

# Tokenize
inputs = tokenizer(sentence, return_tensors='pt')

# Get outputs
with torch.no_grad(): # No training, just inference
    outputs = model(**inputs)

# Get the hidden states (embeddings)
embeddings = outputs.last_hidden_state # Shape: (batch_size, sequence_length,
hidden_size)

print(embeddings.shape) # Example output: torch.Size([1, 11, 768])
```



Introduction to Large Language Models

Generative AI

- Generative AI, also known as GenAI, refers to a type of artificial intelligence that uses machine learning algorithms to generate new, synthetic data, such as images, videos, music, text, and more.
- Unlike traditional AI, which focuses on analyzing and processing existing data, generative AI models create novel data samples that resemble existing data.
- Some common applications of generative AI include:
 - Text generation: Creating realistic text, such as articles, stories, or chatbot responses, for use in applications like content generation, language translation, and conversational interfaces.
 - Image and video generation: Generating realistic images, videos, or 3D models for use in various industries, such as entertainment, advertising, and architecture.
 - Music and audio generation: Generating music, sound effects, or voice synthesis for use in music production, audiobooks, or voice assistants.
 - Data augmentation: Generating new data samples to augment existing datasets, which can help improve the performance of machine learning models.
 - Art and design: Creating new artistic styles, designs, or patterns for use in various creative fields.

Introduction to LLMs

- Large Language Models (LLMs) are AI models trained on massive amounts of text data to generate human-like text, answer questions, and assist in various applications.
- Some of the most advanced LLMs today come from leading AI research labs, each with unique architectures, capabilities, and use cases.
- This lecture covers the major LLMs, their features, differences, and code examples to interact with them.

Well Known Models:

Company	Model(s)	Key Features
OpenAI	GPT-4, GPT-4-turbo, GPT-3.5	Strong reasoning, multimodal (text & image in GPT-4), API for developers
Anthropic	Claude 1, 2, 3	Focus on safety, long-context handling, efficient responses
Google	Gemini 1, Gemini 1.5	Strong in reasoning, image understanding, Google search integration
Cohere	Command R+, Command R	Enterprise-focused, retrieval-augmented generation (RAG)
Meta	LLaMA 2, LLaMA 3 (upcoming)	Open-source, efficient models for researchers
Perplexity	Perplexity AI	AI-powered search engine, real-time web browsing

What is OpenAI?

- OpenAI is an artificial intelligence research and deployment company.
- Founded in December 2015 by Elon Musk, Sam Altman, Greg Brockman, Ilya Sutskever, John Schulman, and Wojciech Zaremba.
- The organization's mission is to ensure that artificial general intelligence (AGI) benefits all of humanity.
- Initially a non-profit, it transitioned in 2019 to a "capped-profit" model through the creation of OpenAI LP to attract capital while maintaining mission alignment.

Timeline of Key Models

Model	Release Year	Parameters	Training Data	Key Capabilities	Notes
GPT	2018	~117 million	Books, websites, Wikipedia, Common Crawl	Basic text generation and completion	First Generative Pretrained Transformer; proof of concept
GPT-2	2019	1.5 billion	8 million documents from the internet	Coherent paragraphs, summaries, text generation	Withheld initially due to concerns about misuse
GPT-3	2020	175 billion	~570GB filtered text (Common Crawl + curated data)	Question answering, translation, text generation, code assist	Major leap in coherence and contextual understanding
Codex	2021	Built on GPT-3	Text + 100M+ GitHub code repositories	Code generation, code completion, natural language → code	Powers GitHub Copilot
ChatGPT	2022	GPT-3.5 fine-tuned	GPT-3.5 + Human Feedback	Conversational AI, tutoring, brainstorming, creative writing	Optimized for dialogue using RLHF
GPT-4	2023	Estimated 1+ trillion*	Diverse multimodal data (text, image, web)	Multilingual, logic reasoning, coding, image input (limited)	Parameters not publicly confirmed
GPT-4o	2024	Unknown (GPT-4 variant)	Text, images, audio	Real-time interaction with vision, voice, and text capabilities	"o" stands for "omni"; supports seamless multimodal input/output

OpenAI and Microsoft Partnership

- Microsoft is a key partner and investor in OpenAI.
- Azure cloud services are used to deploy OpenAI models.
