BUSINESS INTELLIGENCE AND DATA ANALYTICS

University of Petra

Institute of Analytics
Recognized Programme

University of Petra

# 307307
# Part 2 – Introduction to Large Language Models

BUSINESS INTELLIGENCE AND DATA ANALYTICS
– WE PREDICT THE FUTURE –

Institute of Analytics
Recognized Programme

University of Petra

# Introduction to Neural Networks

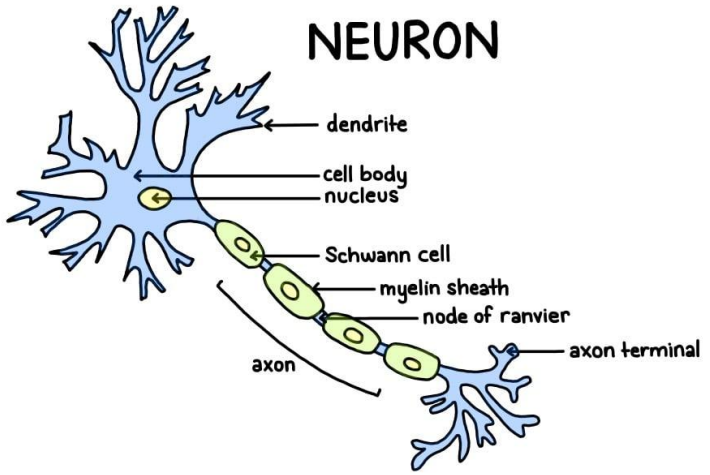
## The Perceptron

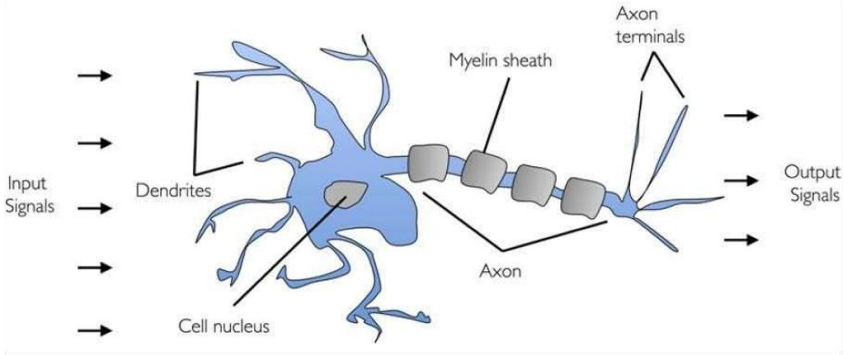
# Objectives

- Understand the biological inspiration for neural networks
- Learn about the perceptron model
- Implement a simple perceptron in Python
- Explore activation functions

# Biological Neuron

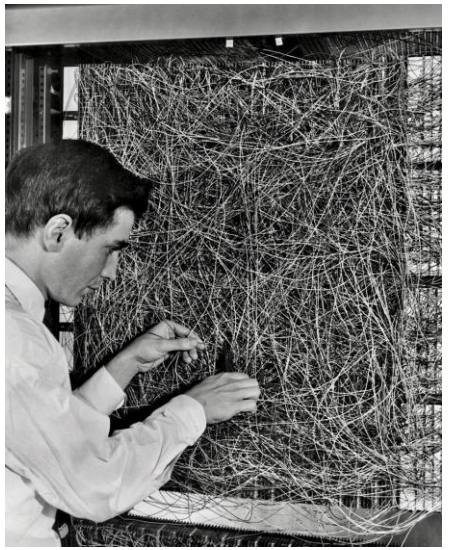
- Artificial neural networks are inspired by the human brain
- Neurons receive signals through dendrites
- When input exceeds a threshold, neuron "fires" through the axon
- Synapses connect neurons and modulate signal strength

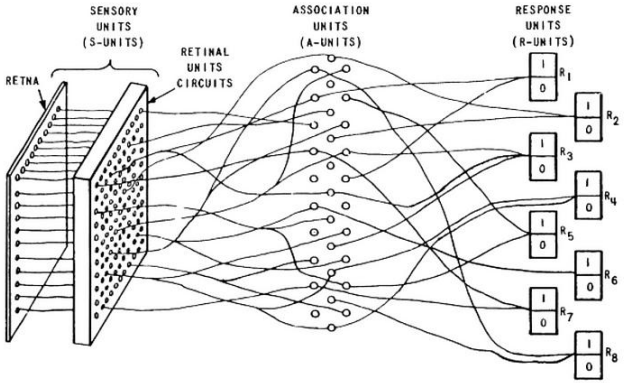


# The Perceptron: Building Block of Neural Networks

- Invented by Frank Rosenblatt in 1958
- Simplest form of a neural network
- Binary classifier: separates data into two categories
- Models a single neuron with multiple inputs and one output

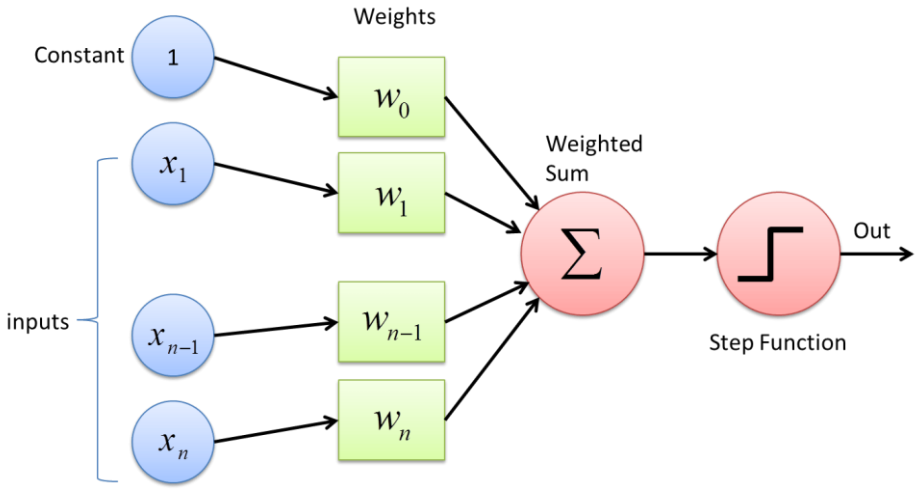





F. Rosenblatt

# The Perceptron

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

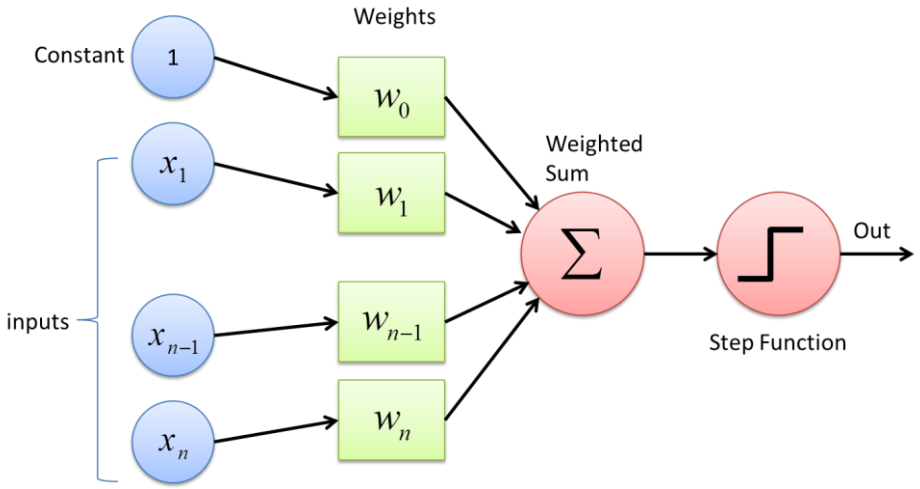


# How a Perceptron Works

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

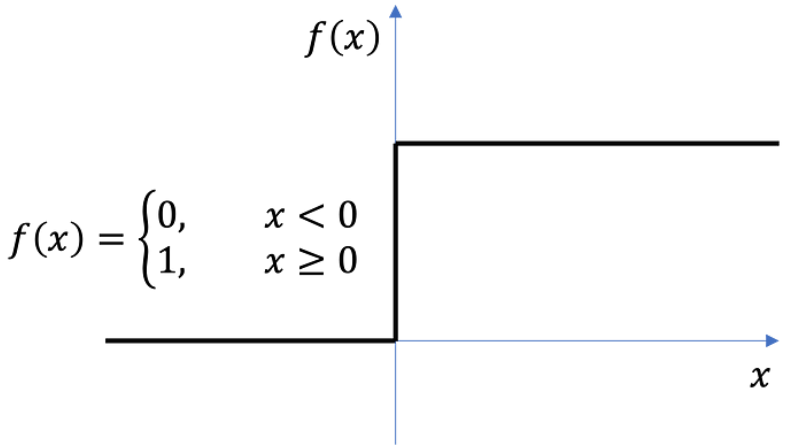
Mathematically:

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



# Perceptron Activation Function

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



# Perceptron Learning Rule

For each training example:

