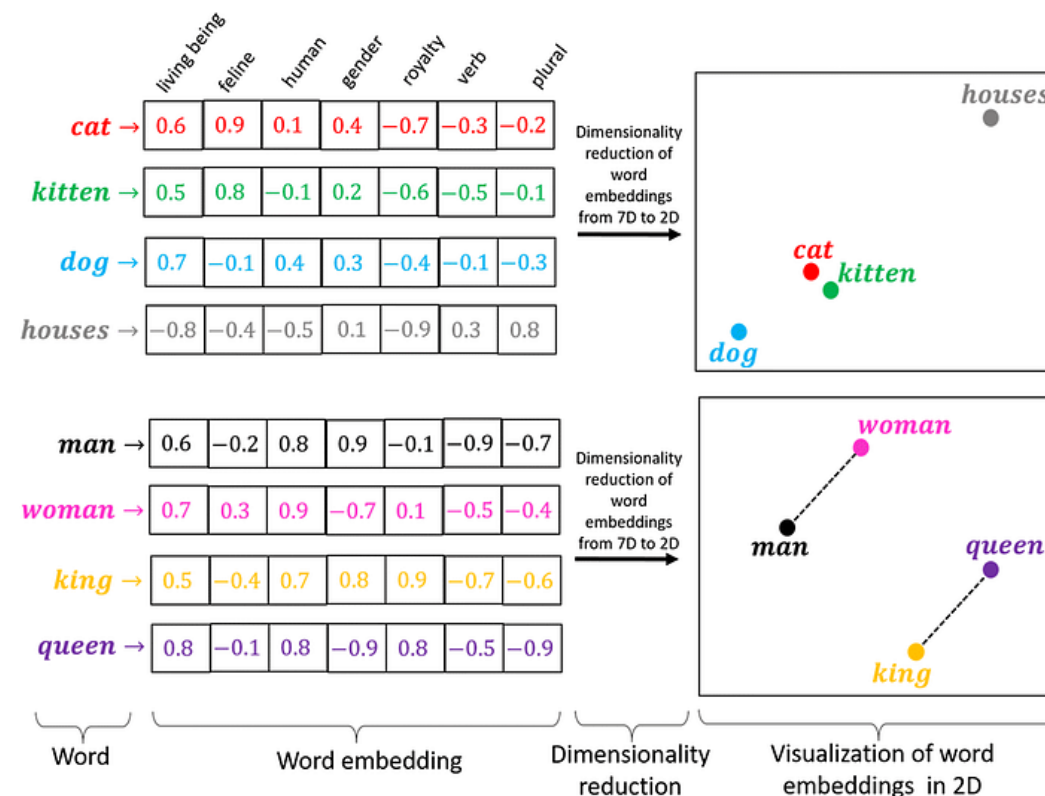


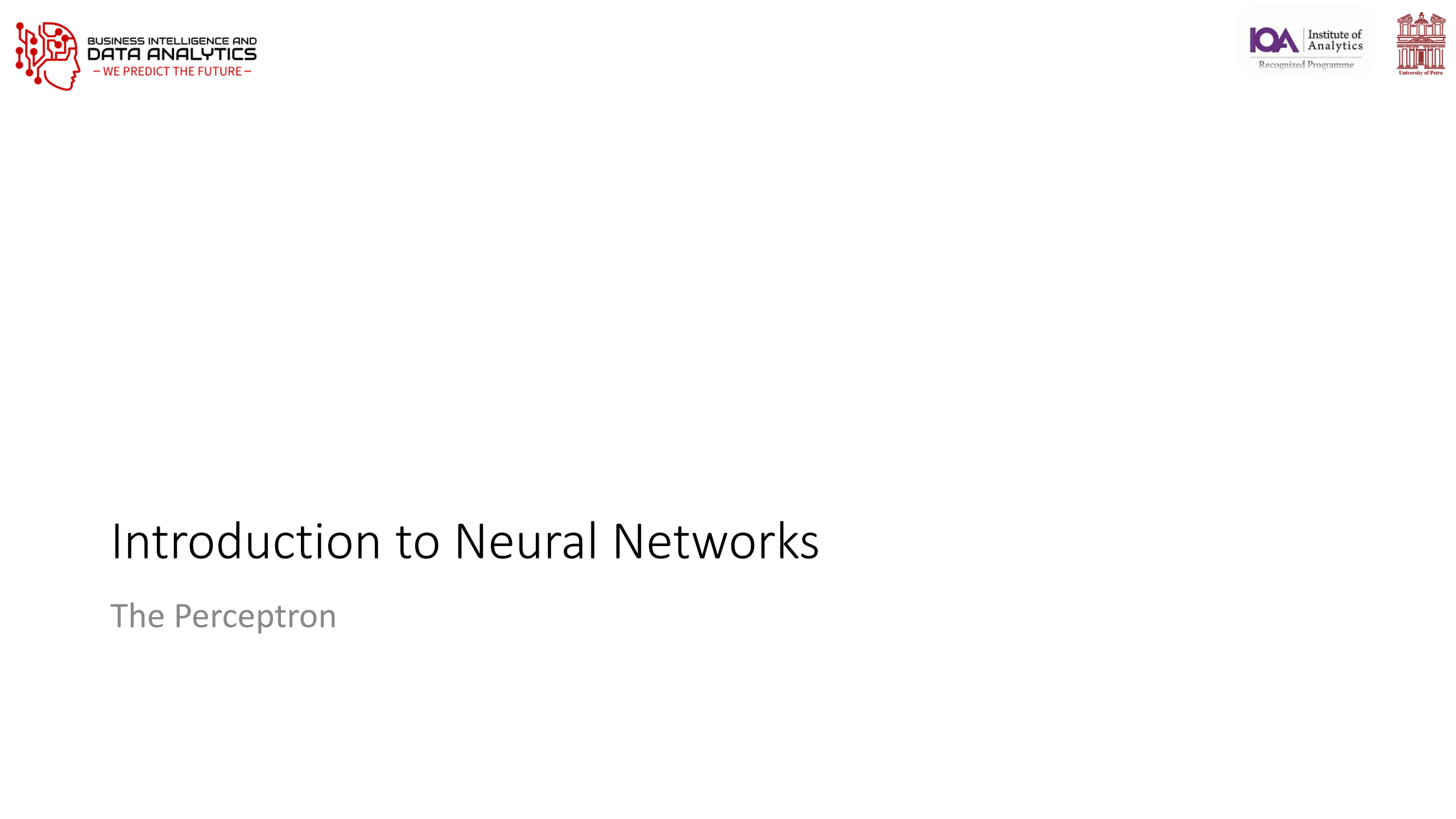
307307

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

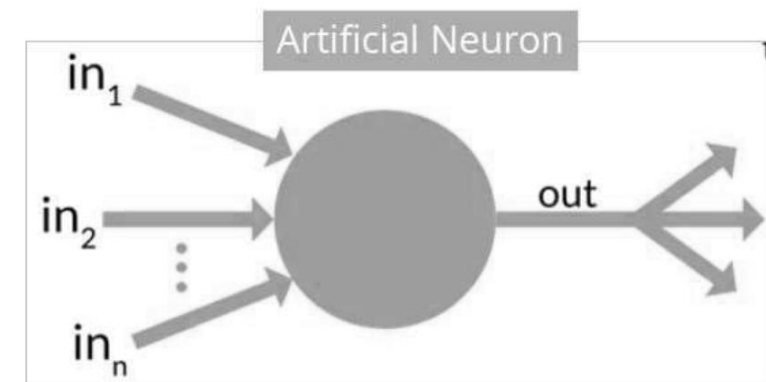
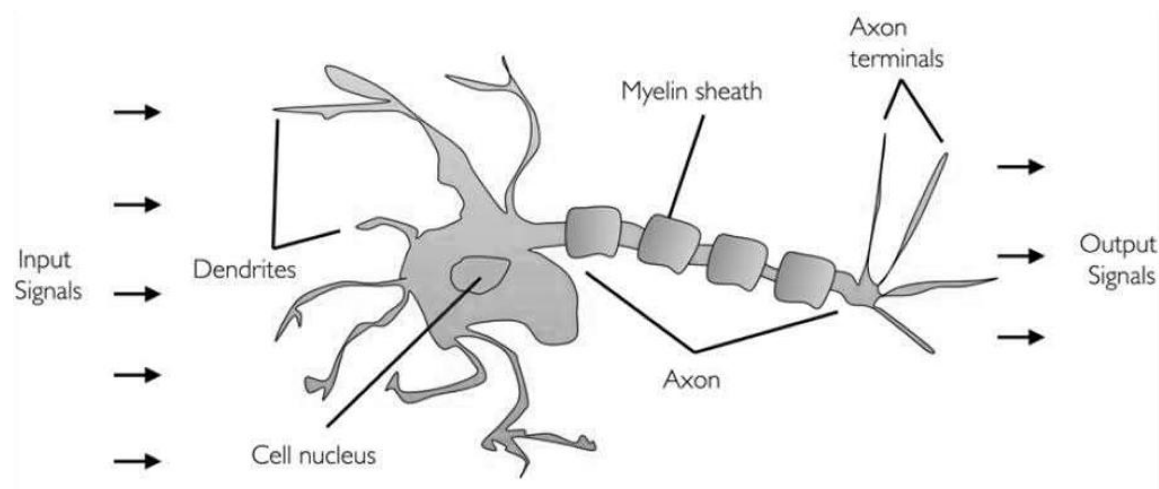
- In 1943, researchers Warren McCulloch and published their first concept of simplified brain cell.
- This was called McCulloch-Pitts (MCP) neuron.
- They described such a nerve cell as a simple logic gate with binary outputs.
- Multiple signals arrive at the dendrites and are then integrated into the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

