multiply

w0

Constant

wn-1

**…**

w1

output

**…**

wn

inputs

weights

activation function

Weighted sum

output

**Artificial Intelligence**

Visual perception

Natural language processing

Intelligent robot

Automatic programming

Automatic reasoning

Knowledge representation

**…**

**…**

**Machine learning**

Principal components analysis

Support vector machine

Linear/logistic regression

K-nearest neighbors

K-Means

Decision trees

Random forest

**Neural Networks**

**…**

**…**

Boltzmann neural networks

MPL

**Deep Learning**

**…**

**…**

CNN

RNN

GAN

DBN

**…**

**Input Layer**

**Output Layer**

**Hidden Layers**

**Hidden Layer 02**

**(10 neurons)**

**Hidden Layer 01**

**(10 neurons)**

**Input Layer**

**(784 pixels)**

**Output Layer**

**(10 neurons)**

**mathematics behind AI on Digit Recognition**

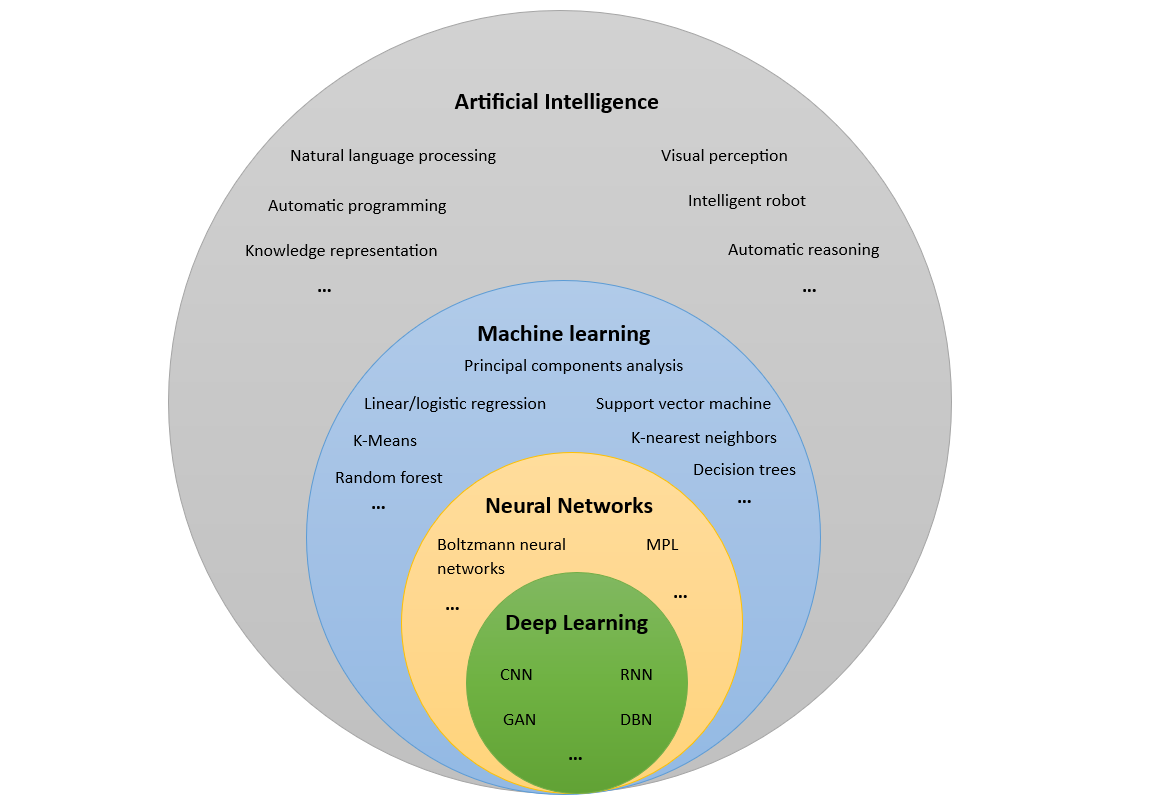
# Introduction:

*Hi everyone! I’m* [*devloker*](https://www.linkedin.com/in/dev-loker)*, and today I’m excited to share a project I’ve been working on: a digit recognition system implemented using pure math functions in Python. This project aims to help beginners grasp the mathematics behind AI and digit recognition without relying on high-level libraries like TensorFlow or PyTorch.*

You can find the complete code on my [GitHub repository](https://www.github/devloker/digits-recognition).