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

# Step-by-Step Hand Calculation for AND Gate

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

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

**First Iteration:**

**Example 1: (0,0) → 0**

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

# Step-by-Step Hand Calculation for AND Gate

**Example 2: (0,1) → 0**

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

**Example 3: (1,0) → 0**

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

# Step-by-Step Hand Calculation for AND Gate

**Example 4: (1,1) → 1**

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

**End of Iteration 1**:

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

# Second Iteration

**Example 1: (0,0) → 0**

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

**Example 2: (0,1) → 0**

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

# Python Implementation Perceptron from Scratch

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
Weights: [[2. 2.]]
Bias: [-3.]
Predictions: [0 0 0 1]
```

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

The code:

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

2.Sets up the training data for the AND gate

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

4.Trains the perceptron on the AND gate data

5.Prints the learned weights, bias, and predictions

The output shows:

- **Weights: [[2. 2.]]** - The perceptron learned to assign a weight of 2.0 to both inputs
- **Bias: [-3.]** - The bias is -3.0
- **Predictions: [0 0 0 1]** - The perceptron correctly classified all four examples of the AND gate

With these weights and bias, the decision function is: 2×(input1) + 2×(input2) - 3

For the four input combinations:

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

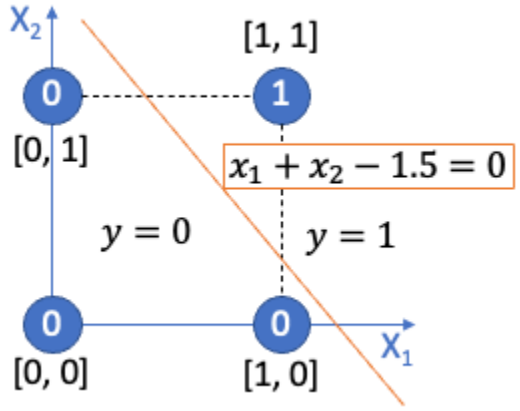
This perceptron implements the AND gate logic.

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

# Decision Boundary

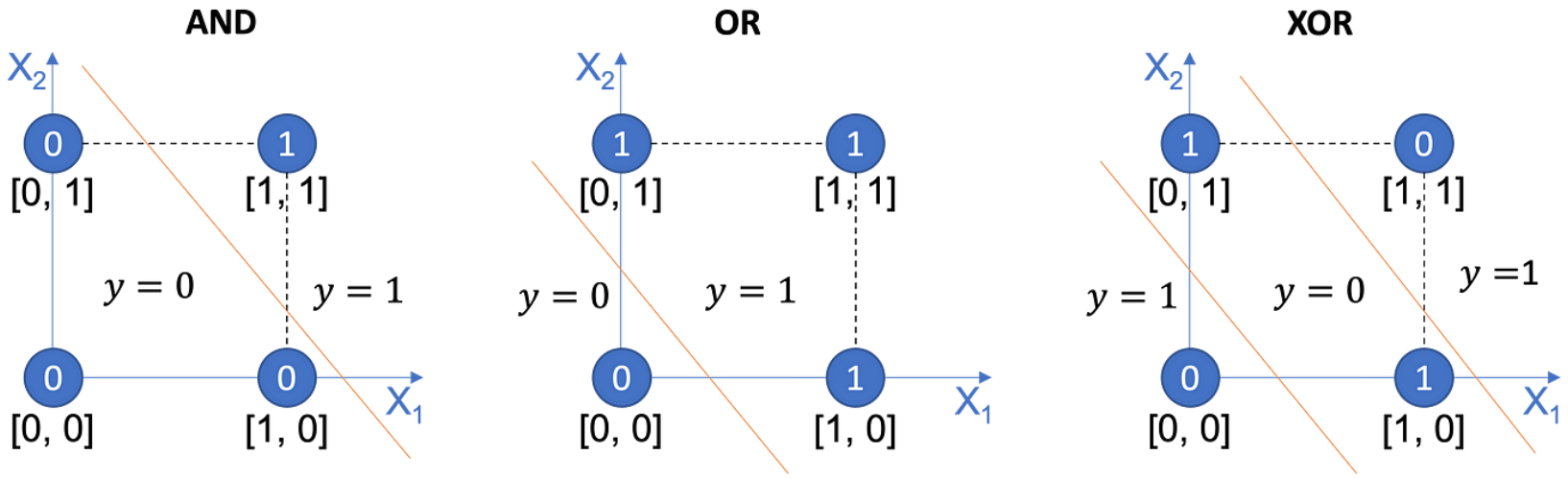
- The perceptron learns a decision boundary: $w_1x_1 + w_2x_2 + b = 0$
- Points above the line are classified as 1
- Points below the line are classified as 0
- For AND gate, only the point (1,1) should be above the line

| $X_1$ | $X_2$ | y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



# Limitations of Simple Perceptron

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



BUSINESS INTELLIGENCE AND DATA ANALYTICS
– WE PREDICT THE FUTURE –

Institute of Analytics
Recognized Programme

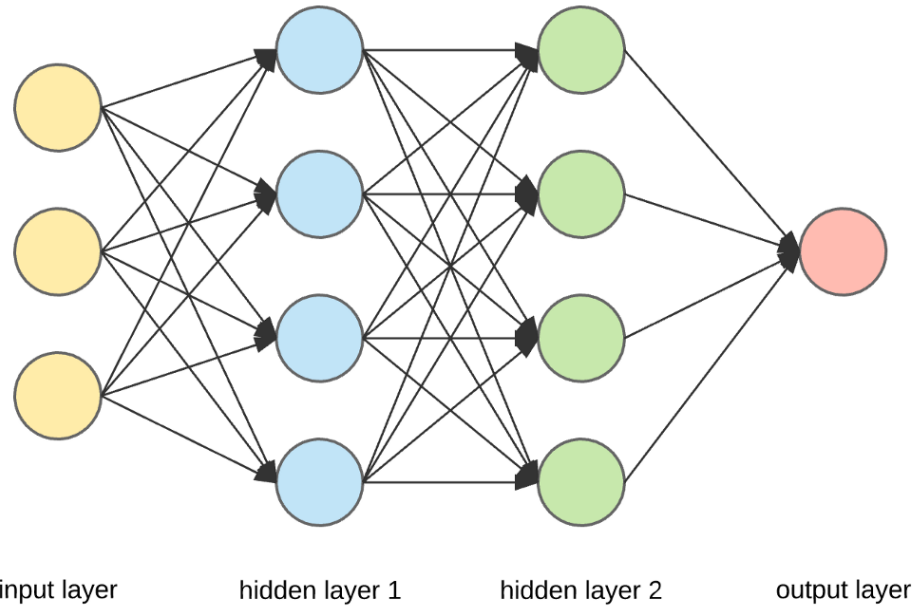
University of Petra

# Introduction to Neural Networks

## Multi-Layer Neural Network

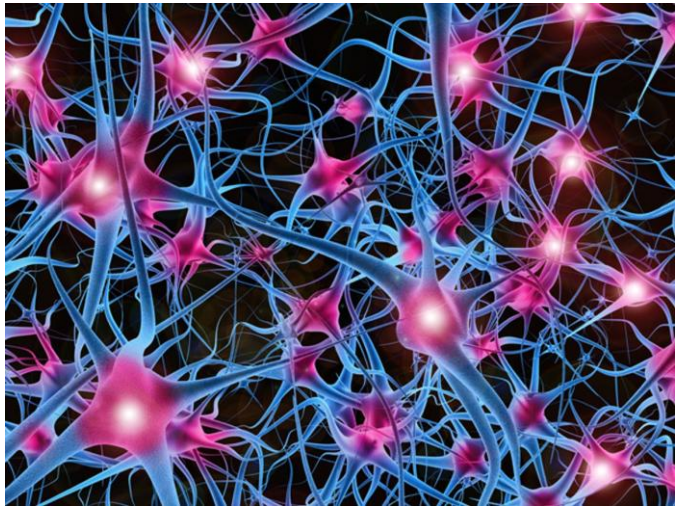
# What is a Neural Network?

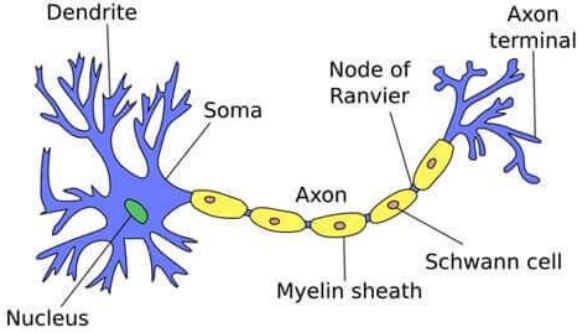
- A neural network is a computational model inspired by the human brain.
- It consists of layers of interconnected nodes (neurons) that process data.
- Used for pattern recognition, decision-making, and AI applications.

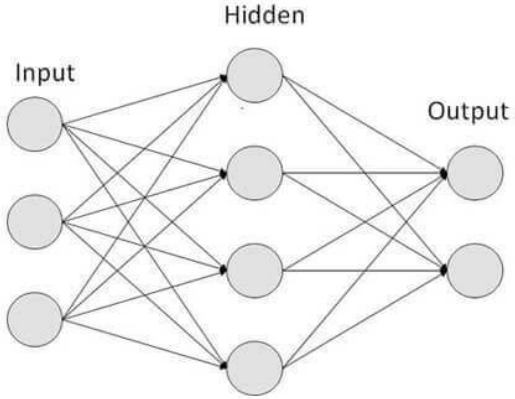


# Inspiration from the Human Brain

- Neural networks are modeled after **biological neurons** in the brain.
- **Neurons** receive inputs, process them, and pass the output to other neurons.
- Just like the brain, networks learn from experience (training).

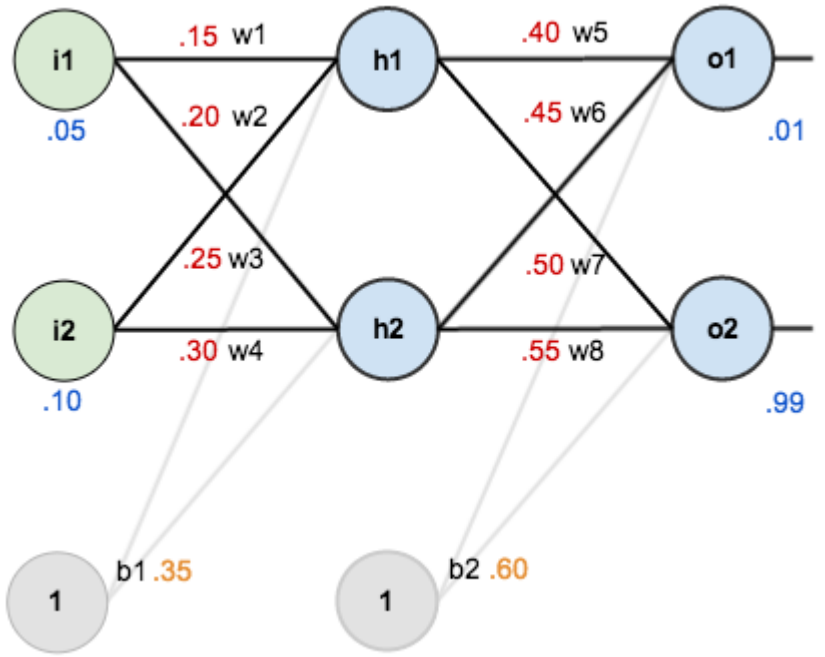




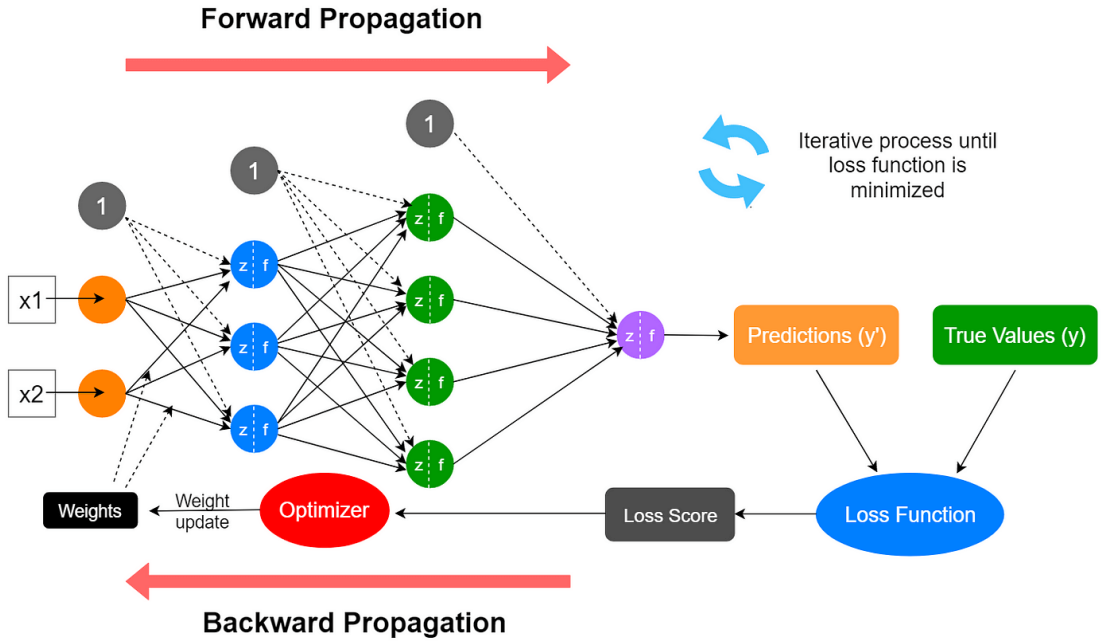
# Structure of a Neural Network

- Three main layers:
  - **Input Layer** – Receives raw data.
  - **Hidden Layers** – Process information using mathematical operations.
  - **Output Layer** – Produces the final result.
- Each connection has a **weight**, which determines importance.



# How Neural Networks Learn

- **Training Process:**
  - Feed data into the network.
  - Compute the output using weights.
  - Compare the output with the correct answer (loss calculation).
  - Adjust weights using **backpropagation** & **gradient descent** to improve accuracy.



# Types of Neural Networks

- **Feedforward Neural Network (FNN)** – Basic type; flows in one direction.
- **Convolutional Neural Network (CNN)** – Used for image recognition.
- **Recurrent Neural Network (RNN)** – Processes sequential data like speech.
- **Transformer Models** – Advanced networks for NLP (e.g., GPT, BERT).

https://towardsai.net/p/deep-learning/main-types-of-neural-networks-and-its-applications-tutorial-734480d7ec8e

# Applications of Neural Networks

- **Image Recognition** (e.g., Face Detection, Object Recognition)
- **Natural Language Processing** (e.g., Chatbots, Translation)
- **Speech Recognition** (e.g., Siri, Google Assistant)
- **Medical Diagnosis** (e.g., Detecting diseases from X-rays)

# Challenges & Limitations

- **High computational cost** – Needs powerful hardware (GPUs, TPUs).
- **Large datasets required** – More data = better accuracy.
- **Black-box nature** – Difficult to interpret decisions.
- **Overfitting** – Model may memorize instead of generalizing.

# Future of Neural Networks

- More efficient architectures (e.g., **Transformer models**).
- **Quantum AI** – Using quantum computing for better performance.
- **Smaller & faster models** for edge devices (e.g., mobile AI).
- **Ethical AI** – Addressing bias, fairness, and transparency.

# How Does Neural Networks Work?

# Neural Networks Intuition

- Suppose we have this data, and we want to build a model that establishes the relationship between x and y e.g. y = a * x
- One can clearly make out that y = 4 * x.
- But how can a neural network derive the relationship between y and x?

| x | y |
|---|---|
| 0 | 0 |
| 1 | 4 |
| 2 | 8 |
| 3 | 12 |
| 4 | 16 |