- OpenAI tools are integrated into Microsoft products like Word, Excel, and GitHub Copilot.
- Microsoft has exclusive licenses to integrate GPT models in enterprise settings.

How OpenAI Models Work

- OpenAI's models are based on the Transformer architecture, introduced by Vaswani et al. in 2017.
- These models are trained on massive datasets containing text, code, and other data from the internet.
- The models learn to predict the next word or token in a sequence (autoregressive training).
- Some models, like ChatGPT, use Reinforcement Learning from Human Feedback (RLHF) to improve response quality.
- GPT-4o is multimodal: it can accept and generate text, images, and audio.

Core OpenAI Technologies and Products

- **GPT (Generative Pretrained Transformer):** Language model that generates human-like text.
- **ChatGPT:** AI chatbot used for conversation, education, and productivity.
- **DALL\ue0b7E:** Text-to-image model that generates pictures from textual descriptions.
- **Codex:** Powers AI programming assistants like GitHub Copilot.
- **Whisper:** Open-source automatic speech recognition system.

Ethical Considerations and Safety Measures

- OpenAI aims to make AI systems that are safe and aligned with human values.
- Safety teams work on reducing harmful outputs and biases.
- Content moderation tools are integrated in public-facing applications.
- OpenAI supports responsible use and restricts applications related to misinformation, surveillance, or violence.

OpenAI Compared to Other AI Labs

- **DeepMind (Google):** Focuses on scientific discovery and AGI through models like AlphaGo and AlphaFold.
- **Anthropic:** Founded by former OpenAI researchers; emphasizes alignment and safety.
- **Meta AI (Facebook):** Focuses on open research in large language models and computer vision.
- OpenAI uniquely balances research, product development, and commercial deployment.

Applications Across Industries

- **Education:** AI tutoring, essay writing support, language learning.
- **Business:** Report generation, customer support automation, data analysis.
- **Healthcare:** Administrative assistance, medical information summarization.
- **Software Development:** Coding help through Copilot and Codex.
- **Creative Work:** Art generation, script writing, music composition.

Concerns and Challenges

- **Bias:** Models may reflect societal biases present in training data.
- **Privacy:** Risk of sensitive data being used in model training or output.
- **Misinformation:** Ability to generate plausible but false content.
- **Job Displacement:** Automation of cognitive tasks may affect employment.
- OpenAI acknowledges these concerns and actively works on mitigation strategies.

Future of OpenAI


- Continued improvement in model capabilities and safety.
- Focus on user customization (e.g., custom GPTs).
- Expansion of real-time multimodal interaction (text, voice, vision).
- Engagement with policymakers and public institutions to guide ethical AI development.
- OpenAI's long-term goal is to achieve safe, beneficial artificial general intelligence (AGI).

Conclusion and Discussion Points

- OpenAI is a leader in AI research and product development.
- It has transformed industries and raised important ethical questions.
- It remains committed to making AGI safe and beneficial.
- **Questions for Discussion:**
 - Should access to powerful AI models be open or restricted?
 - How should governments regulate AI?
 - What are the roles of education and ethics in the age of AI?

groq Playground

groqcloud

Playground API Keys Dashboard Documentation ⚡ Upgrade  Personal

Playground Chat Studio Llama 4 Scout 17B 16E ↕ 📄 </> View code

SYSTEM Enter system message (Optional)

Welcome to the Playground

- You can start by typing a message
- Click submit to get a response
- Use the <> icon to view the code

USER User Message... 📄 Submit Ctrl + ↵

PARAMETERS 📄

Temperature 1

Max Completion Tokens 1024

Stream ☒

JSON Mode ☐

Advanced ^

Moderation: llamaguard ☐

Top P 1

Seed

Stop Sequence

<https://console.groq.com/playground>

Create API Key – groq.com


Playground

API Keys

Dashboard

Documentation



⚡ Upgrade

 Personal

API Keys

Create API Key

Manage your API keys. Remember to keep your API keys safe to prevent unauthorized access.

NAME	SECRET KEY	CREATED	LAST USED	USAGE (24HRS)	
initial_testing	gsk_...h1Xg	26/02/2025	29/04/2025	12 API Calls	<div><div></div><div></div></div>

Access Cloud Based LLMs (groq.com)