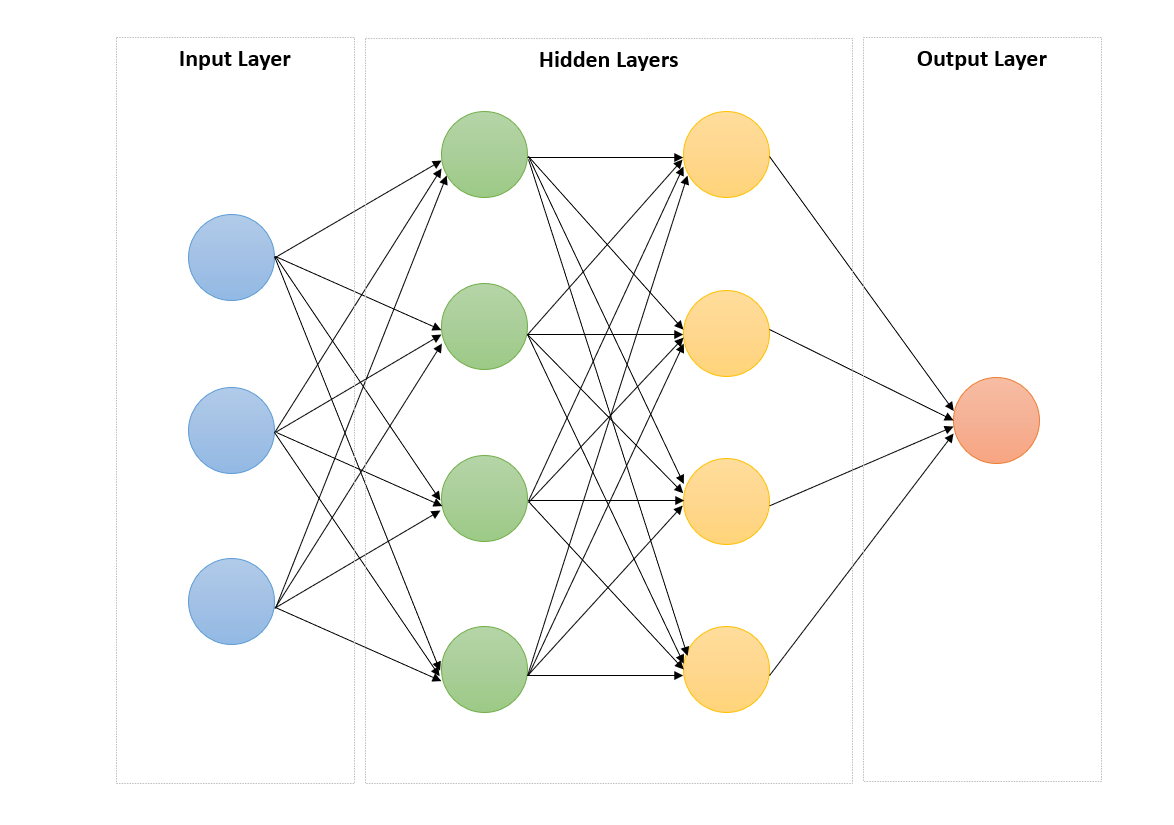
# Fundamental concepts in AI world

Artificial Intelligence (AI): Artificial Intelligence (AI) is a broad field of computer science focused on creating systems capable of performing tasks that typically require human intelligence. These tasks include problem-solving, understanding natural language, recognizing patterns, and making decisions. AI can be categorized into several subdomains, each with its own focus and techniques.