# How can a neural network find the value of a?

**Step1:** Random Guess

- We need to start somewhere, we can randomly guess the desired multiplier is 2.
- The relationship will now look like y = 2 * x

**Step 2:** Forward Propagation

- Here we are propagating the calculation forward from the input —> output.
- So now the table would look like this:

| X | Predicted Y’ |
|---|---|
| 0 | 2 * x = 0 |
| 1 | 2 * x = 2 |
| 2 | 2 * x = 4 |
| 3 | 2 * x = 6 |
| 4 | 2 * x = 8 |

# Calculate the Error

**Step 3**: Error

- Let us compare the prediction versus the actual.
- We calculate the error as the square of deviation of the actual from the predicted y'. This way we penalize the error irrespective of the direction of error.

| x | Actual y | Predicted y' | Squared error$(y-y')^2$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 4 | 2 | 4 |
| 2 | 8 | 4 | 16 |
| 3 | 12 | 6 | 36 |
| 4 | 16 | 8 | 64 |
| TOTAL | | | 120 |

# Minimize Error

**Step 4:** Minimize Error

- The next logical step is to minimize the error, we randomly initialized 2 as the multiplier.
- Based on that, the error was 120, we can try to deal with many other values ranging from -100 to 100.
- Since we know the real value (4), we will end up with the correct value quickly.
- But, in practice, when you deal with a lot more variables (here there is only one: x) and with more complex relationships, trying out different random values for x is not scalable, and not an efficient use of computing resources.

# The Derivates Method
# إيجاد الحل باستخدام الاشتقاق أو معدل تغير الاقتران عند قيمة (x) معينة

- A popular way to solve this problem is with the help of differentiation or derivatives.
- Basically, derivatives are the rate of change of a function with respect to one of its variables.
- In our example, it is the rate of change in the error or loss function with respect to x.
- To observe the effect of change, let's change the multiplier or weight or the parameter from 2 to 2.0001 i.e. increase it by 0.0001 and then decrease it by the same amount.
- In practice, this increase is even smaller, let us observe the impact:

| x | Actual y | Weight = 2 | Squared Error @ 2 | Weight = 2.0001 | Squared Error @ 2.0001 | Weight = 1.9999 | Squared Error @ 1.9999 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 4 | 2 | 2 | 2.0001 | 3.99 | 1.9999 | 4.0004 |
| 2 | 8 | 4 | 16 | 4.0002 | 15.99 | 3.9998 | 16.0016 |
| 3 | 12 | 6 | 36 | 6.0003 | 35.99 | 5.9997 | 36.0036 |
| 4 | 16 | 8 | 64 | 8.0004 | 63.99 | 7.9996 | 64.006 |
| 4 | 16 | 8 | 64 | 8.0004 | 63.99 | 7.9996 | 64.006 |
| TOTAL | | | **120** | | **119.988** | | **120.012** |

# The Derivates Method

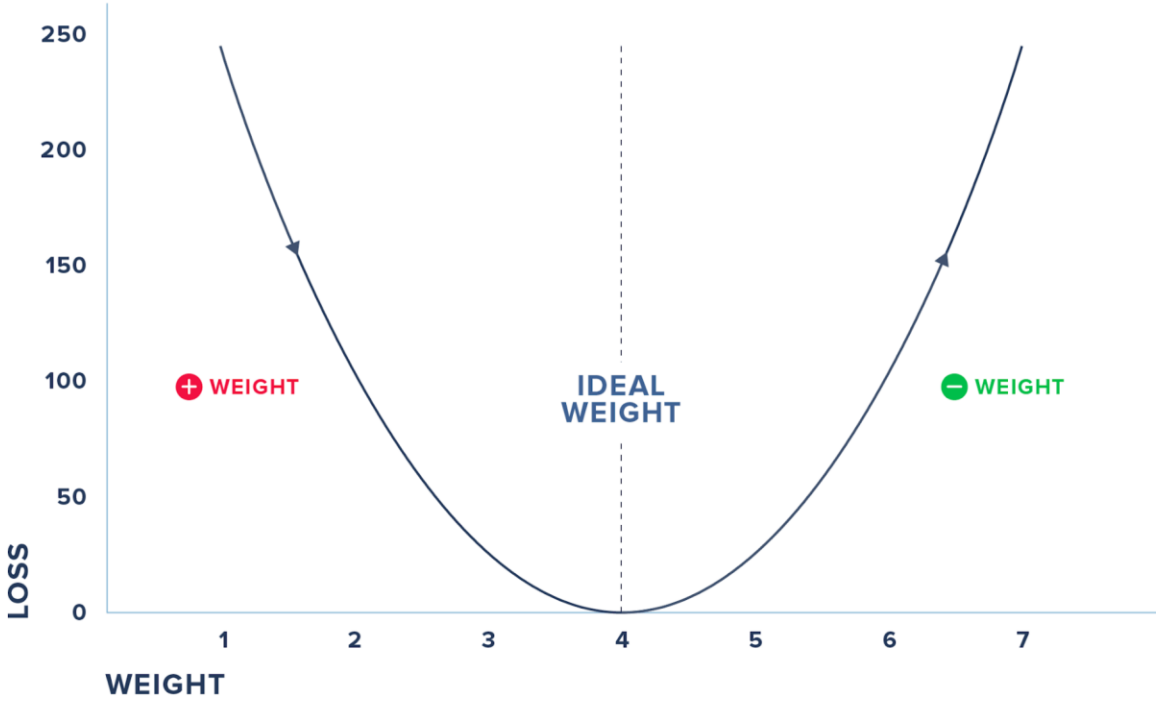
- With the increase in weight by 0.0001, the loss (the total error) decreased by 0.012. Also, when we decreased the weight by 0.0001, the loss increased by 0.012.
- For a 0.0001 increase in weight, the loss decreased by 120 times (0.012 / 0.0001).
- So how much should we increase our weight to reduce the loss?
- If we take a different arbitrary number such as 7 for the weight value, the loss increases to 270. So, we know now that the weight should be somewhere between 2 and 7.

Findings:

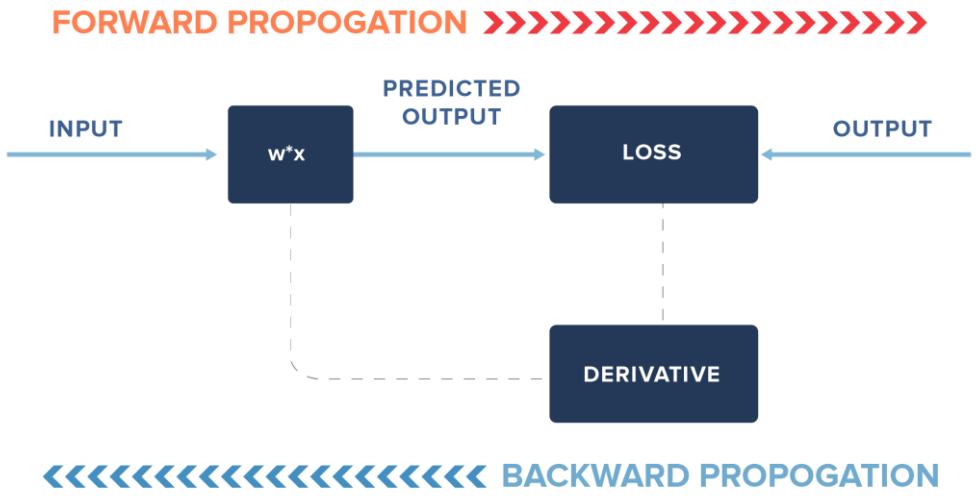
- If we decrease the weight from 2, the loss increases.
- If we increase the weight from 2, the loss decreases up to a point and then it increases as loss is high at weight = 7

# The Derivates Method

- We cannot do this iterative process of randomly checking for various weights and arriving at the magical parameter.
- Moreover, in practice, you will have to deal with many input variables and not just one.
- Hence, we can use derivatives where we check the effect on loss for a small change in the input.
- The figure summarizes the steps where we check for the derivative and increase the weights, if the loss decreases and vice-versa before we hit the weight where the loss reaches 0.
- We do this iteratively and update the weights and the loss till we minimize the loss to 0.
- The process by which we calculate the derivatives needed in the calculation of weights is called back propagation.



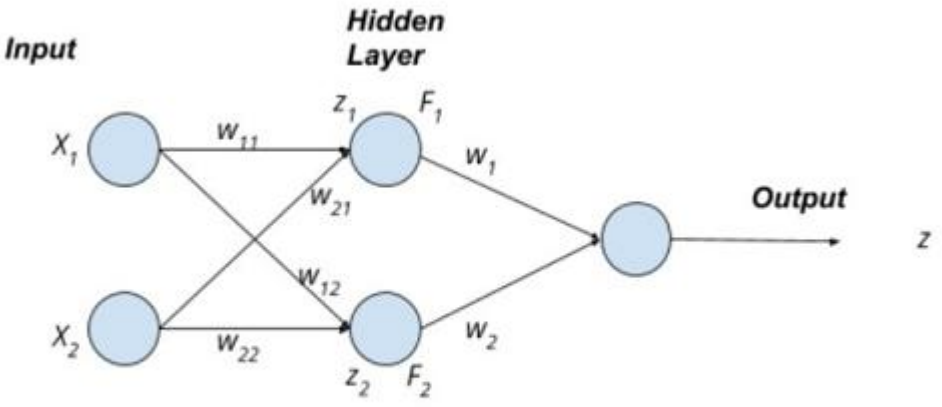
# The Derivates Method




- In practice, we may not update our weights fully but only by a fraction of the new learned change in weights (i.e. instead of 0.0001, we may update it by a fraction of 0.0001).
- Additionally, we may stop earlier than the loss reaches 0, say for example, we may stop if the loss reaches 0.000001.

# The Derivates Method

- In general, neural networks have more than one input variable and also they have hidden layers, but the general concept of computation is similar to what we have discussed.
- In the diagram shown on the right, we have 4 inputs or nodes in the input layer, 2 nodes in the hidden layer, and 1 node in the output layer.
- The lines indicate the connection each node has with the node in the layer following it.
- The connections are through the weights involved, which are learned via forward and back propagation.



Neural Network : Simplest Deep Learning Model with two Inputs, one Hidden Layer and one Output

*z1 = x1 * w11 + x2 * w21*

*z2 = x1 * w12 + x2 * w22*

- Here, “w” refers to weights connecting input layer to hidden layer. Likewise, we have a set of weights from hidden layer to output layer. The earlier case had just one of the input nodes and one output node with the connection.
- In case we are tasked with classifying items such as digits from 0 to 9 (multi-class output), we will have 10 nodes (each digit represented by one node) in the output layer instead of one. Likewise, we can have multiple hidden layers too.

# Code Examples

# Example 1: A Simple "Neuron" in Python