```
import os
from groq import Groq

# Load the Groq API key from an environment variable
api_key = os.environ.get("GROQ_API_KEY")

if not api_key:
    print("Please set the GROQ_API_KEY environment variable.")
    exit()

# Initialize the Groq client
client = Groq(api_key=api_key)

# Define a sample prompt
prompt = "What is the capital of France?"

# Create a chat completion
completion = client.chat.completions.create(
    model="compound-beta",
    messages=[
        {"role": "user", "content": prompt},
    ],
    max_tokens=2048,
    stop=None,
)

# Print the response
print(completion.choices[0].message.content)
```

1. Import necessary libraries:

- `import os`: You import Python's `os` module, which provides functions to interact with the operating system (like accessing environment variables).
- `from groq import Groq`: You import the `Groq` client, which is used to communicate with Groq's API to call language models.

2. Load the API key:

- `api_key = os.environ.get("GROQ_API_KEY")`: This attempts to read the environment variable named `GROQ_API_KEY` that should contain your API key.
- If the key is not found (`if not api_key:`), the script prints a message asking the user to set it and exits immediately (`exit()`).

3. Initialize the Groq client:

- `client = Groq(api_key=api_key)`: A new instance of the Groq client is created, authenticated using the loaded API key.

4. Prepare a prompt:

- `prompt = "What is the capital of France?"`: You define the user's question as a simple string, intended to be sent to the language model.