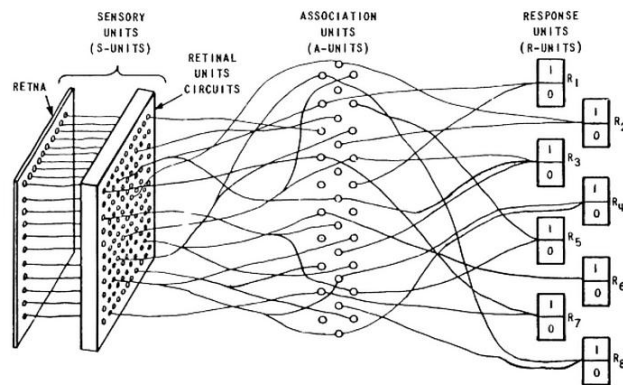


Walter Harry Pitts, Jr.
(1923 – 1969)

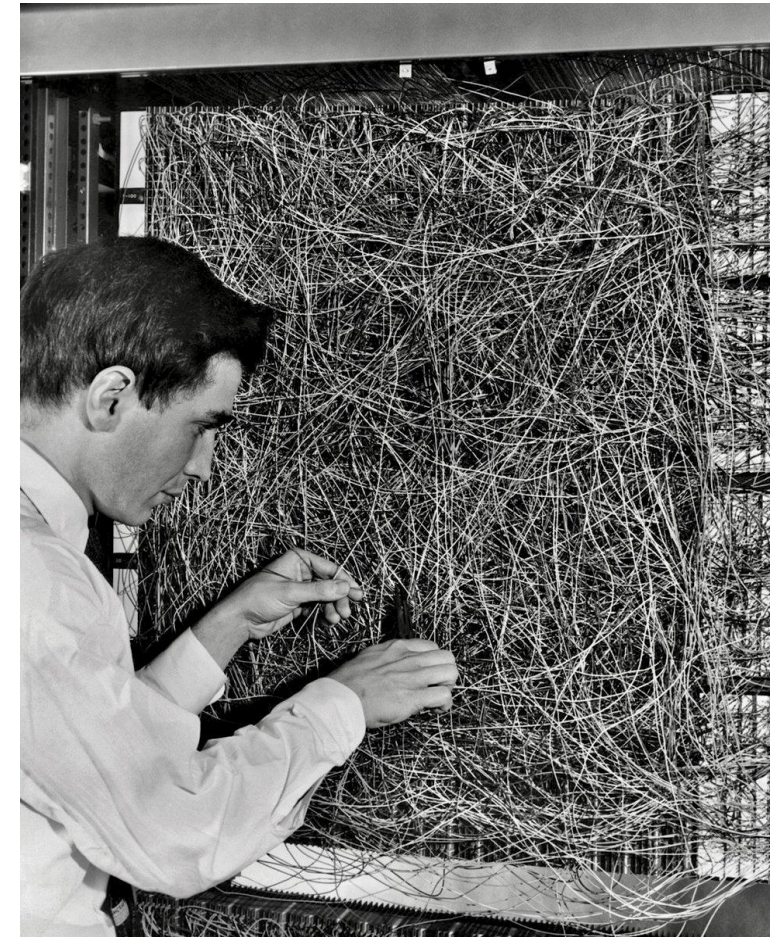


The Perceptron: Building Block of Neural Networks

- 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

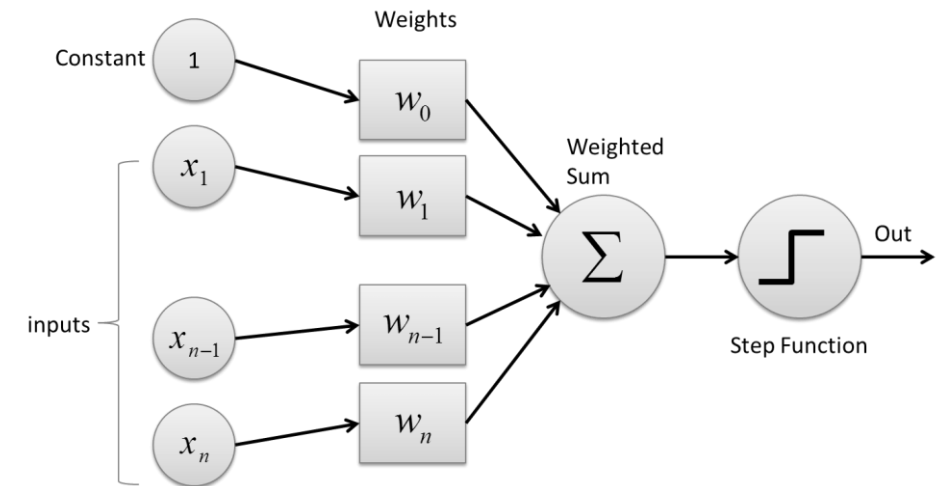


F. Rosenblatt



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

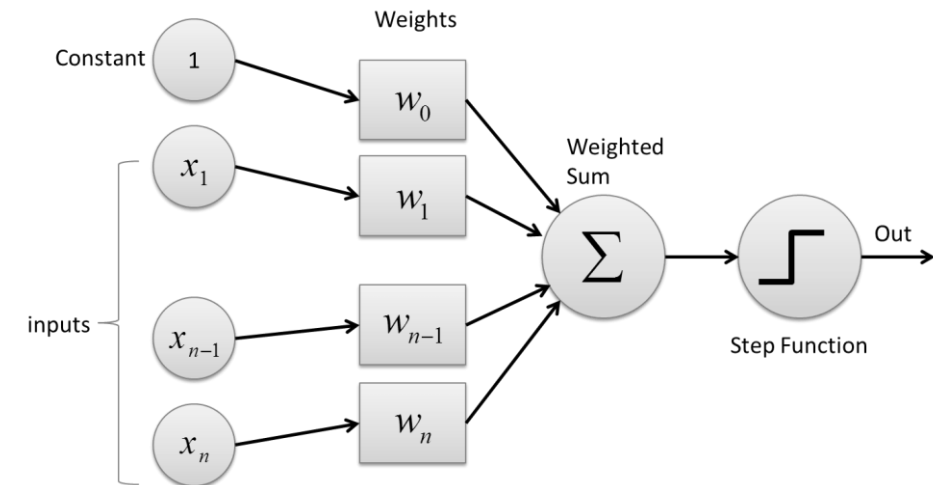


How a Perceptron Works

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

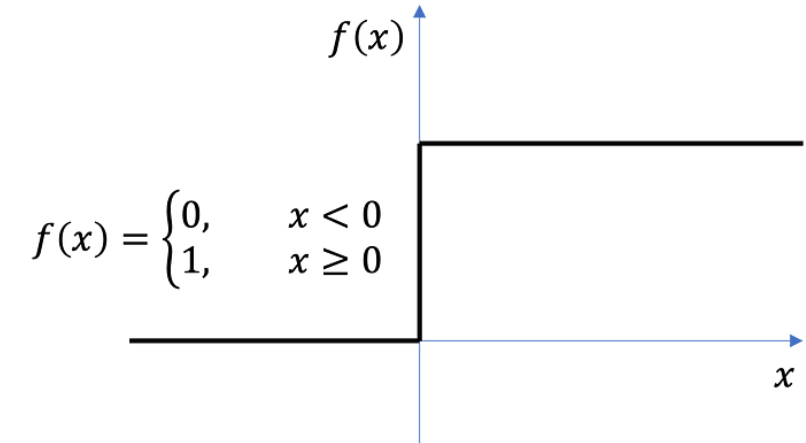
Mathematically:

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



Perceptron Activation Function

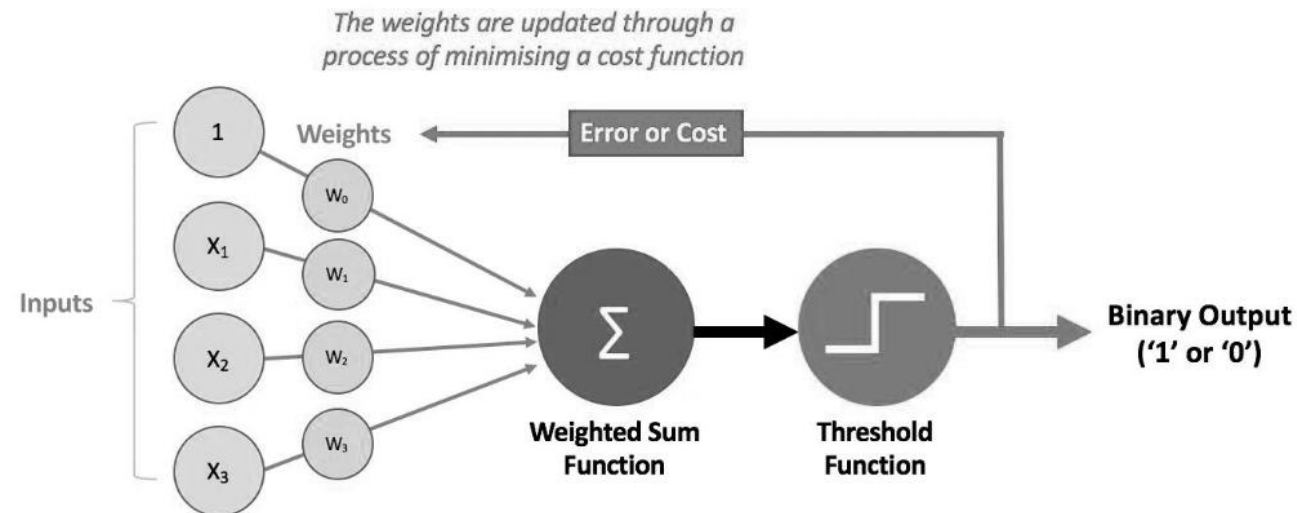
- **Step Function:**
 - Output: 1 if $z \geq 0$, 0 if $z < 0$
 - Used in original perceptrons
 - Not differentiable at 0



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: $\text{error} = y - \text{output} = 0 - 1 = -1$
- 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]: $0.2 \times 1 + 0.2 \times 1 - 0.2 = 0.2 > 0 \rightarrow \text{output } 1$

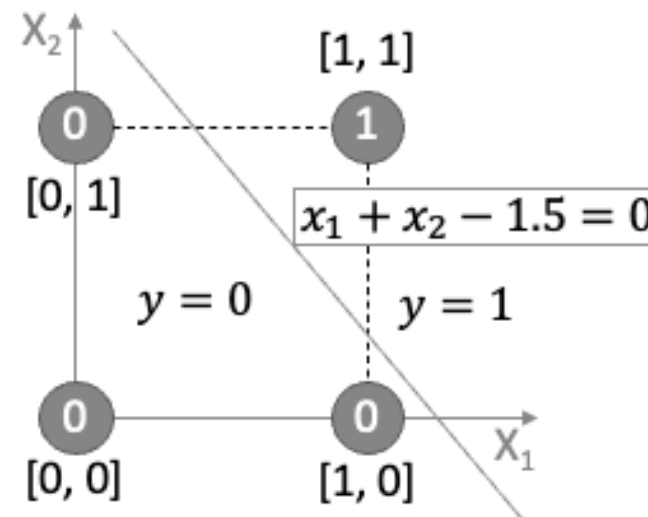
This perceptron implements the AND gate logic.

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

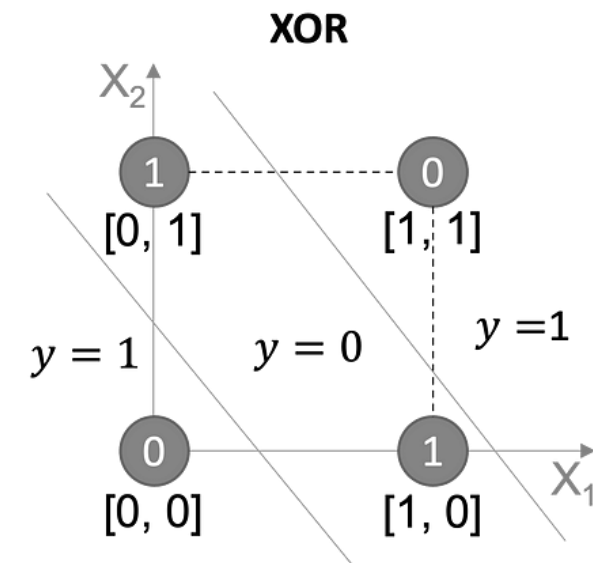
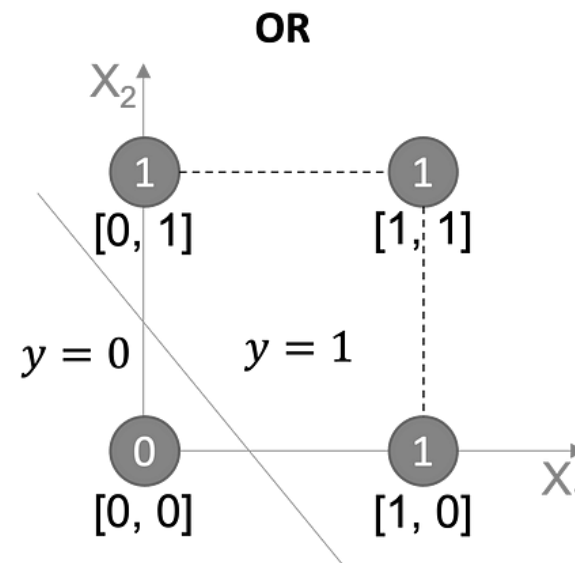
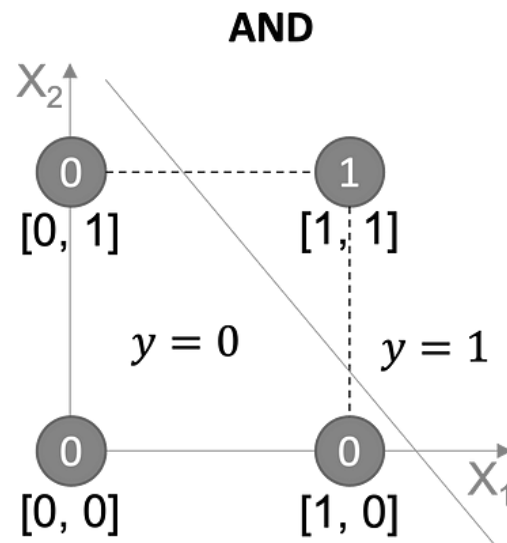
- The perceptron learns a decision boundary: $w_1x_1 + w_2x_2 + b = 0$
- Points above the line are classified as 1
- Points below the line are classified as 0
- For AND gate, only the point (1,1) should be above the line

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

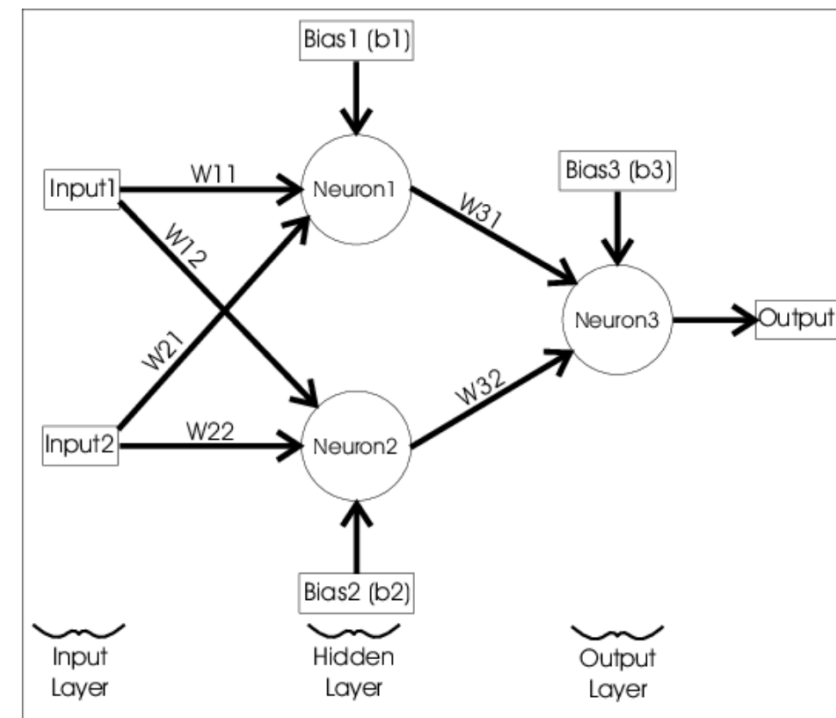
- Receives the raw input features
- One neuron per input feature
- No computation occurs here; inputs are simply passed forward