Machine Learning (ML): Machine Learning is a subset of AI that emphasizes the development of algorithms and statistical models that enable computers to learn from and make predictions or decisions based on data. Rather than being explicitly programmed to perform a task, ML models are trained using large datasets, allowing them to improve their performance over time.

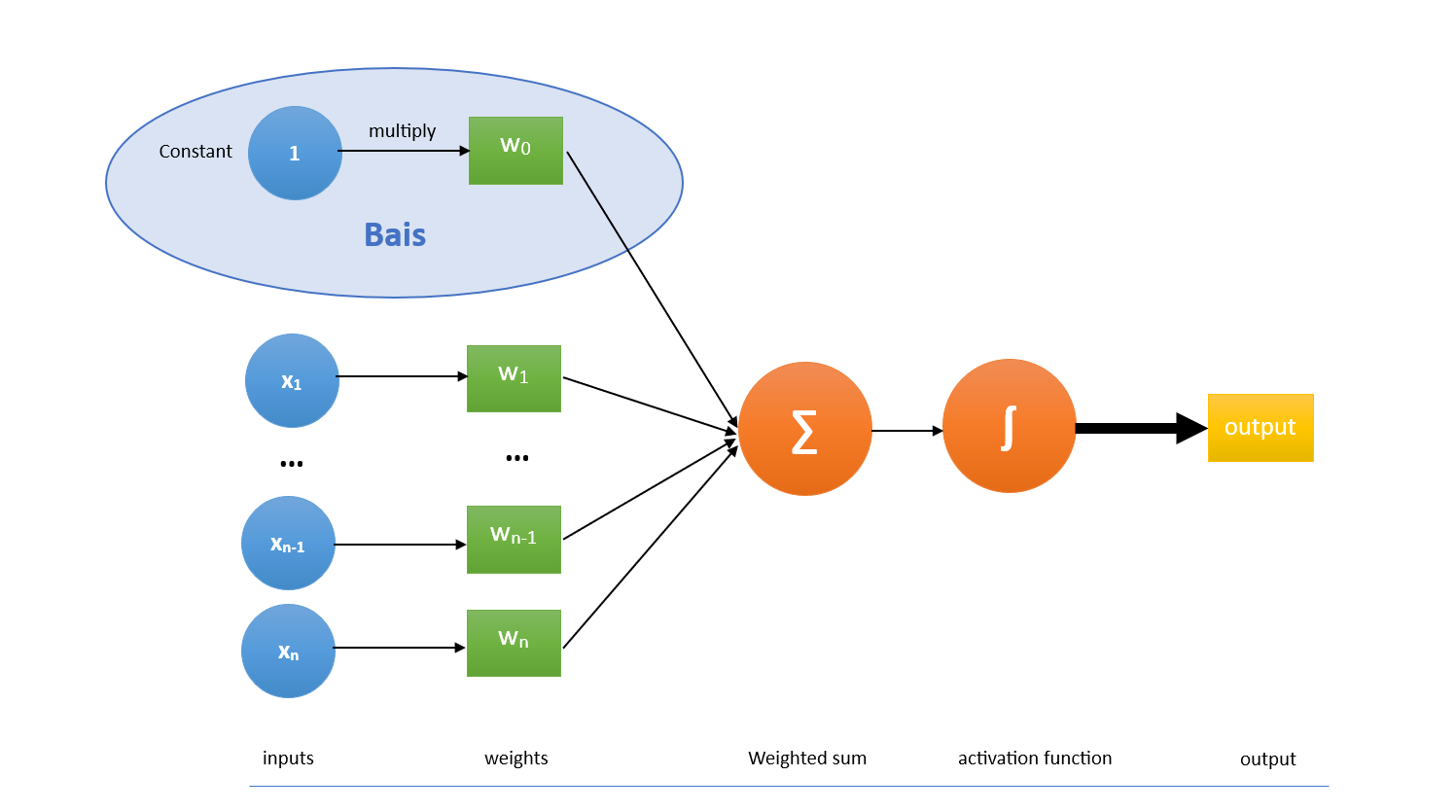
Artificial Neural Networks (ANN): Artificial Neural Networks are a type of machine learning model inspired by the structure and function of the human brain. They consist of layers of interconnected nodes (or neurons), each performing simple computations.