```python
import numpy as np


# Inputs (features)
inputs = np.array([1.0, 2.0, 3.0])  # Example input data
# Weights (coefficients)
weights = np.array([0.5, -0.2, 0.3])  # Randomly chosen weights
# Bias (extra term)
bias = 0.7
# Compute output (weighted sum + bias)
output = np.dot(inputs, weights) + bias
print("Output of the neuron:", output)
```

This code demonstrates the computation implemented by a single neuron.
The neuron computes the expression: 1.0*0.5 + 2.0*(-0.2) + 3.0*0.3 + 0.7
It uses the dot product command to perform matrix multiplication.

# Example 2: The Perceptron
# Similar to the previous example only with a step function

```
import numpy as np
# Step activation function (Binary output: 0 or 1)
def step_function(x):
    return 1 if x > 0 else 0
# Inputs and weights
inputs = np.array([1.0, 2.0, 3.0])
weights = np.array([0.5, -0.2, 0.3])
bias = 0.7
# Compute output
output = np.dot(inputs, weights) + bias
perceptron_output = step_function(output)
print("Perceptron output:", perceptron_output)
```

- A perceptron typically includes a step activation function (e.g., y = 1 if output > 0 else 0).
- Without activation, it's just a linear function (weighted sum).
- The step_function ensures binary classification (0 or 1).
- If output is positive, perceptron fires (1); otherwise, it doesn't (0).

# A Simple Neural Network (No Libraries)

```
import numpy as np

# Sigmoid activation function

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

# Inputs

inputs = np.array([0.5, 0.3])  # Two inputs


# Weights for hidden layer (2 inputs → 2 neurons)

weights_hidden = np.array([[0.4, 0.7], [-0.2, 0.5]])


