# Module 4

## Deep Learning Models

## 📌 Shallow vs Deep Neural Networks

### 🔹 Differences Between Shallow and Deep Networks

Although there is no strict cutoff, the following is a commonly accepted rule of thumb:

* A **shallow neural network** has **one or two hidden layers**.
* A **deep neural network** has **three or more hidden layers**, typically with a **large number of neurons per layer**.

The increase in depth allows deep networks to model much more **abstract**, **hierarchical**, and **non-linear** representations of the input data.

| **Feature** | **Shallow Neural Networks** | **Deep Neural Networks** |
| --- | --- | --- |
| Depth | 1–2 hidden layers | 3 or more hidden layers |
| Input | Typically vectorized features | Capable of handling raw data (images, text, etc.) |
| Feature Engineering | Often requires manual feature extraction | Automatically extracts features |
| Representation Power | Limited | High (can learn hierarchical representations) |
| Use Cases | Simple, structured data | Complex data like images, audio, NLP, etc. |

Shallow networks are easier to understand and train, which makes them useful for **conceptual learning**, **prototyping**, and **simple tasks**.

Deep networks, on the other hand, are preferred when dealing with **complex input data** and problems that require **high abstraction**.

### 🔹 Why Deep Learning Took Off

Although neural networks have existed for decades, the explosive growth and adoption of **deep learning** in recent years can be attributed to **three major breakthroughs**:

1. **Advancements in the Field:**

The development of **better activation functions**, especially **ReLU (Rectified Linear Unit)**, helped address one of the most critical limitations in deep learning: the **vanishing gradient problem**. This allowed deeper networks to train more effectively and enabled the creation of models with dozens or even hundreds of layers.

✅ ReLU maintains stronger gradients during backpropagation, which makes deep networks trainable at scale.

1. **Availability of Large Datasets**

Deep neural networks have an incredible capacity to **memorize and model large volumes of data**. However, this ability can lead to overfitting if the data is limited. The widespread availability of **large, labeled datasets** has unlocked the full potential of deep architectures.

✅ Unlike traditional machine learning algorithms, deep networks continue to improve with more data.

1. **Computational Power (GPUs)**

Training deep networks is computationally intensive. The rise of **powerful GPUs** (and later, TPUs and distributed computing) has made it feasible to:

* Train deep models on massive datasets.
* Perform experiments rapidly.
* Deploy models to production with manageable costs.

✅ What once took days or weeks can now be trained in hours using modern hardware.

### ✅ Takeaways

✅ **Shallow neural networks** are defined by **1–2 hidden layers**, and typically require **vectorized inputs** and **manual feature extraction**.

✅ **Deep neural networks** consist of **multiple hidden layers**, and are capable of **automatically extracting features** from **raw input data** such as images and text.

✅ The modern success of deep learning is attributed to:

* **Better training techniques and architectures** (e.g., ReLU)
* **Widespread access to large datasets**
* **Powerful hardware that accelerates training**

## 📌 Convolutional Neural Networks

**Convolutional Neural Networks (CNNs)** are a specialized type of deep learning architecture designed specifically for handling **image data** and other data with a grid-like structure (e.g., audio spectrograms).

Although CNNs are composed of neurons, weights, biases, and activation functions like standard dense neural networks, their design incorporates unique **architectural assumptions** that allow them to learn from raw image data more efficiently and with fewer parameters.

### 🔹 Why Deep Learning Took Off

CNNs make the **explicit assumption** that inputs are **images**, and they process them using filters that take advantage of the **spatial structure** of the data.

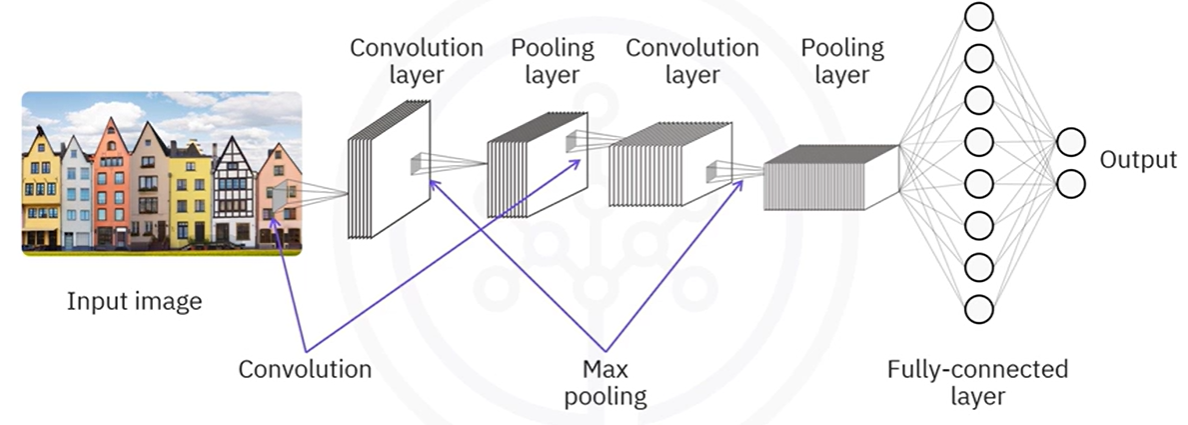
Input shapes:

* Grayscale images: n x m x 1
* Color images: n x m x 3 (where 3 channels represent RGB)

CNNs are particularly powerful for:

* Image classification
* Object detection
* Image segmentation
* Face recognition
* Medical imaging analysis

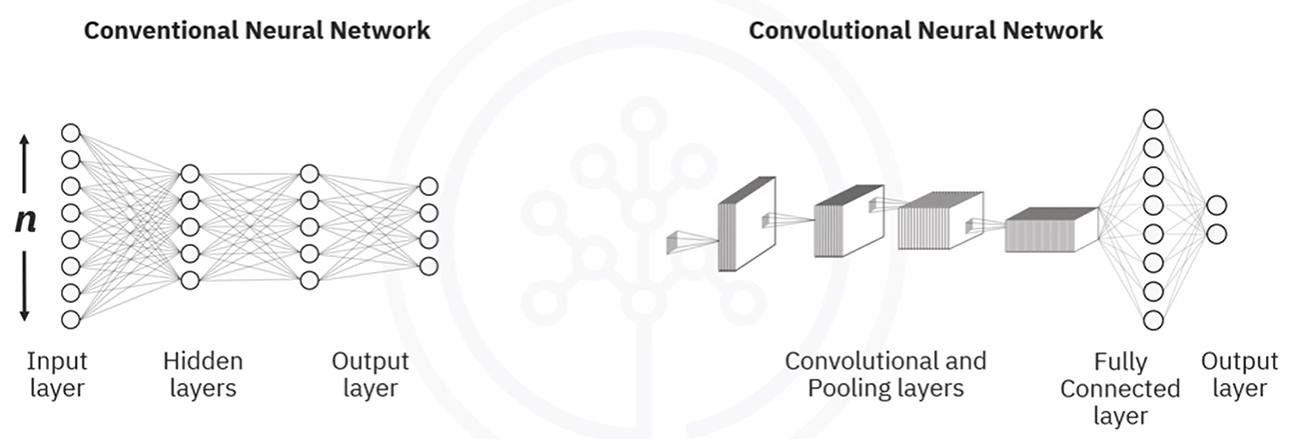
### 🔹 CNN Architecture Breakdown



A typical **Convolutional Neural Network (CNN)** is composed of several key building blocks organized in a specific order to progressively extract spatial features from input images and make accurate predictions.

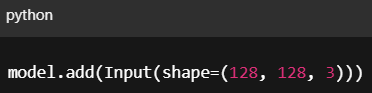
CNNs differ from the fully connected networks introduced earlier in that they assume **images as input**, maintaining their spatial structure. Instead of flattening the data up front, CNNs process **3D input tensors** using specialized layers that are better suited for handling **pixel-level patterns**.

1. **Input Layer:**



Unlike shallow networks that accept a **1D vector**, CNNs accept **3D inputs**:

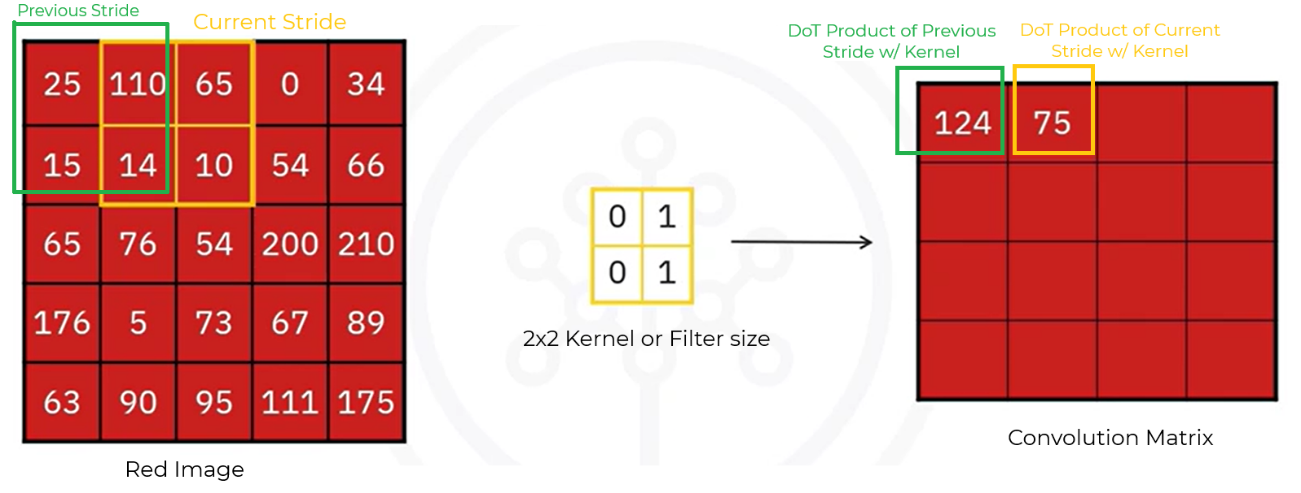
* For grayscale images: height × width × 1
* For color images: height × width × 3 (RGB channels)

For example, a 128×128 RGB image would have an input shape of (128, 128, 3).

This structure allows the model to **preserve spatial relationships** between pixels — crucial for tasks like object detection and image classification.

1. **Convolutional Layer:**

This is the **core feature-extracting layer** in a CNN. It applies a set of **filters** (also called kernels) that scan across the image in small patches.



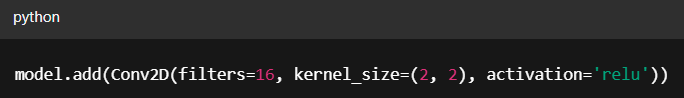
In the convolutional layer, we basically define filters and we compute the convolution between the defined filters and each of the three images (RGB).

This is the **core feature-extracting layer** in a CNN. It applies a set of **filters** (also called kernels) that scan across the image in small patches.

This structure allows the model to **preserve spatial relationships** between pixels — crucial for tasks like object detection and image classification.

Each filter performs a **dot product** between its values and a local patch of the image and stores the result in a new **feature map**.

* Typical filter sizes: 2×2, 3×3, 5×5
* Stride: how far the filter moves per step (usually 1)
* Depth: number of filters determines number of feature maps



**⚠️ Why not flatten the input image into an [ (n x m) x 1 ] vector and pass it into a dense network?**

* Flattening large images would result in **huge parameter counts**, increasing computational cost and overfitting risk.
* Convolution drastically reduces parameters by reusing filters across regions of the image, while maintaining spatial locality.

ℹ️ **ReLU Activation (within Conv2D)**

Each convolutional layer is followed by a **ReLU (Rectified Linear Unit)** activation function. This layer:

* Keeps only **positive values**
* Sets all **negative values to zero**
* Adds **non-linearity** to the model

ReLU ensures the model can learn complex, non-linear representations efficiently, and avoids the vanishing gradient issues of older activations (like sigmoid).

1. **Pooling Layer (MaxPooling2D / AveragePooling2D):**

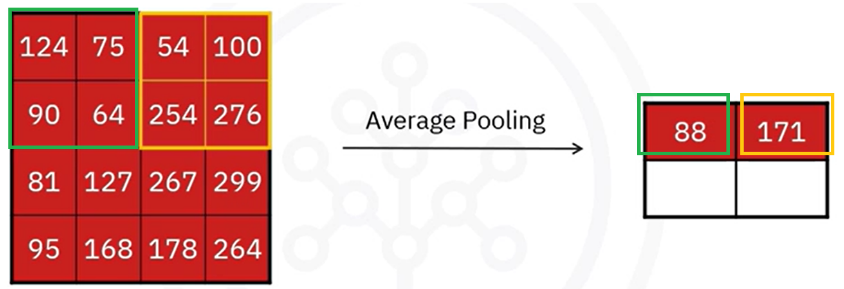
The pooling layer **reduces the spatial dimensions** of the feature maps. It helps the network become more invariant to small shifts or distortions in the input image.

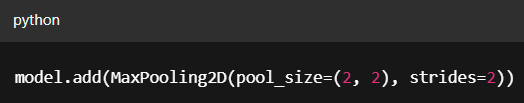
There are two common types:

* **Max Pooling**: keeps the largest value in each region.