## Neuron

Neurons are the basic building blocks of artificial neural networks, inspired by biological neurons in the human brain. In AI, a neuron is a mathematical function that receives one or more inputs, applies weights to these inputs, sums them up, applies an activation function, and produces an output. In the context of artificial neural networks, a neuron performs the following operations:

* Input Features: The neuron takes multiple input features, each input represents a characteristic or attribute of the input data which represented as x1 ,x2​ , ... , xn​. In the context of digit recognition, these could be the pixel values of an image.
* Weights: Each input feature is associated with a weight w1, w2​, ...,wn​. which indicates the importance of the feature in making the prediction. During training, these weights are adjusted to learn the optimal values.
* Summation Function: Each input is multiplied by its weight, the weighted inputs are summed together, often with an added bias term: z = sum(xi \* wi for xi, wi in zip(x, w)) + b
* Bias: The bias b is an additional parameter that allows to make adjustments that are independent of the input, which helps the model make accurate predictions.
* Activation Function: This function decides whether the neuron should be fired or not based on weighted sum, introducing non-linearity to the model. Common activation functions include softmax, sigmoid, tanh, and ReLU (Rectified Linear Unit).
* Output: the neuron's output is the obtained result after applying the activation function. This output can be fed as input to the next layer of neurons or can be the final output in the case of the output layer, the final output represents the decision or prediction based on the input and the weights, in digits recognition problem, the output might represent a predicted class (from 0 to 9).



These operations work together to enable a neuron to learn and make predictions, while a single neuron can only solve linearly separable problems, combining multiple neurons into layers allows the creation of more complex models capable of solving non-linear problems. This structure forms the basis of multi-layer neural networks used in deep learning.

Deep Learning (DL):

Deep Learning is a subfield of machine learning that focuses on neural networks with many layers (hence "deep" networks). These networks are capable of learning from vast amounts of data and can model complex, high-dimensional patterns. Deep learning has been particularly successful in fields like speech & image recognition, natural language processing, medical diagnosis, and game playing. These models require vast amounts of data and computational power to train effectively but can achieve remarkable accuracy and performance.

Deep learning models consist of multiple layers of neurons. The common types of layers include:

* Input Layer: The first layer, which receives the initial data (e.g., pixel values of an image).
* Hidden Layers: Intermediate layers that transform the input of previous layer into more abstract representations through weighted connections and activation functions.
* Output Layer: The final layer, which produces the final prediction or classification (e.g., the probabilities of each digit in digit recognition).

Training deep networks involves adjusting the weights and biases of the network to minimize the error in predictions. This is done using backpropagation and optimization algorithms like gradient descent.

* **Forward Propagation**: Calculate the output of the network for given inputs.
* **Loss Function**: Measure the error between the predicted output and the actual output.
* **Backword Propagation**: Compute the gradient of the loss function with respect to each weight and bias, propagating the error backward through the network.
* **Weight Update**: during training, the perceptron learns by adjusting its weights and bias based on the difference between the predicted output and the true output.

Types of Deep Neural Networks:

* **Feedforward Neural Networks (FNNs)**: The simplest type where connections between the nodes do not form a cycle.
* **Convolutional Neural Networks (CNNs)**: Primarily used for image processing, recognizing patterns using convolutional layers.
* **Recurrent Neural Networks (RNNs)**: Suitable for sequence data like time series or text, where outputs from previous steps are fed as inputs to the next step.
* **Generative Adversarial Networks (GANs)**: Consist of two networks (generator and discriminator) that compete against each other, useful for generating synthetic data.