# Bias for hidden layer

bias_hidden = np.array([0.1, 0.2])

# Compute hidden layer output

hidden_layer_input = np.dot(inputs, weights_hidden) + bias_hidden

hidden_layer_output = sigmoid(hidden_layer_input)

print("Hidden layer output:", hidden_layer_output)
```

Explanation:

- Uses sigmoid activation to keep values between 0 and 1.
- Basic feedforward step for two neuron

# Building a Neural Network using TensorFlow/Keras

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
# Generate dummy data (X: features, y: labels)
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  # Inputs (XOR gate)
y = np.array([[0], [1], [1], [0]])  # Expected output (XOR results)
# Create a simple neural network
model = keras.Sequential([
    keras.layers.Dense(4, activation='relu'),  # Hidden layer (4 neurons)
    keras.layers.Dense(1, activation='sigmoid')  # Output layer (1 neuron)
])
# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy',
   metrics=['accuracy'])
# Train the model
model.fit(X, y, epochs=100, verbose=0)  # Train for 100 epochs silently
# Make a prediction
prediction = model.predict(X)
print("Predictions:", prediction)
```

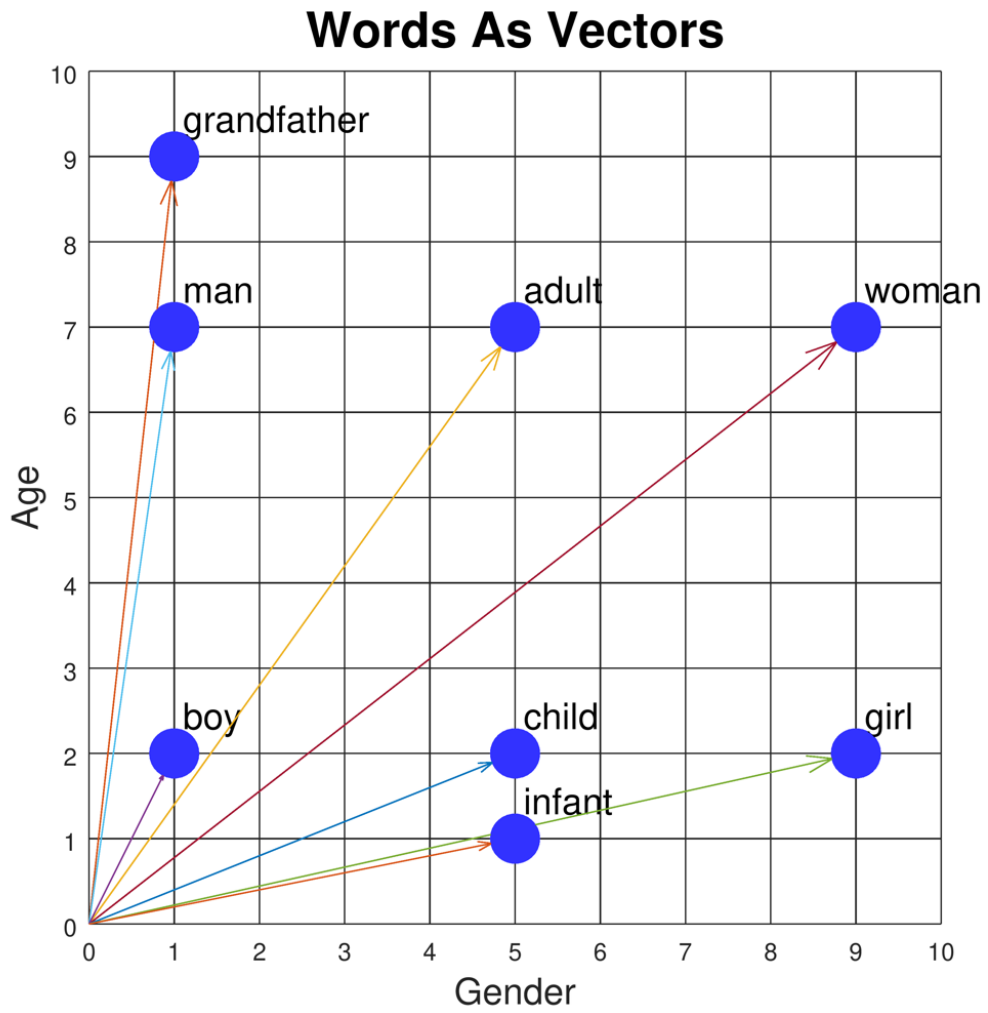
Explanation:

- Uses Keras to create a simple XOR classifier.
- ReLU for hidden layer, sigmoid for binary classification.
- Trains for 100 epochs to find best weights.

# Introduction to Word Embeddings

# Introduction to Word Embeddings

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



# The Evolution of Word Representations

**Traditional Methods (Pre-2013)**

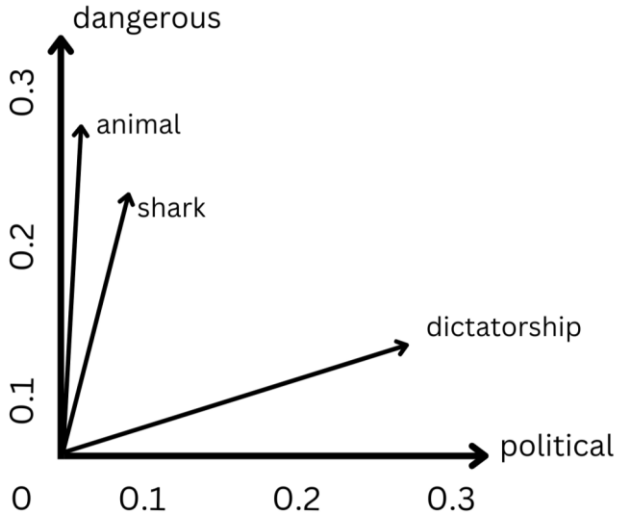
- **One-hot encoding**: Sparse, binary vectors (dimension = vocabulary size)
- **Limitations**:
  - No notion of similarity between words
  - Curse of dimensionality
  - No semantic information captured

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

# The Evolution of Word Representations

- **Problem**: How do we represent meaning mathematically?
- **Solution**: Distributional hypothesis - "You shall know a word by the company it keeps" (J.R. Firth, 1957)