* **Average Pooling**: computes the average of each region



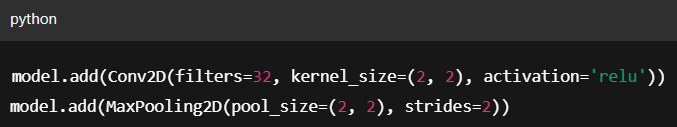
Example: with a 2×2 filter and a stride of 2, the pooling operation slides over the image in blocks, shrinking the dimensions while retaining key features.

✅ Pooling offers:

* **Dimensionality reduction**
* **Noise suppression**
* **Spatial invariance**

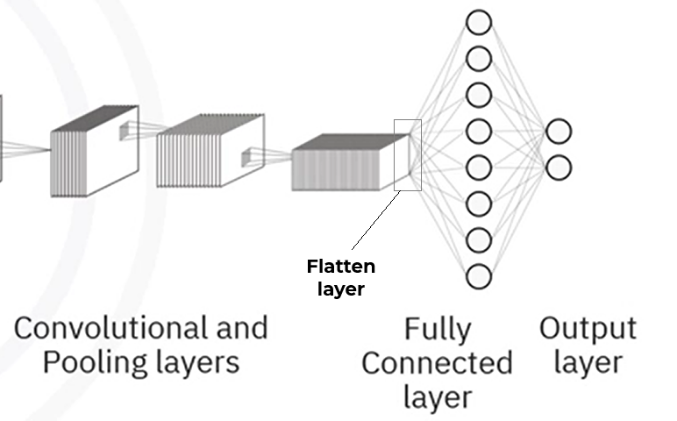
#### 🔁 Repeat: Additional Convolution + Pooling Blocks

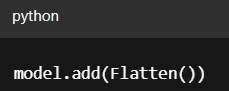
Deeper CNNs stack multiple convolution and pooling layers. Typically:



* Deeper layers use **more filters** to capture more abstract and complex features.
* Each additional block allows the network to **build a hierarchy of features**, from edges and textures to shapes and object parts.

1. **Flatten Layer:**



Before passing data to a fully connected layer, the 3D output of the final convolution/pooling layer is **flattened** into a 1D vector.

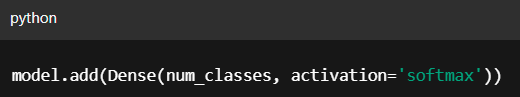
⚠️ This conversion is **necessary** because **dense layers expect flat inputs**. The flattened vector represents all the high-level features extracted from the image.

1. **Fully-Connected Layer:**

These layers operate like the traditional dense layers introduced earlier:

* Each node in one layer is connected to **every node** in the next.
* Often used to **combine features** and make the final prediction.

1. **Fully-Connected Layer:**

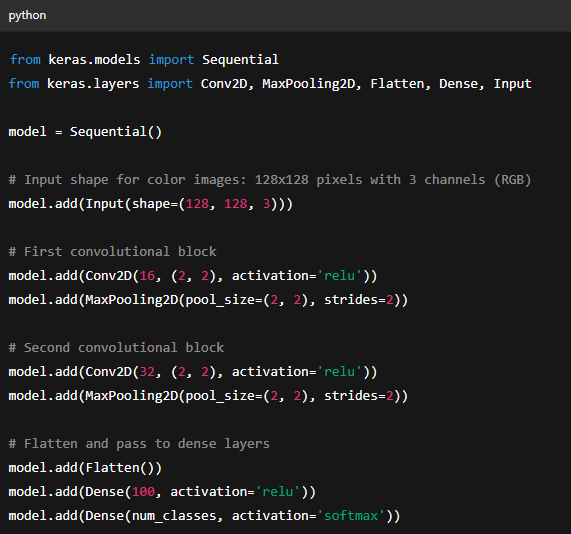
****The final output layer produces predictions. Its size depends on the number of classes:

* Use **softmax activation** to convert output values into probabilities that sum to 1.
* Each neuron corresponds to one possible class.

### 🔹 Complete Flow of CNN Layers

| **Layer** | **Purpose** |
| --- | --- |
| Input | Accepts image in 3D shape (e.g., 128×128×3) |
| Convolution (Conv2D) | Detects features by applying filters |
| ReLU | Adds non-linearity, filters out negative values |
| Pooling | Reduces spatial size and introduces spatial invariance |
| (Repeat Conv + Pool) | Extracts deeper and more abstract patterns |
| Flatten | Converts 3D data into 1D vector for Dense layers |
| Dense (Hidden) | Learns global patterns, combines features |
| Dense (Output) | Outputs class probabilities (softmax) |

### 🔹 Building a CNN in Keras

****

**Conv2D:** Applies filters to extract features from images.

**MaxPooling2D:** Reduces spatial size and adds spatial invariance.

**Flatten:** Converts 3D feature maps to 1D vector.

**Dense**: Learns class-specific patterns and outputs final predictions.

### ✅ Takeaways

✅ CNNs are optimized for processing **image data** using a **layered structure** of convolutions, activations, pooling, and dense layers.