# Digits recognition process

Digit recognition is a classic application of neural networks where the goal is to correctly identify handwritten digits (0-9) from images. This task involves several key steps:

**1. Preparing the Data**

To start with digit recognition, we first need to prepare our data. We'll be using the MNIST dataset, a standard dataset consisting of 60,000 training images and 10,000 testing images of handwritten digits (0-9).

* **Loading the Data**: Load the MNIST dataset, which contains images of handwritten digits.
* **Normalizing**: Normalization involves scaling pixel values to a range of 0 to 1. This helps the model converge faster during training. Each pixel value, originally between 0 and 255, is divided by 255.
* **Reshaping**: Each image in the MNIST dataset is 28x28 pixels. We'll reshape these 2D arrays into 1D vectors of 784 elements (28 \* 28). This reshaped vector will serve as the input features for our model.

Here’s a sample code snippet for data preparation:

import numpy as np

from keras.datasets import mnist

*# Load the MNIST dataset*

(X\_train, y\_train), (X\_test, y\_test) = mnist.load\_data()

*# Normalize the images to values between 0 and 1*

X\_train = X\_train / 255.0

X\_test = X\_test / 255.0

*# Reshape the images from (28, 28) to (784,)*

X\_train = X\_train.reshape(-1, 784)

X\_test = X\_test.reshape(-1, 784)

**X\_train**: A numpy array containing the training images. Each image is a 28x28 pixel grayscale image of a handwritten digit (0-9). The shape of X\_train is typically (60000, 28, 28), where 60000 is the number of training images.

**y\_train**: A numpy array containing the labels for the training images. Each label is an integer (0-9) representing the digit shown in the corresponding training image. The shape of y\_train is typically (60000,), where 60000 is the number of training labels.

**X\_test**: A numpy array containing the testing images. Similar to X\_train, each image is a 28x28 pixel grayscale image. The shape of X\_test is typically (10000, 28, 28), where 10000 is the number of testing images.

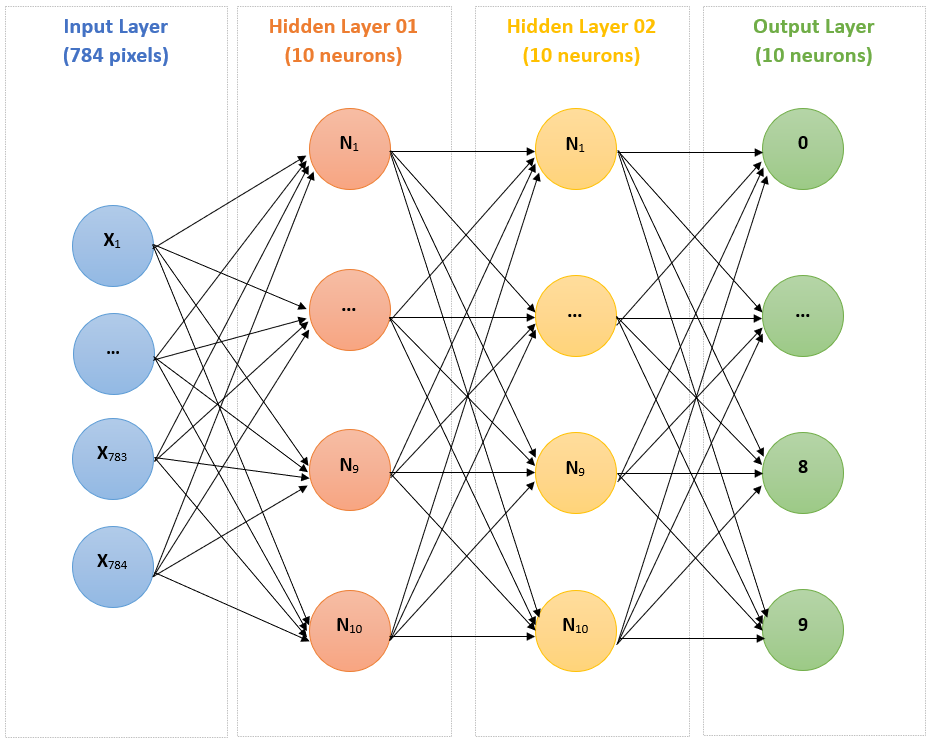
**y\_test**: A numpy array containing the labels for the testing images. Each label is an integer (0-9) representing the digit shown in the corresponding testing image. The shape of y\_test is typically (10000,), where 10000 is the number of testing labels.