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




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

# Word Similarities

```
import numpy as np

from sklearn.metrics.pairwise import cosine_similarity

# Fake word vectors (3D for simplicity)

word_vectors = {

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

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

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

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

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

}

def similarity(w1, w2):

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


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

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

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

# Word2Vec (Mikolov et al., 2013)

**Key Innovation**

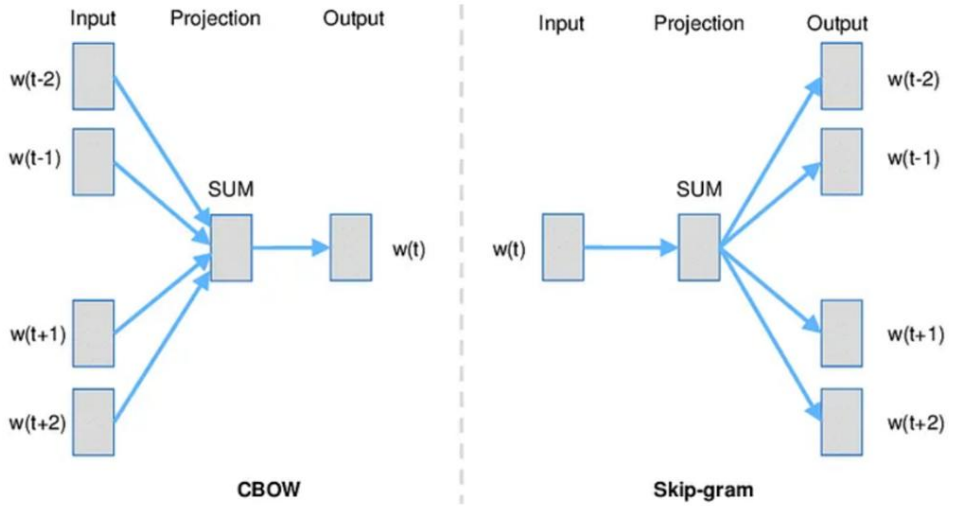
- Transformed NLP by creating dense vector representations through prediction-based models

**Two Architectures**

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

**Characteristics**

- Typically 100-300 dimensions (vs. vocabulary size)
- Linear relationships: king - man + woman ≈ queen
- Efficient training through negative sampling
- Limitations: Fixed vectors, one vector per word regardless of context



# Real Embeddings

- pip install spacy
- python -m spacy download en_core_web_md

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


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

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

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

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

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

# Practical Implementation: Word2Vec in Python

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

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

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

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

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

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

# Practical Implementation: GloVe in Python

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

# Practical Implementation: BERT in Python

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

# What is a Large Language Model?

A **Large Language Model (LLM)** is an advanced artificial intelligence system trained on vast amounts of text data to understand, generate, and manipulate human language. These models use **deep learning**, particularly **transformers**, to predict and generate text in a coherent and context-aware manner.

Key features:

- Trained on billions of words from books, articles, and the internet.
- Can perform tasks like translation, summarization, code generation, and answering questions.
- Examples include **GPT-4, ChatGPT, BERT, PaLM, LLaMA, and Claude**.
- LLMs power modern AI applications, including chatbots, virtual assistants, and content creation tools.

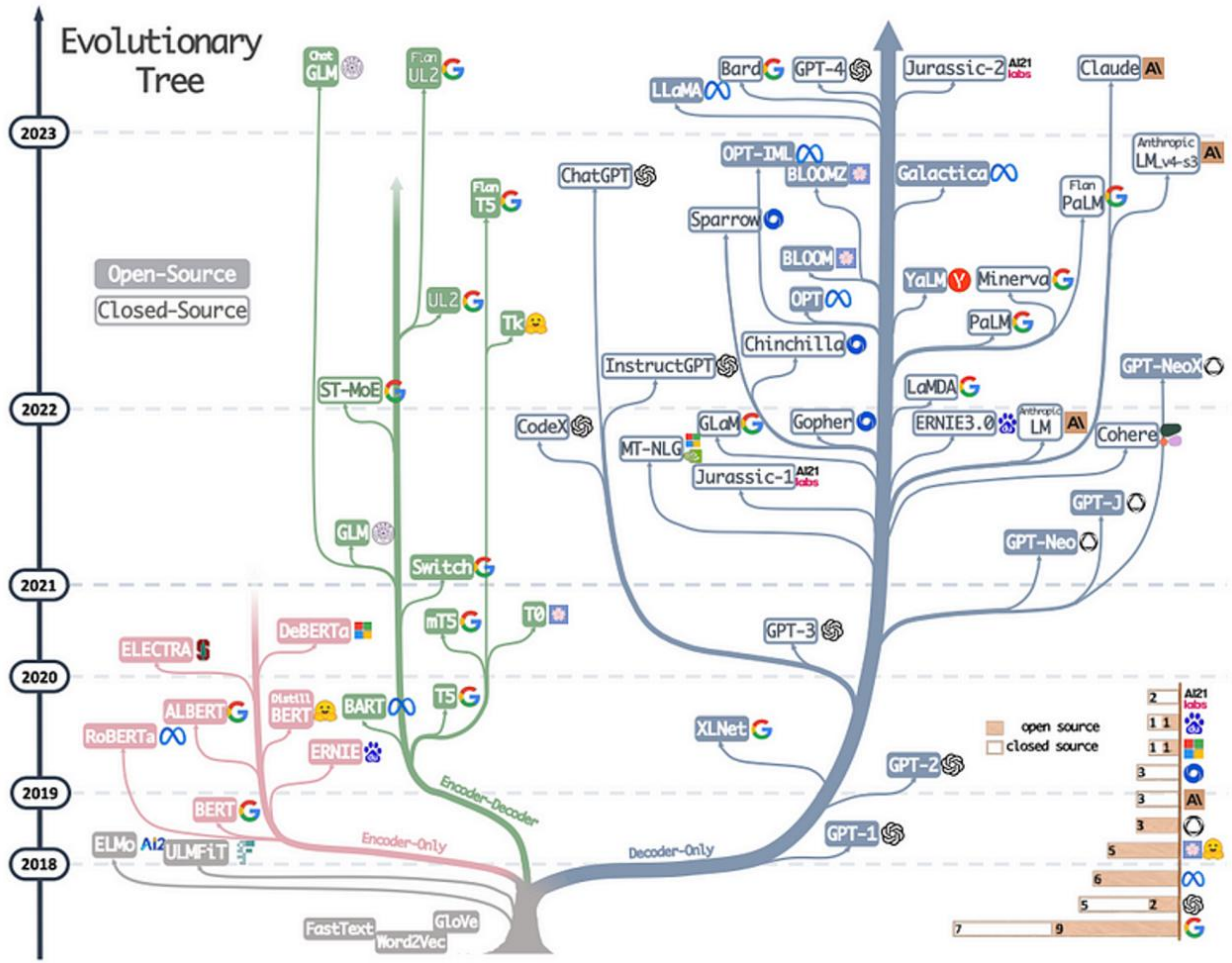
# Evolution of Large Language Models

The image is an **evolutionary tree of natural language processing (NLP) models**, showing their development from 2018 to 2023. It categorizes models into **open-source (light color) and closed-source (darker color)**, tracking major advancements in transformer-based architectures.

Key highlights:

- **Early models (2018-2019):** Word embeddings like **Word2Vec, FastText, GloVe**, evolving into transformer-based models like **BERT, RoBERTa, and XLNet**.
- **Growth in 2020-2021:** Expansion into encoder-only (e.g., **ELECTRA, ALBERT**), encoder-decoder (e.g., **T5, BART**), and decoder-only models (**GPT-2, GPT-3**).
- **2022-2023 advancements:** Introduction of **ChatGPT, GPT-4, Claude, PaLM, LLaMA, Bard, BLOOM, and more**, reflecting both **research-driven (open-source) and commercial (closed-source) efforts**.

The diagram illustrates the competition and innovation between **Google, OpenAI, Meta, AI21 Labs, and Anthropic**, among others.



# Introduction to HuggingFace

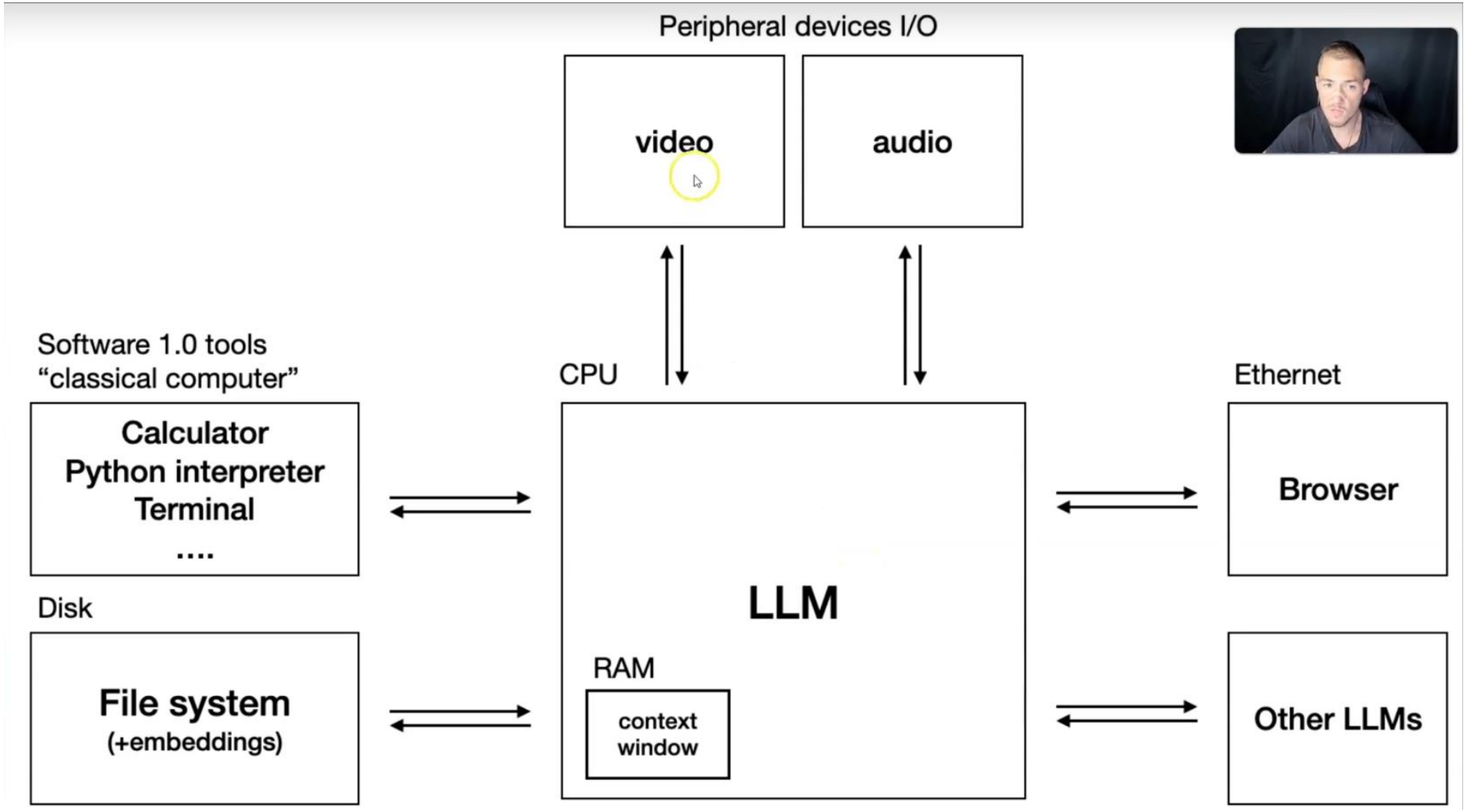
BUSINESS INTELLIGENCE AND DATA ANALYTICS
– WE PREDICT THE FUTURE –

Institute of Analytics
Recognized Programme

University of Petra

# Introduction to LLM Agents

# LLM Agents


Peripheral devices I/O
video
audio
Software 1.0 tools
“classical computer”
CPU
Ethernet
Calculator
Python interpreter
Terminal
....
Browser
Disk
LLM
RAM
File system
(+embeddings)
context
window
Other LLMs

# Core LLM Concepts

# Encoder vs. Decoder Architectures

- **Encoder Architecture**
  - Bidirectional context processing (sees full context)
  - Focused on representation learning
  - Ideal for understanding tasks: classification, NER, sentiment analysis
  - Examples: BERT, RoBERTa, DeBERTa
  - Output: contextual embeddings for each token
- **Decoder Architecture**
  - Unidirectional/autoregressive processing (sees only past context)
  - Focused on text generation
  - Ideal for generative tasks: text completion, summarization, translation
  - Examples: GPT models, LLaMA, Claude
  - Output: probability distribution for next token
- **Encoder-Decoder Architecture**
  - Combines both approaches
  - Encoder processes input, decoder generates output
  - Ideal for sequence-to-sequence tasks: translation, summarization
  - Examples: T5, BART, PaLM-E
  - Output: generated sequence based on encoded input

# Self-Attention Mechanisms

- **Key Innovation in Transformers**
  - Computes relationships between all tokens in a sequence
  - Allows modeling of long-range dependencies
  - Parallel computation (unlike sequential RNNs)
- **Self-Attention Computation**
  - For each token, create query (Q), key (K), and value (V) vectors
  - Compute attention scores: how much each token should attend to others
  - Apply softmax to get attention weights
  - Create weighted sum of value vectors
  - Formula: Attention(Q, K, V) = softmax(QK^T/√d_k)V
- **Multi-Head Attention**
  - Run multiple attention mechanisms in parallel
  - Each "head" can focus on different aspects of relationships
  - Outputs concatenated and projected to original dimension
  - Enables the model to jointly attend to information from different representation subspaces

# Positional Encoding

- **The Position Problem**
  - Self-attention is permutation-invariant (order insensitive)
  - Language inherently depends on word order
  - Need to inject position information
- **Sinusoidal Positional Encoding (Vaswani et al.)**
  - Fixed encoding using sine and cosine functions
  - Different frequencies for different dimensions
  - Formula:
    - PE(pos, 2i) = sin(pos/10000^(2i/d_model))
    - PE(pos, 2i+1) = cos(pos/10000^(2i/d_model))
  - Allows model to extrapolate to sequence lengths not seen during training
- **Learned Positional Encoding**
  - Position embeddings learned during training
  - Can capture more nuanced position relationships
  - Limited to maximum sequence length seen during training
  - Used in models like BERT, RoBERTa
- **Relative Positional Encoding**
  - Encode relative distances between tokens rather than absolute positions
  - Improves handling of longer sequences
  - Examples: T5, DeBERTa

# Tokenization Strategies

- **Purpose of Tokenization**
  - Convert raw text into discrete tokens for model processing
  - Manage vocabulary size and out-of-vocabulary words
  - Balance between granularity and context window efficiency
- **Word-Level Tokenization**
  - Each token is a complete word
  - Pros: Semantically meaningful units
  - Cons: Large vocabulary, many OOV words
  - Example: early Word2Vec implementations
- **Character-Level Tokenization**
  - Each token is a single character
  - Pros: Tiny vocabulary, no OOV issues
  - Cons: Loses word-level semantics, requires longer sequences
  - Example: Character-level CNN/RNN models
- **Subword Tokenization**
  - Balance between word and character levels
  - Common algorithms:
    - Byte-Pair Encoding (BPE): Iteratively merge most frequent character pairs
    - WordPiece: Similar to BPE but uses likelihood rather than frequency
    - Unigram Language Model: Probabilistic approach