✅ The architecture:

* Preserves spatial structure
* Reduces the number of parameters
* Extracts hierarchical features

✅ CNNs are the go-to architecture for solving complex **computer vision problems** like classification, object detection, and segmentation.

✅ Keras provides a clean and modular way to define CNNs using Conv2D, MaxPooling2D, Flatten, and Dense layers.

## 📌 Recurrent Neural Networks (RNNs)

### 🔹 What Are RNNs?

Traditional neural networks treat each input data point as **independent** of the others. This assumption works well for tasks like image classification or tabular data, but fails for any problem where **sequence and order matter**.

For example:

* Understanding a sentence in natural language.
* Predicting stock prices based on prior days.
* Analyzing audio or video streams.

In such cases, the context provided by **previous inputs** is essential. This is where **Recurrent Neural Networks (RNNs)** come in.

**Recurrent Neural Networks (RNNs)** are a type of neural network architecture specifically designed to handle **sequential or time-series data**.

**✅ Key difference:**

RNNs process inputs **one at a time**, maintaining an internal **state** (or memory) that reflects previous inputs. Each step in the sequence considers both:

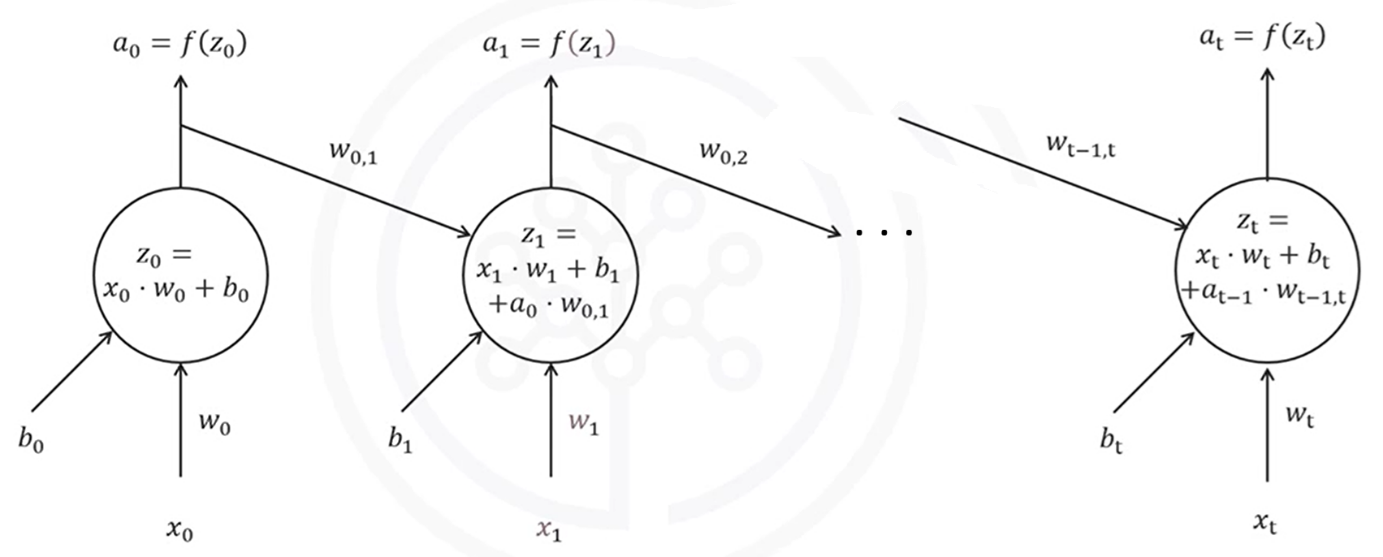
* The **current input**
* The **output or hidden state from the previous step**

This creates a **loop** within the architecture, allowing the network to build **temporal dependencies** between data points.

### 🔹 RNN Architecture

The key architectural idea is that the **output from a previous time step is fed back into the network** along with the current input.

Let’s walk through the architecture based on the diagram:



**🧩 Notation**

* : Input at time step t.
* : Weighted sum (pre-activation) at time step t.
* : Activation (output) at time step t.
* : Weight matrix for input xₜ.
* : Weight matrix for previous activation .
* : Bias term at time step t
* : Activation function (e.g., tanh or ReLU).