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

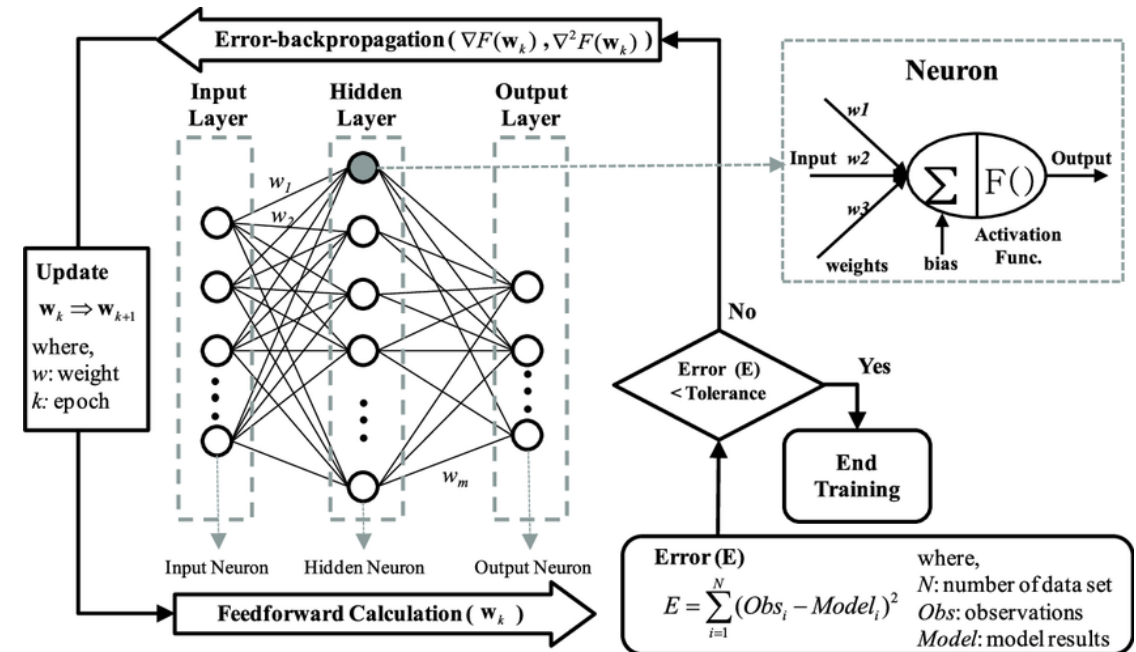


The Neural Network Model to solve the XOR Logic (from: <https://stopsmokingaids.me/>)

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).

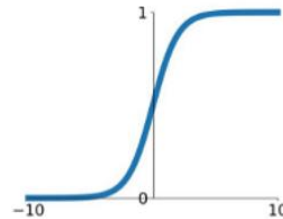


Activation Functions other than Step Function

- 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).

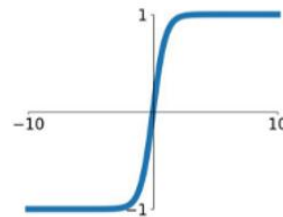
Sigmoid

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



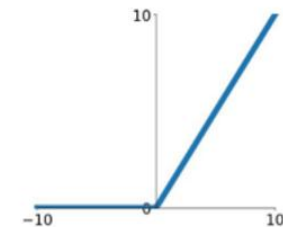
tanh

$$\tanh(x)$$



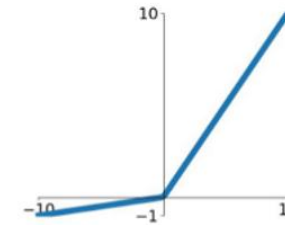
ReLU

$$\max(0, x)$$



Leaky ReLU

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

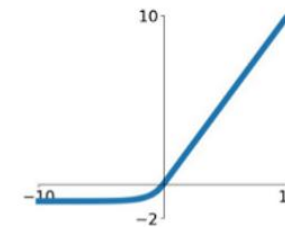


Maxout

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

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)

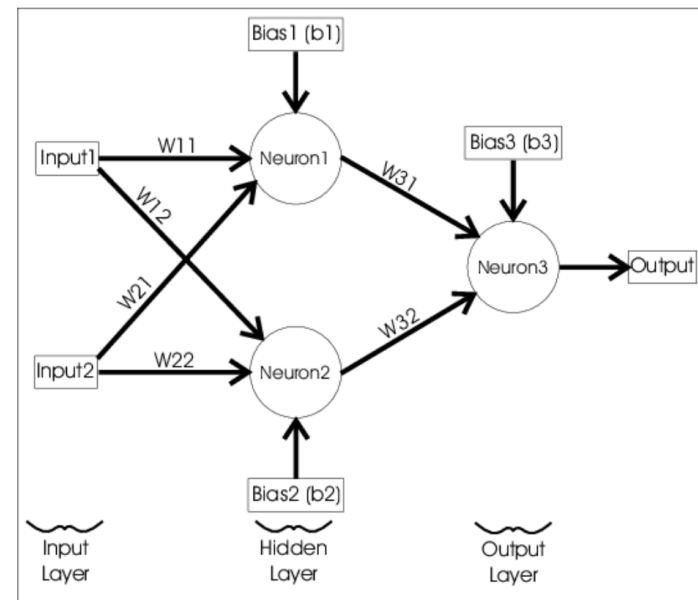
print("Predictions:\n", predictions)

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

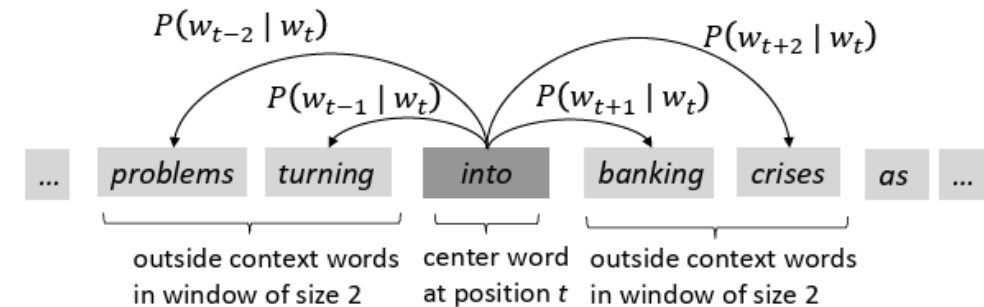
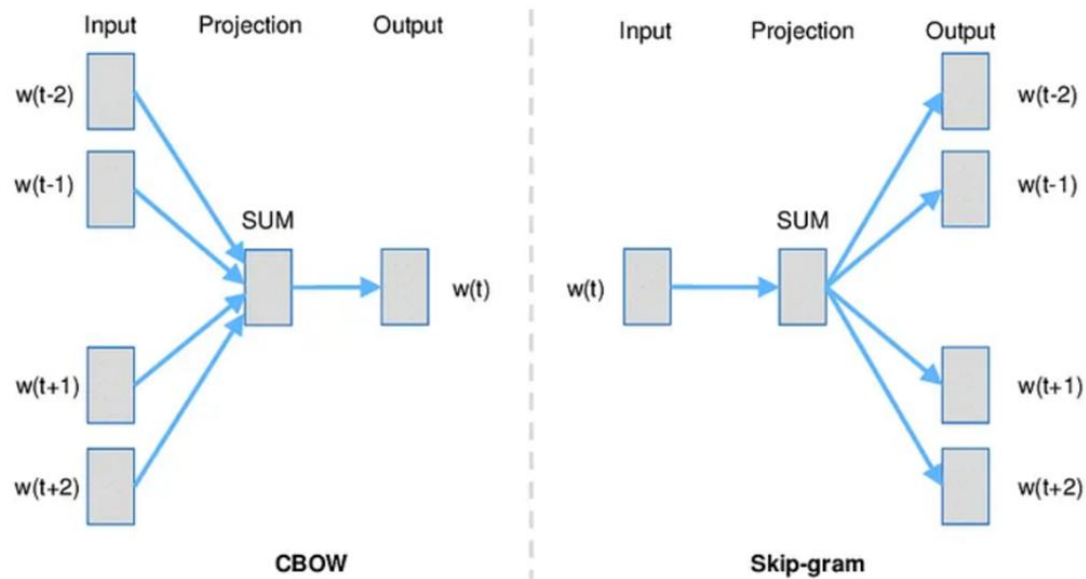
- **Continuous Bag of Words (CBOW):**
 - Predicts target word from context words
 - Faster training, better for frequent words
- **Skip-gram:**
 - Predicts context words from target word
 - Better for rare words, captures more semantic information

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

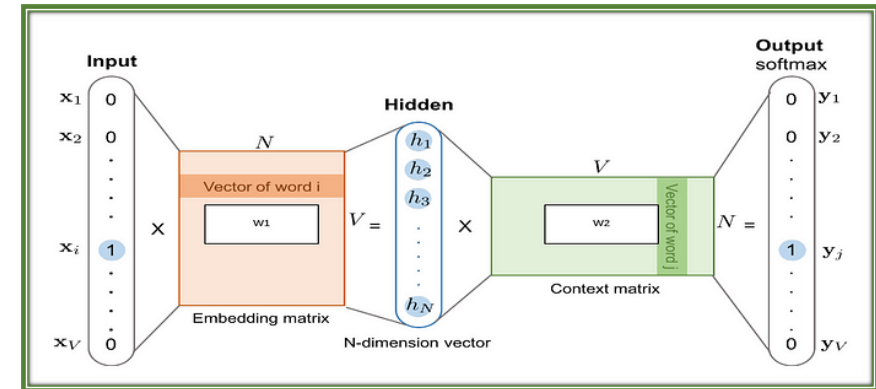


For each position $t=1, \dots, T$, predict context words within a window of fixed size m , given center word w_t .