    - SentencePiece: Language-agnostic, treats whitespace as a character
  - Examples: GPT models use BPE variants, BERT uses WordPiece

# Transfer Learning and Fine-tuning

- **Transfer Learning Paradigm**
  - Pre-train model on large general corpus
  - Adapt to downstream tasks with less data
  - Leverages general language knowledge for specific applications
- **Pre-training Approaches**
  - Self-supervised learning on unlabeled text
  - Masked Language Modeling (MLM) for encoders
  - Causal Language Modeling (CLM) for decoders
  - Seq2Seq objectives for encoder-decoder models
- **Fine-tuning Methods**
  - Full Fine-tuning: Update all model parameters
    - Most flexible but computationally expensive
    - Prone to catastrophic forgetting
  - Parameter-Efficient Fine-tuning (PEFT):
    - LoRA (Low-Rank Adaptation): Add trainable low-rank matrices
    - Adapter Layers: Insert small trainable modules
    - Prompt Tuning: Optimize continuous prompt embeddings
    - Prefix Tuning: Prepend trainable vectors to key and value
  - Benefits of PEFT:
    - Reduces computational requirements
    - Minimizes storage needs
    - Prevents catastrophic forgetting
    - Enables multi-task learning

# Additional Key Concepts

- **Attention Mask**
  - Controls which tokens can attend to which others
  - Essential for implementing causal attention in decoders
  - Handles variable sequence lengths and special tokens
- **Layer Normalization**
  - Normalizes activations across features
  - Stabilizes and accelerates training
  - Applied before or after attention and feed-forward blocks
- **Residual Connections**
  - Allow direct flow of information through layers
  - Help mitigate vanishing gradient problem
  - Enable training of very deep networks
- **Feed-Forward Networks**
  - Two-layer neural networks with ReLU/GELU activation
  - Applied to each position separately and identically
  - Often where most parameters in the model reside
  - Formula: $FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$
- **Activation Functions**
  - GELU (Gaussian Error Linear Unit): Smoother than ReLU
  - SwiGLU: Enhanced variant used in PaLM and newer models
  - Used in feed-forward networks between layers

# Model Architecture Components

- **Embedding Layer**
  - Converts token IDs to dense vectors
  - Typically 768-4096 dimensions in modern LLMs
  - Often tied with output layer weights
- **Transformer Blocks/Layers**
  - Basic building unit of transformer models
  - Each block contains:
    - Self-attention mechanism
    - Feed-forward network
    - Layer normalization
    - Residual connections
  - Modern LLMs contain dozens to hundreds of layers
- **Output Layer**
  - Projects final hidden states to vocabulary size
  - For decoders: produces next-token probabilities
  - For encoders: produces task-specific outputs after fine-tuning
- **Scaling Patterns**
  - Deeper models (more layers)
  - Wider models (larger hidden dimensions)
  - More attention heads
  - Trade-offs between depth, width, and compute efficiency

# Training Methodologies

# Pre-training Objectives

- **Masked Language Modeling (MLM)**
  - Used in encoder models like BERT
  - Random tokens are masked (typically 15%)
  - Model predicts the original tokens
  - Variants:
    - Whole Word Masking: Mask all subwords of a word
    - SpanBERT: Mask contiguous spans instead of random tokens
    - RoBERTa: Dynamic masking (new masks each epoch)
- **Causal Language Modeling (CLM)**
  - Used in decoder models like GPT
  - Predict next token given previous tokens
  - Maximizes likelihood: $P(x_n|x_1,...,x_{n-1})$
  - Training employs teacher forcing (use ground truth as context)
  - Loss function: cross-entropy over vocabulary
- **Denoising Objectives**
  - Text Infilling: Fill in arbitrary spans of text (T5, BART)
  - Sentence Permutation: Recover original order of shuffled sentences
  - Document Rotation: Identify start of document
  - Benefits: More challenging tasks lead to better representations
- **Contrastive Learning**
  - Learn similarities between related text pairs
  - Maximize agreement between positive pairs, minimize for negatives
  - Examples: SimCSE, CLIP (for multimodal learning)

# Scaling Laws

- **Parameter Scaling (Kaplan et al., 2020)**
  - Performance improves as power-law with model size
  - Loss $\propto N^{-a}$ where N is parameter count and $a \approx 0.076$
  - Larger models learn more efficiently per data token
  - Diminishing returns but no observed plateau
- **Data Scaling**
  - Performance improves with training data
  - Loss $\propto D^{-b}$ where D is dataset size and $b \approx 0.095$
  - High-quality, diverse data more important than quantity
  - Data exhaustion becomes limiting factor at extreme scales
- **Compute Scaling**
  - Performance improvements predictable with compute
  - Optimal allocation balances model size vs. training tokens
  - Compute-optimal training: train larger models on fewer steps
- **Chinchilla Scaling (Hoffmann et al., 2022)**
  - Revised optimal scaling rules
  - For compute-optimal training:
    - Model size (N) and dataset size (D) should scale equally
    - 20 tokens per parameter is optimal
  - Smaller, more thoroughly trained models outperform larger, undertrained ones

# Alignment Techniques

- **Supervised Fine-Tuning (SFT)**
  - Train on curated, high-quality examples of desired behavior
  - Examples created by human demonstrators
  - First step in alignment process
  - Relatively straightforward to implement
- **Reinforcement Learning from Human Feedback (RLHF)**
  - Three-stage process:
    - SFT: Initial fine-tuning on demonstration data
    - Reward Modeling: Train model to predict human preferences
    - RL Optimization: Optimize policy using reward model
  - Algorithms:
    - Proximal Policy Optimization (PPO)
    - Rejection Sampling Fine-Tuning
- **Reinforcement Learning with AI Feedback (RLAIF)**
  - Replace human feedback with AI evaluator models
  - Advantages:
    - Scalability: generate more feedback
    - Consistency: less variation in judgments
    - Cost efficiency
  - Still requires human oversight and calibration
- **Constitutional AI (CAI)**
  - Model critiques its own outputs
  - Uses predefined set of principles ("constitution")
  - Process:
    - Generate initial responses
    - Self-critique based on constitutional principles
    - Revise responses based on critique
    - Train on preferred revisions
  - Reduces reliance on direct human feedback

# Emergent Abilities with Scale

- **Emergence Definition**
  - Capabilities not present in smaller models
  - Qualitative jumps rather than smooth improvements
  - Often unpredicted by extrapolation from smaller scales
- **Key Emergent Abilities**
  - In-context learning: Few-shot and zero-shot learning
  - Instruction following without explicit training
  - Chain-of-thought reasoning
  - Code generation and understanding
  - Multimodal reasoning
  - Tool use and planning
- **Scaling Thresholds**
  - Different abilities emerge at different scales
  - Basic instruction following: ~10B parameters
  - Complex reasoning: ~100B parameters
  - Sophisticated tool use: 100B+ parameters
- **Theoretical Perspectives**
  - Phase transitions in learning dynamics
  - Compression breakthrough theory
  - Grokking phenomenon: sudden generalization after extended training

# Training Infrastructure

- **Distributed Training**
  - Data Parallelism: Same model, different data shards
  - Model Parallelism: Different model parts, same data
  - Pipeline Parallelism: Stage model across devices
  - Tensor Parallelism: Split individual tensors across devices
  - ZeRO (Zero Redundancy Optimizer): Optimize memory usage
- **Hardware Considerations**
  - GPU clusters with high-bandwidth interconnects
  - TPU pods for specialized training
  - Memory optimization techniques:
    - Gradient checkpointing
    - Mixed precision training
    - Activation recomputation
    - Optimizer state partitioning
- **Training Optimizations**
  - Optimizer selection: Adam, AdamW, Adafactor
  - Learning rate scheduling: Warmup, decay, cosine
  - Gradient accumulation