**🧠 How the RNN Processes Sequences**

At each time step t, the RNN performs the following operations:

1. **Receives the current input .**
2. **Receives the previous output (activation from the previous time step).**
3. **Computes a weighted sum:**
4. **Applies the activation function to produce the output:**
5. **Passes aₜ to the next time step.**

This process repeats for each element in the sequence, preserving temporal relationships across steps.

**🔁 Time-Unrolled Architecture**

The RNN can be "unrolled" across time steps to visualize how it processes sequences. Each copy of the network at time t:

* Has its **own input**
* Shares weights across all steps
* Passes its activation forward to the next step

Here’s what happens step-by-step:

* **At time step t=0**:
  + The network receives input .
  + Computes:
  + Applies activation:
* **At time step t=1**:
  + Receives and as inputs.
  + Computes:
  + Applies activation:
* **At time step t**:
  + Receives and .
  + Computes:
  + Applies activation:

This chaining of activations makes it possible to retain context over time — unlike standard feedforward networks.

### 🔹 Common Use Cases

RNNs are suitable for any task involving ordered data:

| **Use Case** | **Description** |
| --- | --- |
| Natural Language Processing (NLP) | Sentiment analysis, translation, text generation |
| Time Series Forecasting | Stock prices, weather, sales prediction |
| Genomic Sequence Modeling | Predicting genetic features from DNA/RNA sequences |
| Audio Processing | Speech recognition, music generation |
| Handwriting Generation | Modeling the sequence of pen strokes |

### 🔹 Why RNN Architecture Works

* The **feedback loop** allows the model to accumulate knowledge across steps.
* **Shared weights** reduce the number of parameters and help generalization.
* The structure naturally models **ordered data** (e.g., language, audio, video, sequences).

⚠️ However, traditional RNNs struggle to retain information over long sequences due to the **vanishing gradient problem**. This led to the development of more advanced architectures like **LSTM (Long Short-Term Memory) model**, which include gating mechanisms to better manage memory.

### 🔹 Long Short-Term Memory – A Specialized RNN

A key challenge with standard RNNs is the **vanishing gradient problem**, which makes it difficult to learn long-range dependencies across time steps.

To solve this, **Long Short-Term Memory (LSTM)** units were introduced.

**✅ LSTMs use:**

* **Gating mechanisms** to control what to remember, forget, and output at each step.
* Internal **memory cells** that store relevant past information across long sequences.

Because of this, LSTMs have become the most widely used RNN variant.

🔧 **LSTMs Applications:**

LSTM models have been successfully used in several advanced deep learning applications:

* **Image Generation**: Sequentially generating pixels conditioned on prior ones.
* **Handwriting Generation**: Modeling realistic sequences of handwritten text.
* **Image Captioning**: Generating natural language descriptions for images.
* **Video Description**: Producing textual summaries of video sequences.

### ✅ Takeaways

✅ Standard neural networks treat all data points independently, which limits their use in sequential tasks.

✅ **Recurrent Neural Networks (RNNs)** introduce feedback loops, enabling them to retain memory of past inputs and learn temporal relationships.

✅ **LSTMs** are a more powerful and stable variant of RNNs, designed to capture **long-term dependencies** without suffering from vanishing gradients.

✅ RNNs (and LSTMs) are foundational for tasks involving **language, audio, sequential data**, and **temporal reasoning**.

## 📌 Transformers

**Transformers** are a type of neural network architecture that has fundamentally transformed the field of **Natural Language Processing (NLP)** and beyond. Unlike RNNs or CNNs, which rely on processing data sequentially or in spatial chunks, transformers are capable of modeling **long-range dependencies** in data efficiently — and **in parallel**.

They are the foundation behind some of the most powerful AI tools today, including:

* **ChatGPT, Gemini** (based on Generative Pretrained Transformers)
* **BERT** (used in Google Search and Translate)
* **DALL·E** and **image transformers** (used for text-to-image synthesis in tools like Adobe Photoshop)

### 🔹 The Attention Mechanism

The key breakthrough that enables transformers to outperform previous models is the **attention mechanism**, particularly **self-attention** and **cross-attention**.