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.



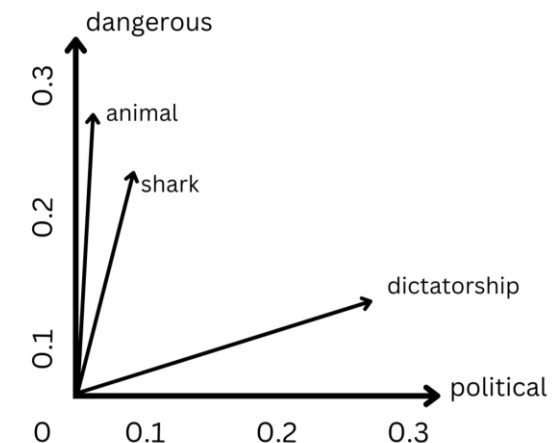
<https://python.plainenglish.io/understanding-word-embeddings-tf-idf-word2vec-glove-fasttext-996a59c1a8d3>

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



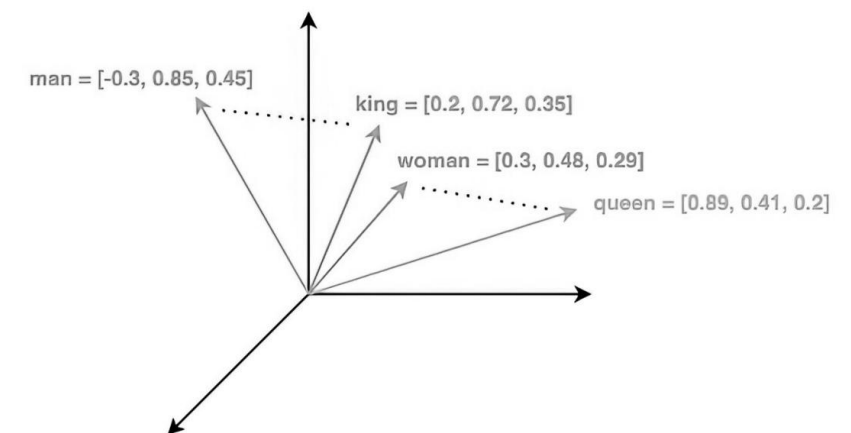
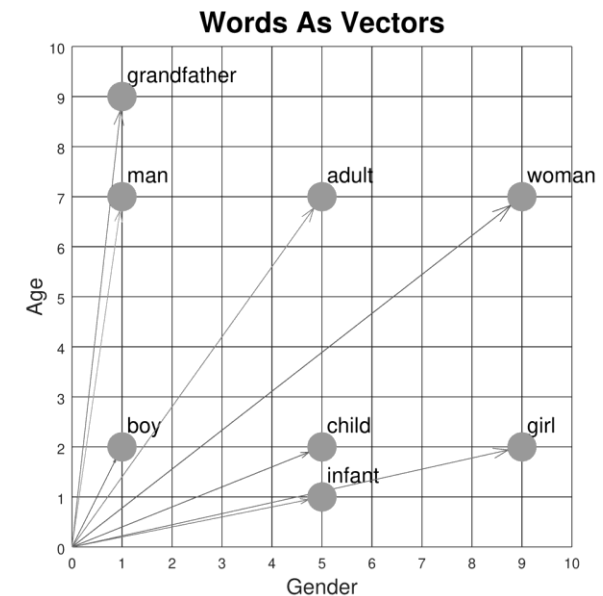
Vector Spaces and Word Embeddings

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**

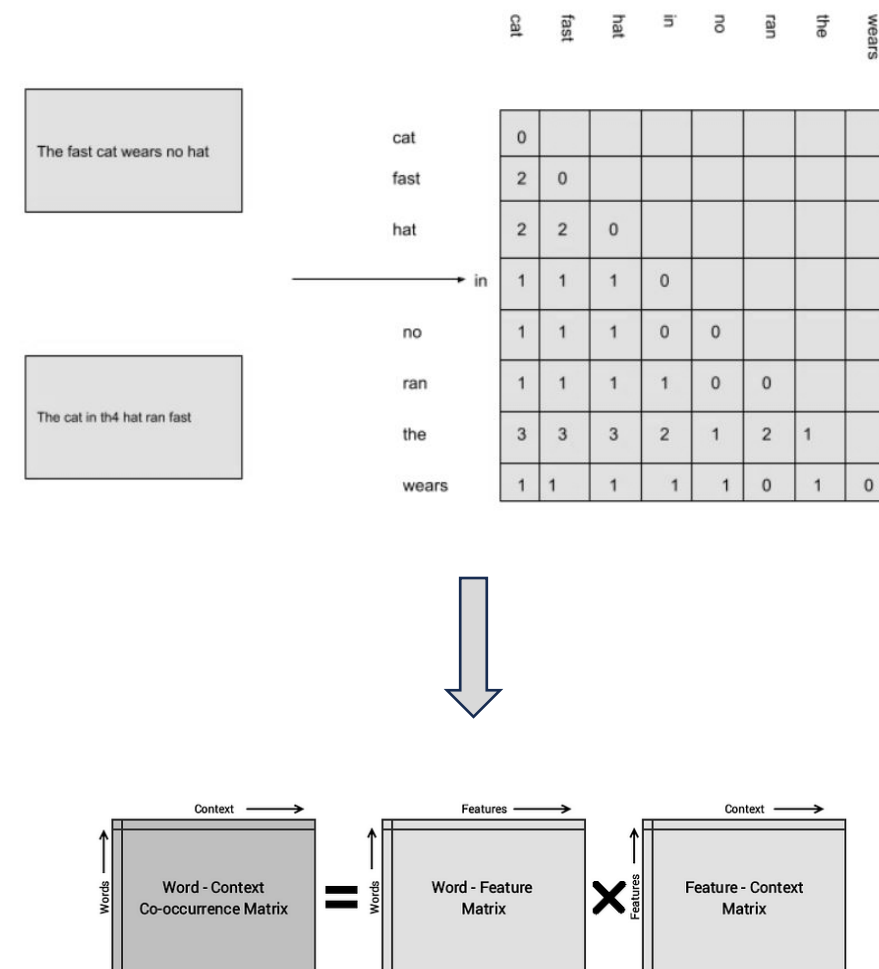
Bridges the gap between count-based methods and prediction-based methods by using global co-occurrence statistics to learn word vectors

- 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**



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

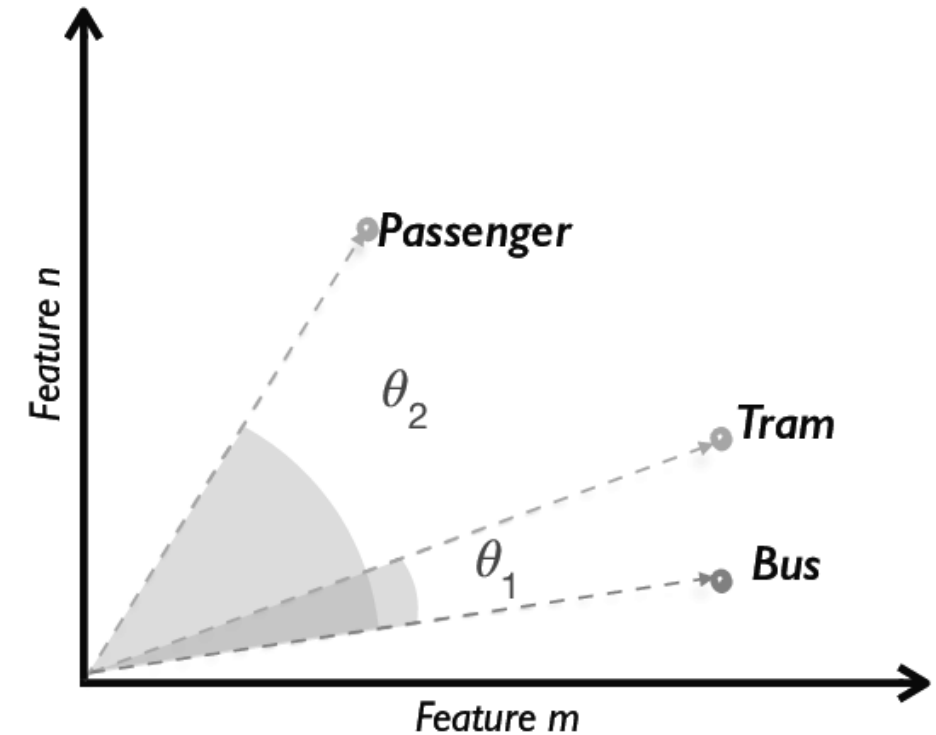
Most common metric used to compare word vectors:

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

- 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**

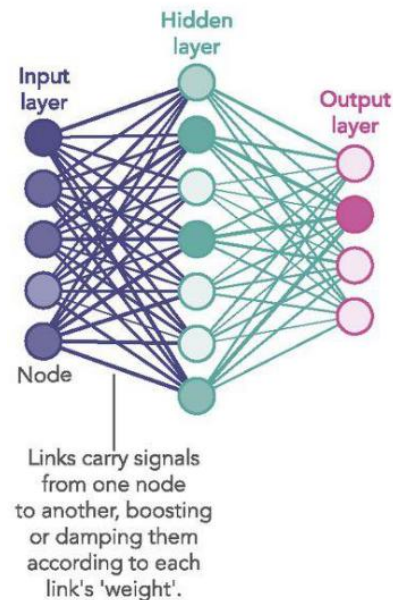
Enhance search engines by retrieving results based on semantic meaning, not just keyword matches.
Example: Searching for "heart attack" returns documents containing "cardiac arrest."

Deep Neural Networks

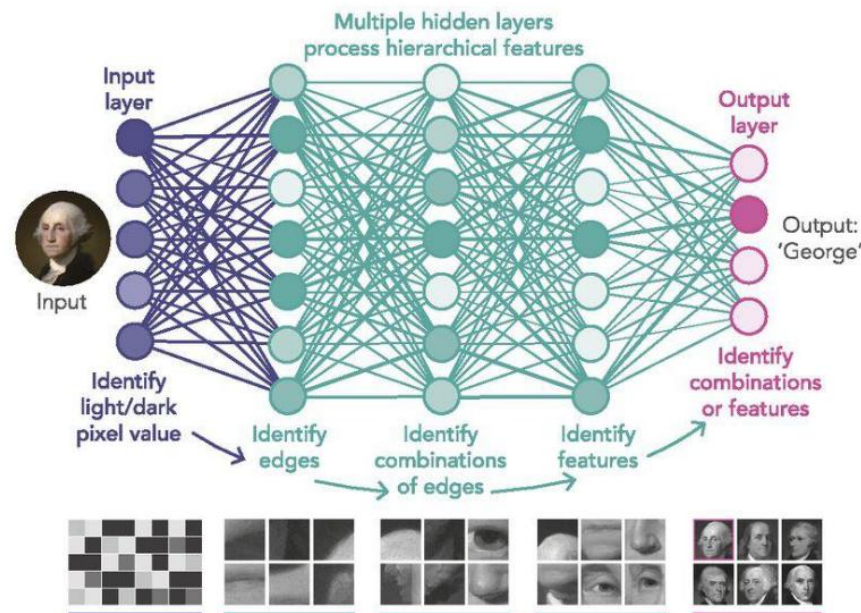
Deep Neural Networks

- A Machine Learning paradigm where raw input is fed to a deep learning algorithm which automatically decides what aspects (features) are important for task at hand.

1980S-ERA NEURAL NETWORK

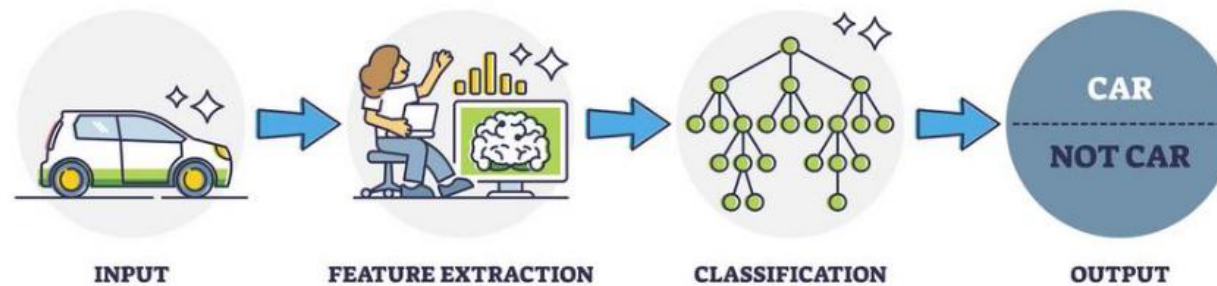


DEEP LEARNING NEURAL NETWORK

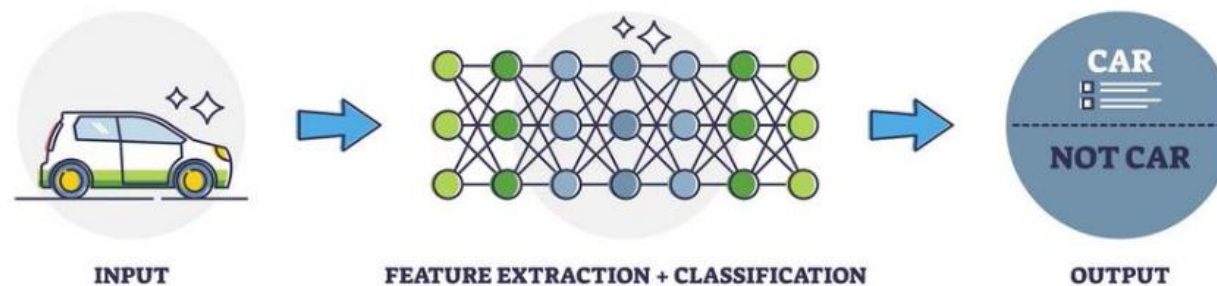


Deep Neural Networks

- A Machine Learning paradigm where raw input is fed to a deep learning algorithm which automatically decides what aspects (features) are important for task at hand.



DEEP LEARNING



Recurrent Neural Networks (RNNs)

What is an RNN?

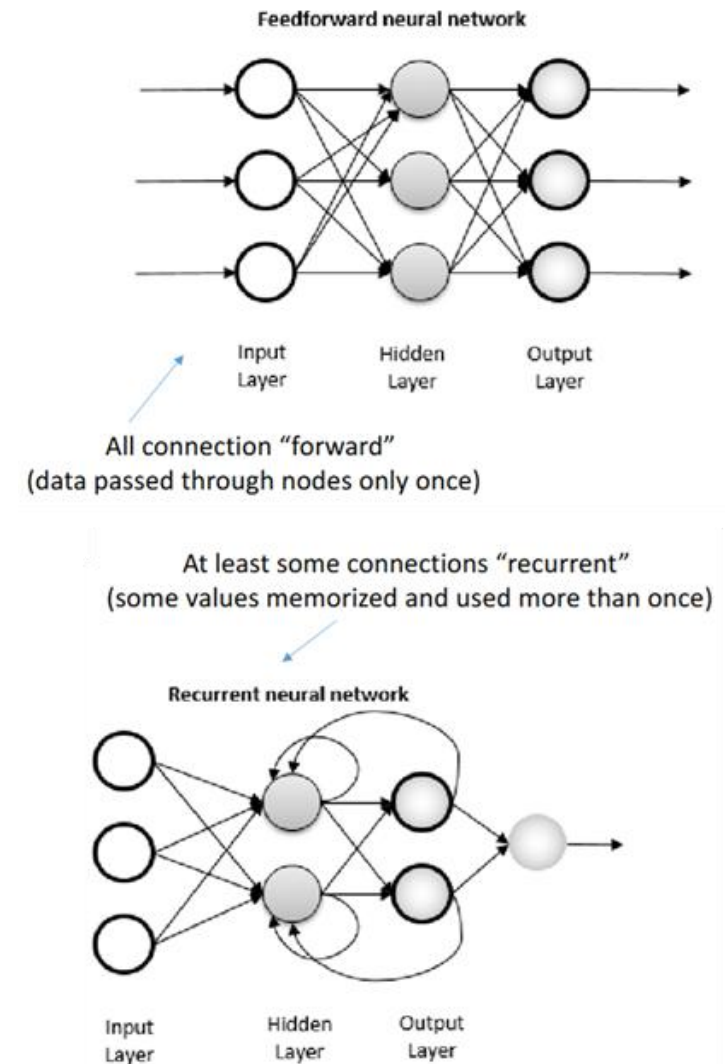
- A neural network designed to handle sequential data (e.g., text, time series)
- Maintains a hidden state to capture information from previous inputs

Key Concepts

- **Sequence Processing:** Inputs processed one timestep at a time
- **Hidden State:** Memory that carries information across steps
- **Weight Sharing:** Same weights used at each timestep
- **Backpropagation Through Time (BPTT):** Used for training over sequences

Variants

- **Vanilla RNN:** Basic form, suffers from vanishing gradients
- **LSTM (Long Short-Term Memory):** Handles long-term dependencies using gates
- **GRU (Gated Recurrent Unit):** Simplified LSTM with comparable performance



RNN Workflow and Applications

Workflow

1. Input sequence (e.g., words, sensor data)
2. For each timestep:
 - Compute new hidden state from input and previous state
3. Final output:
 - Classification (e.g., sentiment), sequence (e.g., translation), or prediction

Training

- Loss: Depends on task (e.g., cross-entropy for classification)
- Optimization: Backpropagation through time + SGD/Adam

Applications

- Natural Language Processing (e.g., sentiment analysis, translation)
- Speech recognition
- Time series forecasting
- Video and motion modeling

LSTM – Motivation and Architecture

What is an LSTM?