5. Call the language model:

- `client.chat.completions.create(...)`: This sends a chat-style API call to the Groq server:
 - `model="compound-beta"` specifies the model to use.
 - `messages=[{"role": "user", "content": prompt}]` creates a conversation history containing only one user message: your question.
 - `max_tokens=2048` defines the maximum length of the model's reply (up to 2048 tokens).
 - `stop=None` means no special stopping sequence is set; the model will stop when it reaches the end or the token limit.

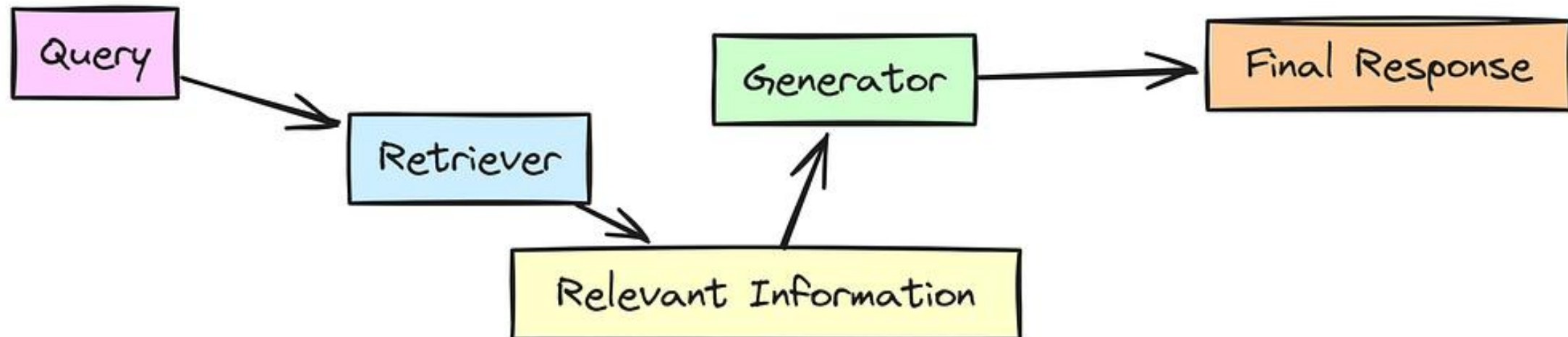
6. Print the response:

- `print(completion.choices[0].message.content)`: After getting the model's reply, you access the first choice's content and print it to the console.

Retrieval Augmented Generation (RAG)

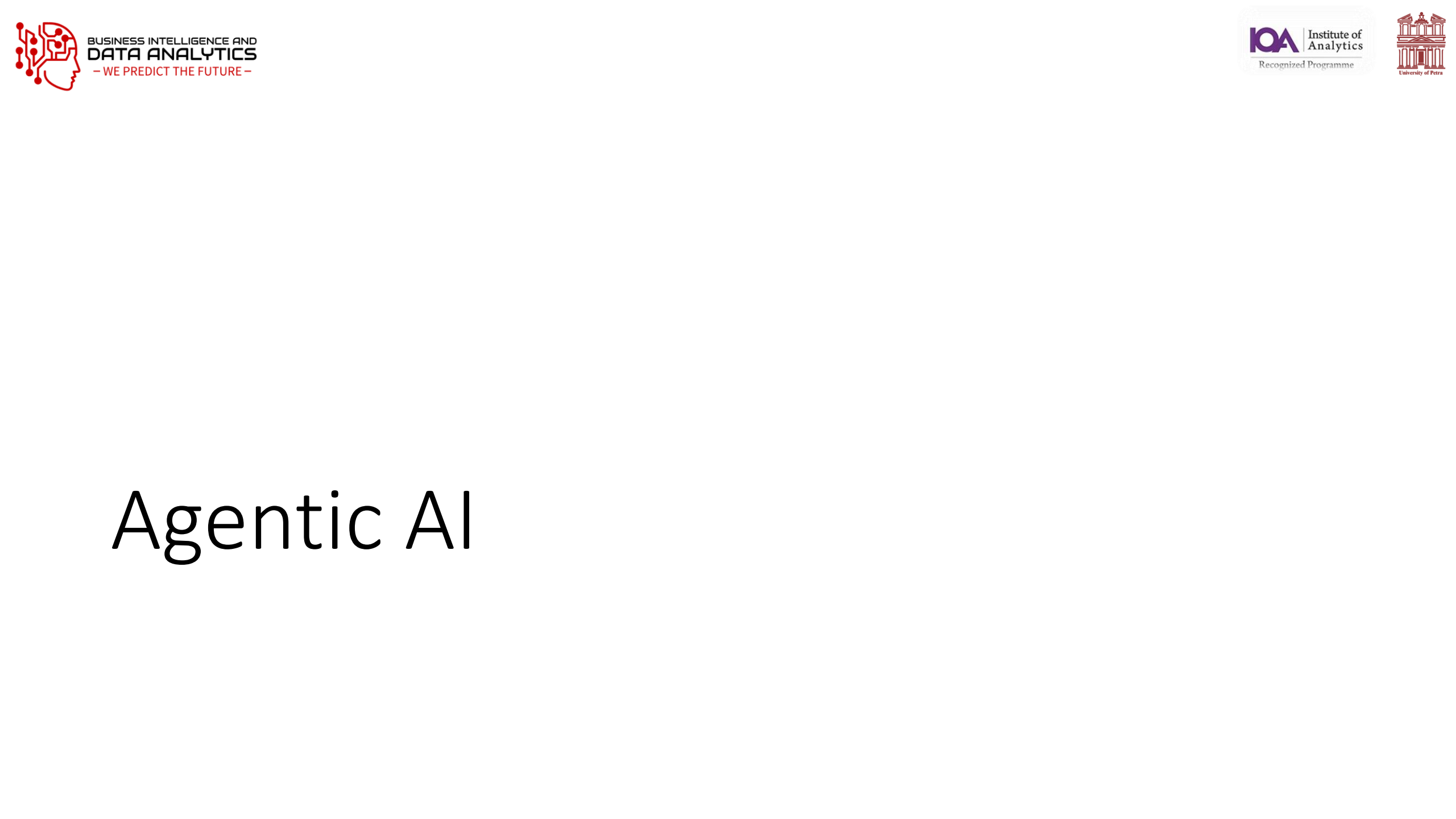
In this process:

- The **Query** is the user input.
- The **Retriever** searches the knowledge base and brings back relevant documents.
- The **Generator** combines the retrieved information and generates the final response.



Next

- Introduce deep neural networks e.g. cnn, rnn, lstm or go directly to BERT (next)
- Attention Role in the process and its Mechanism
- Transformer Architecture, Attention is all you need
- Mohammad Zarrar post about BERT on linkedin
- Basic Topics:
 - LLM Terminology and Playground (temperature, top, context window...etc)
 - Tokenizer Playground
 - Prompt Engineering
- Calling APIS
- RAG Case
- Agentic Case
- Local LLM i.e. Ollama
- N8n



Agentic AI

References

- Mathematics for Machine Learning / Dr. Naveed R. Butt @ GIKI