**2. Model Architecture**

Our neural network will consist of an input layer, two hidden layers, and an output layer. The structure and sizes of weights and biases for each layer are as follows:

* **Input Layer**: 784 neurons (one for each pixel in the image).
* **Hidden Layer 1**: 10 neurons.
  + Each neuron is connected with 784 input neurons from the previous layer, so have 10\*784 different weights, and we can store them in a matrix **(W1)** with size of (10, 784)
  + Each neuron has own bais, in total we have 10 different biases, and we can store them in vector **(B1)** with size of (10,1)
* **Hidden Layer 2**: 10 neurons.
  + Each neuron is connected with 784 input neurons from the previous layer, so have 10\*10 different weights, and we can store them in a matrix **(W2)** with size of (10, 10)
  + Each neuron has own bais, in total we have 10 different biases, and we can store them in vector **(B2)** with size of (10,1)
* **Output Layer**: 10 neurons (one for each digit 0-9),
  + Each digit assigns by one neuron, and each neuron represents the probability of the assigned digit.
  + the predicted digit is the one corresponding to the neuron with the highest probability.

Here’s the structure of our model:



*# Initialize weights and biases*

W1 = np.random.rand(10, 784)

B1 = np.random.rand((10, 1))

W2 = np.random.rand(10, 10)

B2 = np.random.rand((10, 1))

**3. Training the Model**

Training the model involves forward propagation, loss and accuracy calculation, backward propagation, and updating weights and biases.

**Initialize the** neural network's weights W1, W2 and biases B1, B2 with random float between 0 and 1.

For each epoch from 1 to epochs (inclusive):

* + Forward Propagation: Compute the activations A1 and A2 (output of each layer) using the current weights and biases.
  + Compute error: quantifies the error in the predictions using output A2 and the true labels y\_train.
  + Compute Accuracy,which is the proportion of correctly predicted labels (evaluate the performance of the model).
  + Backward Propagation:compute the gradients of the error with respect to the weights and biases. Witch indicate how much each parameter needs to change to reduce the error.
  + Update the model parameters**:** Adjust the weights W1, W2 and biases B1, B2 using the computed gradients and the learning rate.

End For Loop

Return the last version of the trained model parameters (weights W1, W2 and biases B1, B2)

* **Forward Propagation**: Calculating the activations of each layer using the weights and biases. We'll use the ReLU (Rectified Linear Unit) activation function for the hidden layers and softmax for the output layer.

def relu(*z*):

    return np.maximum(0, *z*)

def softmax(*z*):

    exp\_z = np.exp(*z* - np.max(*z*))

    return exp\_z / exp\_z.sum(*axis*=0, *keepdims*=True)

def forward\_propagation(*X*):

    Z1 = np.dot(W1, *X*) + B1

    A1 = relu(Z1)