- A special type of **Recurrent Neural Network (RNN)**
- Designed to **capture long-term dependencies** in sequences
- Solves the **vanishing gradient problem** common in vanilla RNNs

Core Idea

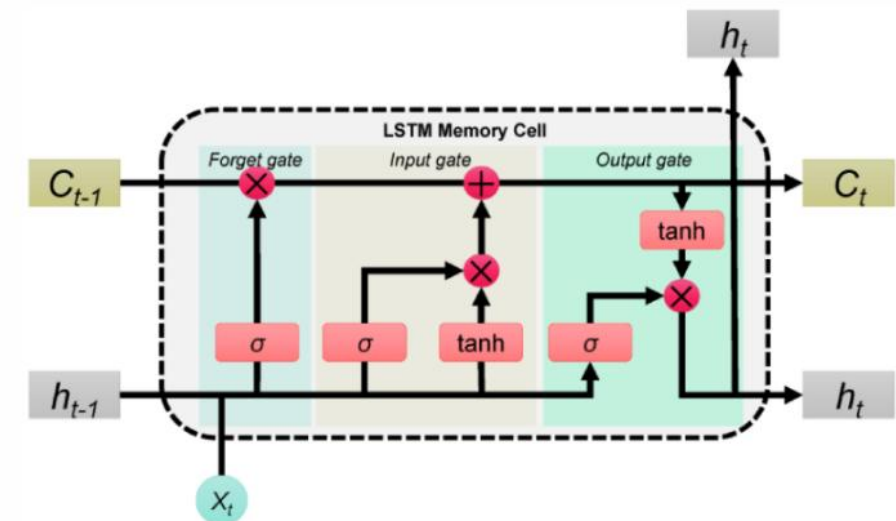
- Introduces a **memory cell** and **gating mechanisms** to control information flow
- Can **learn what to keep, forget, and output** at each timestep

LSTM Components

1. **Forget Gate:** Decides what to discard from the cell state
2. **Input Gate:** Decides what new information to store
3. **Cell State:** Long-term memory track
4. **Output Gate:** Controls what is output from the cell

Update Process

- Combines current input and previous hidden state to update the cell and hidden state
- Enables gradient flow across long sequences



Use Cases and Comparison

Why Use LSTM?

- Retains information over many timesteps
- Handles tasks where context from the distant past is important
- More stable training than vanilla RNNs

Applications

- Language modeling and generation
- Speech recognition
- Time series forecasting
- Video sequence analysis
- Sentiment classification

Comparison to GRU

- GRU: Simpler, fewer gates, faster to train
- LSTM: Slightly better at modeling long-range dependencies in some cases

Limitations

- Sequential processing slows training
- Still outperformed by attention-based models (e.g., Transformers) in most large-scale NLP tasks

Convolutional Neural Networks

What is a CNN?

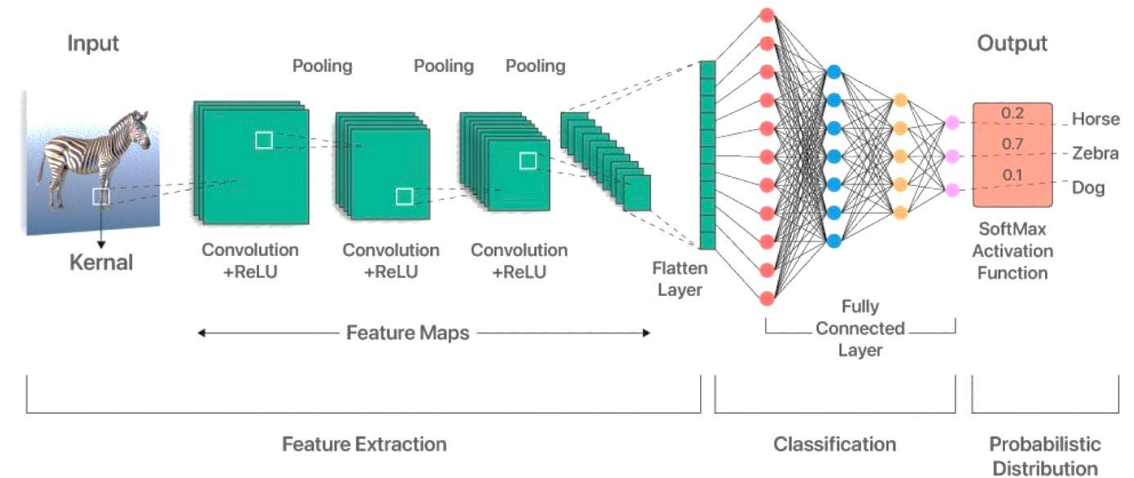
- A deep learning model for image recognition and classification
- Uses convolution to extract spatial features from input images

Key Components

- **Convolutional Layers:** Apply filters to detect local features (edges, textures)
- **Activation (ReLU):** Introduces non-linearity
- **Pooling Layers:** Downsample feature maps, reduce spatial dimensions
- **Fully Connected Layers:** Map features to output classes (e.g., Softmax)

Feature Hierarchy

- Early layers: general features (edges, gradients)
- Middle layers: patterns (corners, shapes)
- Deep layers: abstract features (object parts, semantics)



How CNNs Work and Applications

Workflow

1. Input image
2. Convolution + ReLU → Feature maps
3. Pooling → Downsampled maps
4. Flatten → Fully connected layers
5. Output class prediction

Training

- Loss: Cross-Entropy
- Optimization: Backpropagation with Adam/SGD

Applications

- Image classification (e.g., traffic signs, faces)
- Object detection and segmentation
- Medical imaging, autonomous vehicles

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?

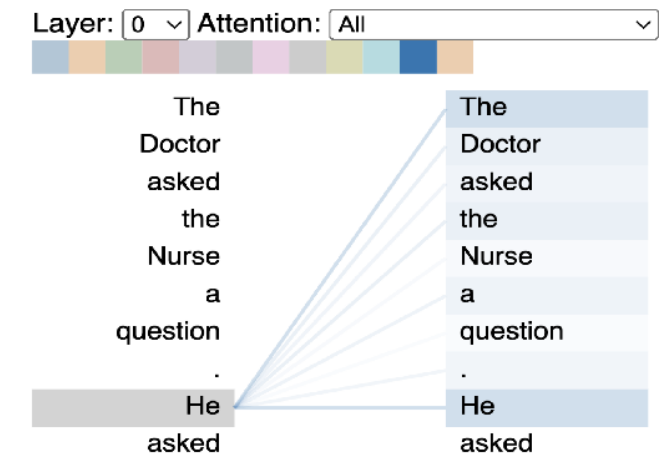
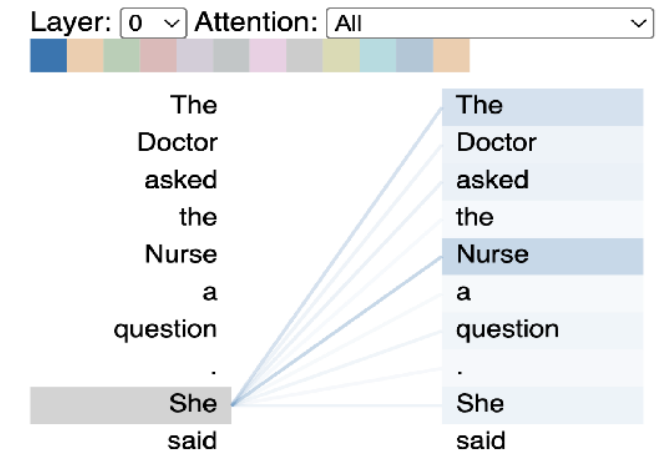
- Overcomes limitations of fixed-size context in RNNs
- Enables better handling of long sequences
- Improves performance in tasks like translation, summarization, and image captioning

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

Key Terms

- **Query (Q)**: The current focus point (e.g., decoder state)
- **Key (K)** and **Value (V)**: Representations of the input elements
- **Attention(Q, K, V)** = $\text{Softmax}(QK^T / \sqrt{d}) \times V$



Types of Attention

- **Additive Attention** (Bahdanau): Uses a feedforward network to compute scores
- **Dot-Product Attention** (Luong): Uses scaled dot products of Q and K
- **Self-Attention**: Query, key, and value all come from the same sequence (used in Transformers)

Self-Attention Highlights

- Computes relationships within a single sequence
- Enables parallel computation (unlike RNNs)
- Scales well to long sequences (used in BERT, GPT, etc.)