  - Flash Attention: Efficient attention computation
  - Activation functions: GELU, SwiGLU

# Advanced Training Approaches

- **Mixture of Experts (MoE)**
  - Sparse activation: only a subset of parameters active per token
  - Router network determines which expert handles each token
  - Benefits:
    - Increases parameter count without proportional compute
    - Specialization of different experts
  - Challenges:
    - Load balancing across experts
    - Router optimization
  - Examples: Switch Transformers, GLaM, Mixtral
- **Continual Pre-training**
  - Update models with new data over time
  - Maintain performance on old tasks while learning new information
  - Addresses temporal drift in knowledge
  - Techniques to prevent catastrophic forgetting:
    - Experience replay
    - Elastic weight consolidation
    - Knowledge distillation from previous versions
- **Multitask Training**
  - Train on multiple objectives simultaneously
  - Improves generalization and transfer learning
  - Examples:
    - T5: Multi-task language understanding
    - Flan: Instruction tuning across thousands of tasks
    - PaLM: Interleaved training on diverse data sources
- **Multimodal Training**
  - Incorporate non-text modalities: images, audio, video
  - Alignment between modalities using contrastive objectives
  - Examples: CLIP, Flamingo, GPT-4V
  - Challenges: Cross-modal attention, modality gaps

# Evaluation Frameworks

# Benchmark Datasets

- **General Language Understanding**
  - GLUE (General Language Understanding Evaluation)
    - Collection of 9 sentence/paragraph-level tasks
    - Tasks: sentiment analysis, paraphrase detection, NLI
    - Simple single-sentence or sentence-pair tasks
  - SuperGLUE
    - More challenging successor to GLUE
    - Includes more complex reasoning tasks
    - Example tasks: WiC (word sense disambiguation), ReCoRD (reading comprehension)
- **Knowledge and Reasoning**
  - MMLU (Massive Multitask Language Understanding)
    - 57 subjects across STEM, humanities, social sciences
    - Multiple-choice format tests world knowledge
    - Designed to assess breadth of knowledge
  - BIG-Bench
    - 204 diverse tasks beyond traditional NLP
    - Evaluates capabilities like logical reasoning, common sense
    - Community-contributed tasks with varying difficulty
- **Specialized Benchmarks**
  - TruthfulQA: Tests factuality and honesty
  - HumanEval/MBPP: Code generation and understanding
  - HellaSwag: Common sense reasoning through completion
  - MATH: Advanced mathematical problem-solving
  - BBH (Big-Bench Hard): Subset of most challenging BIG-Bench tasks

# Human Evaluation Approaches

- **Direct Assessment**
  - Human judges rate outputs on specific dimensions
  - Common dimensions:
    - Helpfulness: Utility of response for stated need
    - Harmlessness: Avoidance of toxic/harmful content
    - Honesty: Factual accuracy and appropriate uncertainty
    - Relevance: Appropriateness to query
  - Typically uses Likert scales (1-5 or 1-7)
- **Comparative Evaluation**
  - A/B testing between model outputs
  - Pairwise preferences more reliable than absolute ratings
  - Methods:
    - Best-of-n selection
    - Elo/Bradley-Terry ranking
    - Relative quality assessments
- **Expert Evaluation**
  - Domain specialists assess outputs in their field
  - Critical for specialized domains: medicine, law, science
  - Evaluate factual correctness and domain appropriateness
  - Examples: MedQA, LegalBench evaluations
- **Adversarial Testing**
  - Red-teaming: Experts try to elicit problematic outputs
  - Stress-testing boundaries of model capabilities
  - Identify failure modes and vulnerabilities
  - Generates examples for model improvement

# Metrics for Different Tasks

- **Text Generation Metrics**
  - BLEU: n-gram precision between generated and reference texts
  - ROUGE: Recall-oriented metric for summarization
  - METEOR: Handles synonyms, stemming, and word order
  - BERTScore: Semantic similarity using contextual embeddings
- **Model Performance Metrics**
  - Perplexity: Exponentiated average negative log-likelihood
    - Lower is better; measures prediction quality
    - Formula: $\exp(-1/N * \Sigma \log P(x_i | x_1 ... x_{i-1}))$
  - Cross-entropy loss: Average negative log-likelihood
  - Accuracy: Proportion of correct predictions (for classification)
  - F1 Score: Harmonic mean of precision and recall
- **Task-Specific Metrics**
  - Question Answering: Exact Match (EM), F1
  - Translation: BLEU, chrF, COMET
  - Summarization: ROUGE-L, BERTScore
  - Dialogue: Human-likeness, consistency, engagement
- **Efficiency Metrics**
  - Inference speed (tokens/second)
  - Memory usage
  - Training compute (FLOPs)
  - Parameter count vs. performance trade-offs

# Evaluation Beyond Accuracy

- **Safety Evaluation**
  - Dimensions:
    - Harmful content generation
    - Privacy violations
    - Fairness and bias
    - Manipulation and persuasion
  - Frameworks:
    - ToxiGen: Test for toxic language generation
    - HONEST: Evaluate for harmful, offensive content
    - RealToxicityPrompts: Real-world toxic prompts
    - Anthropic's Responsible Scaling Policy evaluations
- **Bias and Fairness**
  - Representation bias: Stereotypical associations
  - Allocation bias: Unequal treatment of different groups
  - Quality disparity: Performance varies across demographics
  - Datasets:
    - BOLD: Bias in Open-ended Language Generation
    - WinoBias: Gender bias in coreference resolution
    - BBQ: Bias benchmark for question answering

- **Truthfulness Assessment**
  - Factual consistency: Agreement with known facts
  - Hallucination detection: Identifying unsupported claims
  - Frameworks:
    - TRUE: Tracking Factuality in Generating
    - FActScore: Fact-level output verification
    - FELM: Factually Enhanced Language Models
- **Interpretability & Transparency**
  - Attribution: Identify sources of generated content
  - Confidence: Appropriate uncertainty in responses
  - Explainability: Justification for model decisions
  - Metrics: calibration error, explanation quality ratings

# Holistic Evaluation

- **Capability Taxonomies**
  - EleutherAI Language Model Evaluation Harness
    - Categorizes tasks by capability: reasoning, knowledge, etc.
    - Provides standardized suite across different models
  - Hugging Face Open LLM Leaderboard
    - Aggregates performance across multiple benchmarks
    - Enables comparison between open and closed models
- **Multidimensional Assessment Frameworks**
  - Anthropic's Responsible Scaling Policy
    - Evaluates across harmful capabilities and societal risks
    - Combines automated and human evaluation
  - HELM (Holistic Evaluation of Language Models)
    - Stanford's framework for comprehensive evaluation
    - Measures across accuracy, calibration, robustness, fairness
    - Standardized prompting and evaluation methodology
- **Evaluation Limitations**
  - Benchmark saturation: Models approach ceiling performance
  - Benchmark gaming: Models trained on evaluation data
  - Representation gaps: Missing real-world scenarios
  - Moving targets: Evaluation criteria evolve with capabilities

# Practical Evaluation Approaches

- **Automated Evaluation Pipelines**
  - Continuous testing during development
  - Regression testing across model versions
  - Test suites covering diverse capabilities
  - Red-teaming automation for safety testing
- **Meta-Evaluation**
  - Evaluating the evaluation methods themselves
  - Correlation between automatic metrics and human judgment
  - Reliability and validity of benchmark datasets
  - Reproducibility of evaluation results
- **Evaluation for Specific Use Cases**
  - Domain-specific testing (medical, legal, educational)
  - Task-targeted evaluation for deployment contexts
  - User experience studies for application integration
  - A/B testing in production environments
- **Emerging Evaluation Techniques**
  - LLM-as-judge: Using capable models to evaluate others
  - Self-consistency evaluation: Internal coherence across responses
  - Chain-of-thought evaluation: Assessing reasoning processes
  - Dynamic benchmarks: Adversarially updated to resist gaming