    Z2 = np.dot(W2, A1) + B2

    A2 = softmax(Z2)

    return A1, A2

**Loss and Accuracy Calculation**: Using cross-entropy loss to measure the model's performance and calculating accuracy.

def compute\_loss(*A2*, *Y*):

    m = *Y*.shape[1]

    log\_probs = np.multiply(np.log(*A2*), *Y*)

    cost = -np.sum(log\_probs) / m

    return np.squeeze(cost)

def compute\_accuracy(*A2*, *Y*):

    predictions = np.argmax(*A2*, *axis*=0)

    labels = np.argmax(*Y*, *axis*=0)

    accuracy = np.mean(predictions == labels)

    return accuracy

**Backward Propagation**: Calculating gradients using derivatives to adjust the weights and biases. The chain rule is applied here.

def backward\_propagation(*X*, *Y*, *A1*, *A2*):

    m = *X*.shape[1]

    dZ2 = *A2* - *Y*

    dW2 = np.dot(dZ2, *A1*.T) / m

    dB2 = np.sum(dZ2, *axis*=1, *keepdims*=True) / m

    dA1 = np.dot(W2.T, dZ2)

    dZ1 = dA1 \* (*A1* > 0)

    dW1 = np.dot(dZ1, *X*.T) / m

    dB1 = np.sum(dZ1, *axis*=1, *keepdims*=True) / m

    return dW1, dB1, dW2, dB2

*# Updating weights and biases*

def update\_parameters(*W1*, *B1*, *W2*, *B2*, *dW1*, *dB1*, *dW2*, *dB2*, *learning\_rate*):

*W1* -= *learning\_rate* \* *dW1*

*B1* -= *learning\_rate* \* *dB1*

*W2* -= *learning\_rate* \* *dW2*

*B2* -= *learning\_rate* \* *dB2*

    return *W1*, *B1*, *W2*, *B2*

**Training Loop**: Iteratively performing forward and backward propagation, and updating weights and biases.

learning\_rate = 0.01 # alpha

epochs = 1000 # iterations

for epoch in range(epochs):

    A1, A2 = forward\_propagation(X\_train.T)

    cost = compute\_loss(A2, y\_train.T)

    accuracy = compute\_accuracy(A2, y\_train.T)

    dW1, dB1, dW2, dB2 = backward\_propagation(X\_train.T, y\_train.T, A1, A2)

    W1, B1, W2, B2 = update\_parameters(W1, B1, W2, B2, dW1, dB1, dW2, dB2, learning\_rate)

    if epoch % 100 == 0:

        print(f'Epoch {epoch}, Cost: {cost}, Accuracy: {accuracy}')

**4. Evaluating and Reusing the Model**

After training, we evaluate the model's performance on the test set and discuss how to save and reuse the model.

* **Saving the Model**: The trained model, represented by its weights and biases, can be saved to a file for future use.

import pickle

model\_parameters = {'W1': W1, 'B1': B1, 'W2': W2, 'B2': B2}

with open('digit\_recognition\_model.pkl', 'wb') as file:

    pickle.dump(model\_parameters, file)

**Loading and Evaluating the Model**: Load the saved model and evaluate its performance on new, unseen data.

**Accuracy**: In the context of digit recognition and neural networks, accuracy is a key metric used to evaluate the performance of the model. It represents the proportion of correctly predicted digits out of the total number of predictions made. High accuracy indicates that the model is effectively learning and generalizing from the training data to make correct predictions on unseen data.

with open('digit\_recognition\_model.pkl', 'rb') as file:

    model\_parameters = pickle.load(file)

W1 = model\_parameters['W1']

B1 = model\_parameters['B1']

W2 = model\_parameters['W2']

B2 = model\_parameters['B2']

*# Evaluate the model on the test set*

A1, A2 = forward\_propagation(X\_test.T)

test\_accuracy = compute\_accuracy(A2, y\_test.T)

print(f'Test Accuracy: {test\_accuracy}')

By following these steps, we can build, train, and evaluate a neural network for digit recognition using the MNIST dataset. This process highlights the importance of data preparation, model architecture, training, and evaluation in developing effective machine learning models.