Applications

- Machine Translation (e.g., Transformer, BERT)
- Text Summarization
- Question Answering
- Vision (e.g., Vision Transformers)

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)

Core Components

- **Self-Attention**: Captures relationships between all words in a sequence
- **Multi-Head Attention**: Runs attention multiple times in parallel to capture diverse patterns
- **Positional Encoding**: Injects order information into the sequence (since there's no recurrence)
- **Feedforward Layers**: Apply non-linear transformations after attention
- **Residual Connections + Layer Normalization**: Stabilize and accelerate training

Transformer Block (Encoder/Decoder Structure)

- **Encoder**: Processes the input sequence
- **Decoder**: Generates the output sequence step by step using masked self-attention

Training and Applications

Workflow

1. Input sequence → Embedded with positional encodings
2. Passed through stacked encoder layers
3. Decoder attends to encoder outputs and previous tokens
4. Final output through linear + softmax layer

Advantages

- **Parallelizable:** No recurrence allows efficient training
- **Scalable:** Performs well on large datasets and long sequences
- **Modular:** Can be adapted to many tasks (e.g., BERT, GPT)

Applications

- Machine Translation (e.g., original Transformer model)
- Language Modeling (GPT series)
- Sentence Embeddings and Understanding (BERT, RoBERTa)
- Image and Vision Tasks (Vision Transformers, ViT)

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
- Achieves state-of-the-art results on many NLP benchmarks (e.g., SQuAD, GLUE)

Core Architecture

- Input = Tokens + Segment Embeddings + Positional Encodings
- Multiple stacked encoder layers
- Final output: contextualized embeddings for each token

Pretraining Objectives

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

Fine-tuning and Applications

Fine-Tuning

- Add a simple output layer on top of BERT
- Train on task-specific data for a few epochs
- Examples: classification head, QA span selector, token tagger

Strengths of BERT

- Deep bidirectional context
- Works well across many NLP tasks
- Can be fine-tuned with minimal architecture change

Variants of BERT

- **RoBERTa**: Removes NSP, trains longer
- **DistilBERT**: Lightweight version for speed
- **ALBERT**: Parameter-sharing to reduce size

Applications

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

ELMo – Overview and Architecture

What is ELMo?

- A deep, contextual word embedding model from 2018
- Developed by the Allen Institute for AI
- Generates embeddings that **depend on the entire context** of a word (not static like Word2Vec or GloVe)

Key Characteristics

- Based on a **bi-directional LSTM language model (biLM)**
- Trained to predict next words (left-to-right and right-to-left)
- Each word's embedding is a **function of the full sentence**

Architecture

- Character-level CNN for input representation
- Two-layer bidirectional LSTM for context modeling
- Output: contextualized embeddings for each word, at each layer

Embedding Usage

- Embeddings are extracted from one or more LSTM layers
- Combined (usually with learned weights) and used as input to downstream tasks

Impact and Applications

Why ELMo Was Important

- Introduced **deep contextualized word embeddings**
- Captures **polysemy**: different meanings of a word based on context
- Outperformed prior embeddings on a range of NLP tasks

Applications

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

Comparison to Later Models

- Pre-BERT: ELMo was state-of-the-art for many NLP tasks
- Unlike BERT, ELMo does not use Transformer architecture
- Less flexible for fine-tuning, but still powerful as a feature extractor

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

Key Innovation

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

Approach

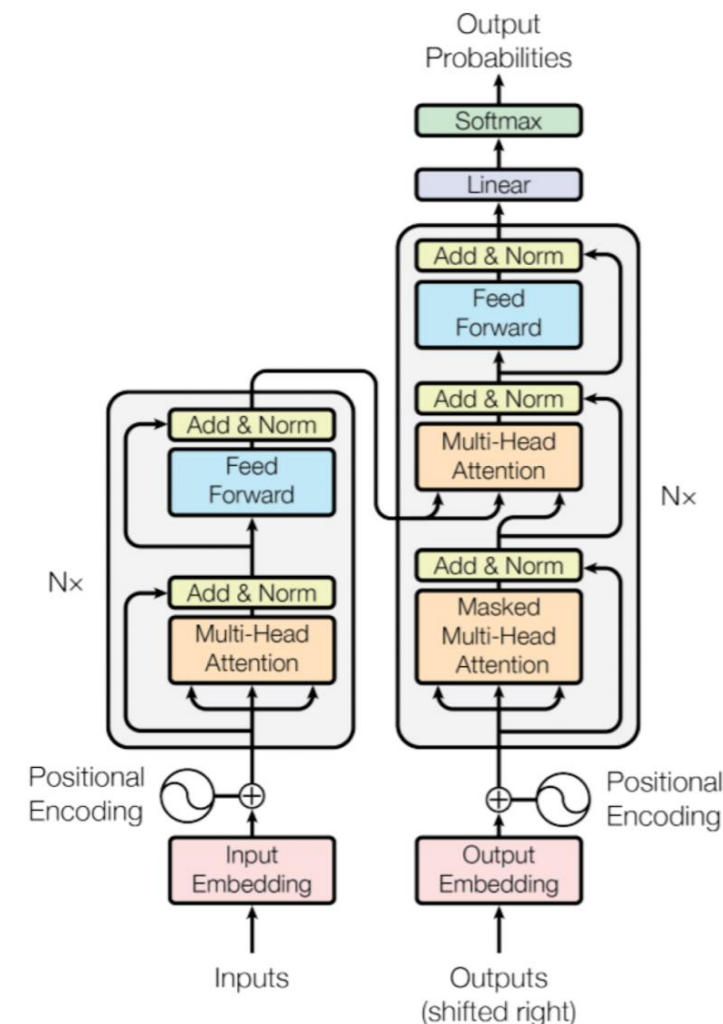
Uses deep, pre-trained neural networks (often transformer-based)
Embeddings are derived from entire sentences, capturing syntax and semantics dynamically

Examples

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

Characteristics

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



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

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
- Loss: Cross-entropy between predicted and actual token

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

```
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()
```


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:
 1. **Text generation:** Creating realistic text, such as articles, stories, or chatbot responses, for use in applications like content generation, language translation, and conversational interfaces.
 2. **Image and video generation:** Generating realistic images, videos, or 3D models for use in various industries, such as entertainment, advertising, and architecture.
 3. **Music and audio generation:** Generating music, sound effects, or voice synthesis for use in music production, audiobooks, or voice assistants.
 4. **Data augmentation:** Generating new data samples to augment existing datasets, which can help improve the performance of machine learning models.
 5. **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

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>

groq Play Ground

groqcloud

PlaygroundAPI KeysDashboardDocumentationUpgradePersonal

Playground

ChatStudio

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

Temperature1

Max Completion Tokens1024

Stream☒

JSON Mode☐

Advanced

Moderation: llamaguard☐

Top P1

Seed

Stop Sequence

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.

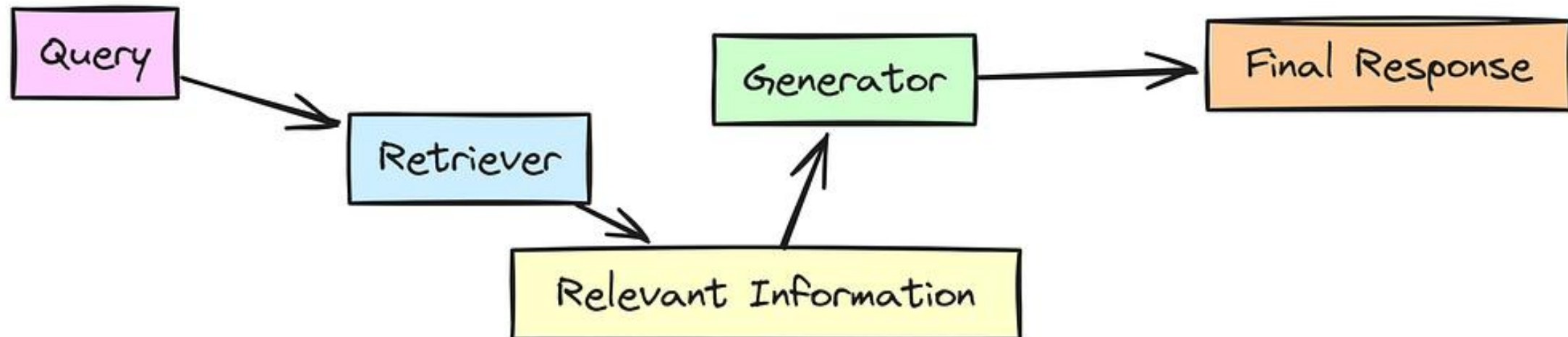
6. Print the response:

- `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.



References

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