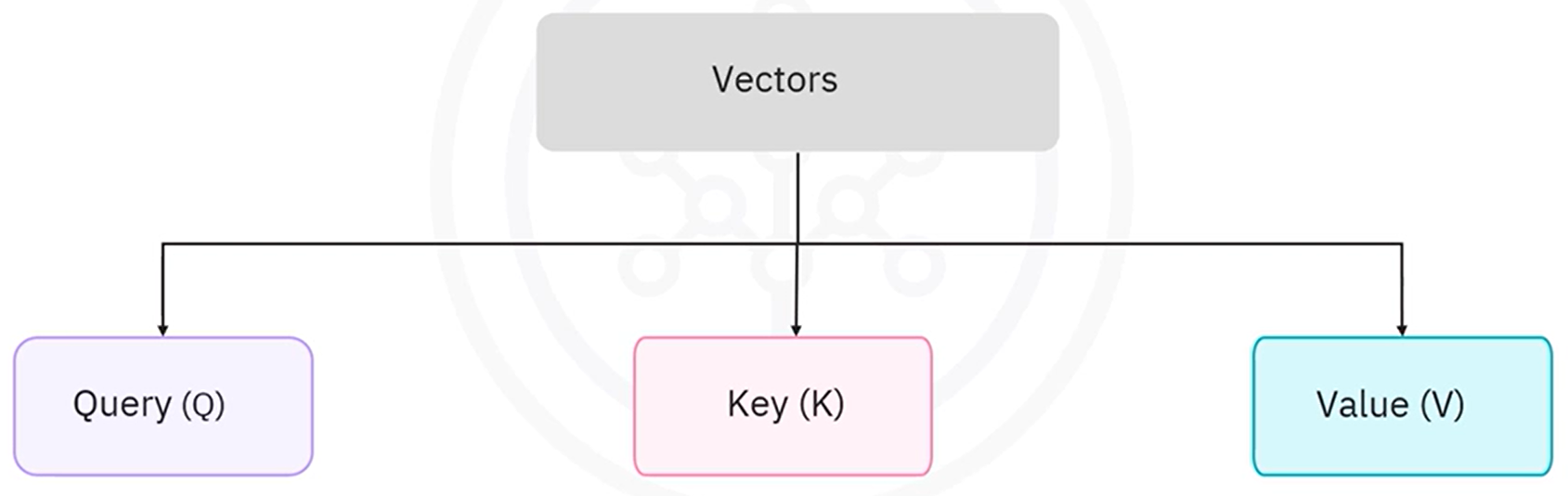
This mechanism allows the network to **evaluate relationships between input tokens**, weighting the importance of each part of a sequence **relative to every other part**. As a result, the model captures **global context** more effectively than traditional architectures.

### 🔹 Self-Attention Mechanism (for Text)

The transformers use a self-attention mechanism to process textual data.

The **self-attention mechanism** enables a model to contextualize a token (word, character, etc.) by attending to other tokens in the same input sequence. It consists of three major components:

* 1. **Query, Key, and Value Vectors (Q, K, V)**

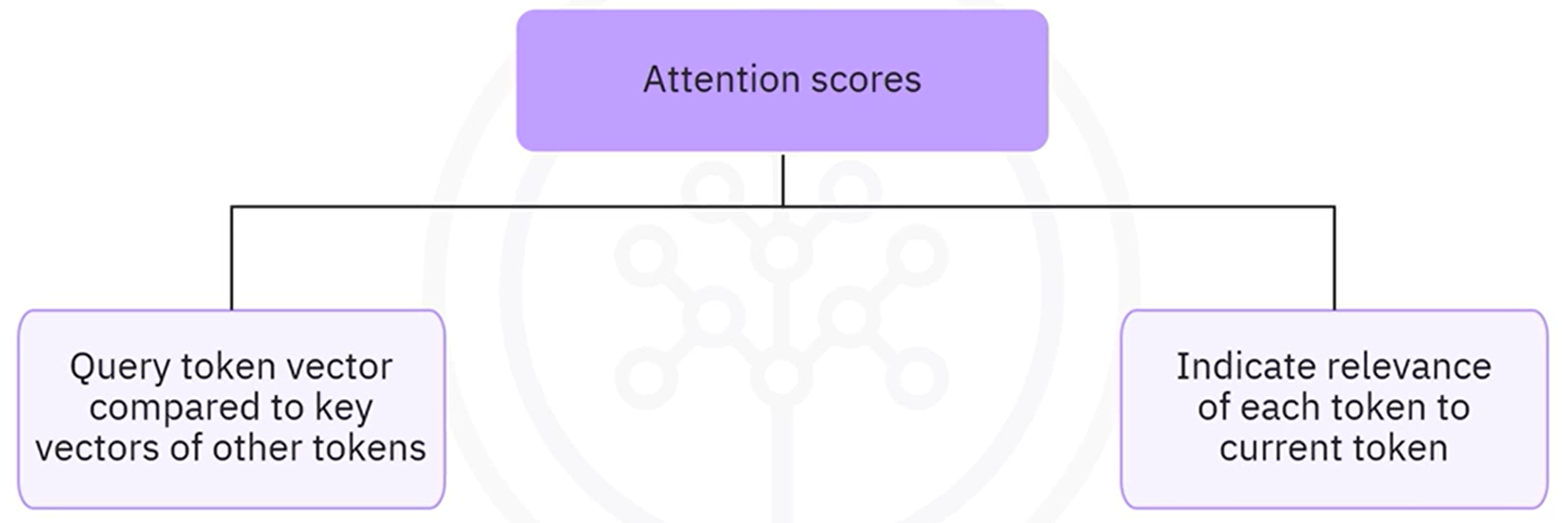


For each input token, the model creates three vectors:

* **Query (Q):** Represents the current token being processed. Is used to ask: “Which other words are important to me?”.
* **Key (K):** Represents all tokens being compared against. Each word’s identity.
* **Value (V):** Carries the actual information to be passed forward to next layer.

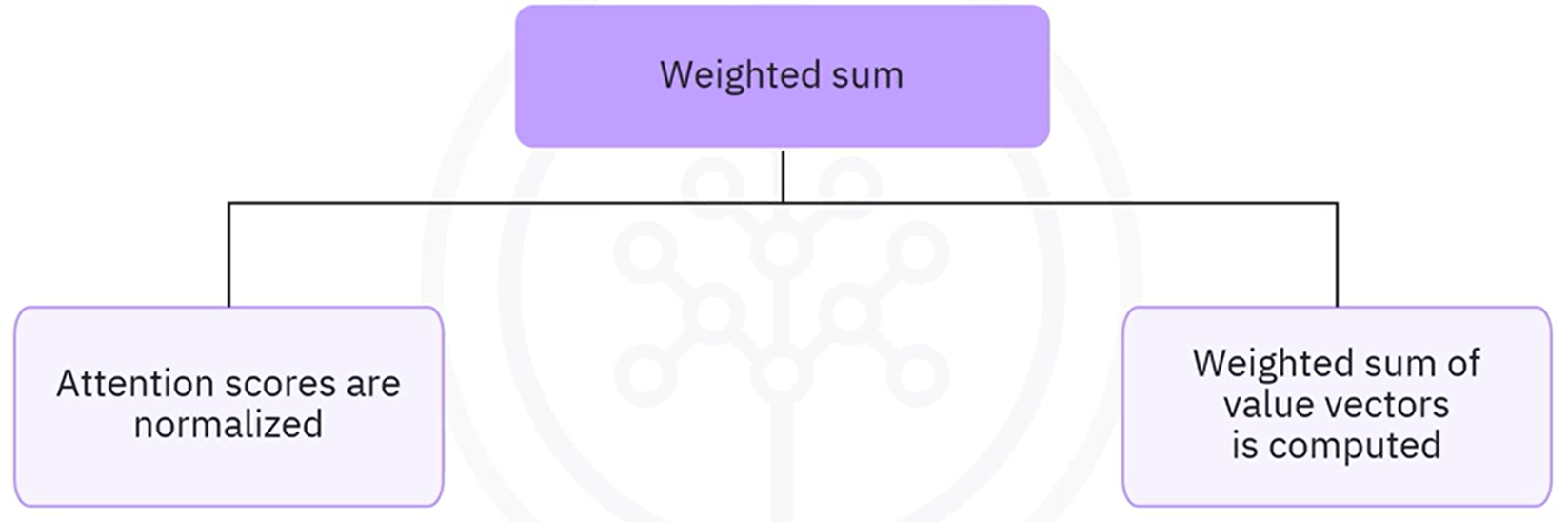
Each of these vectors is obtained through linear projections from the token embeddings.

* 1. **Attention Scores**



* The **attention score** is computed using the **dot product** between the query vector of the current token and the key vectors of all tokens in the sequence.
* These raw scores indicate **how relevant** other tokens are to the current one

1. **Weighted Sum**



* The scores are scaled and passed through a **softmax function** to generate normalized attention weights (probabilities).
* A **weighted sum** of the value vectors is computed based on these attention scores.
* This results in a **context vector** for each token that captures global information from the sequence.

The final contextual embeddings are passed to the next layer, enabling deep modeling of sentence-level semantics.

#### ⚙️ Example of Self-Attention

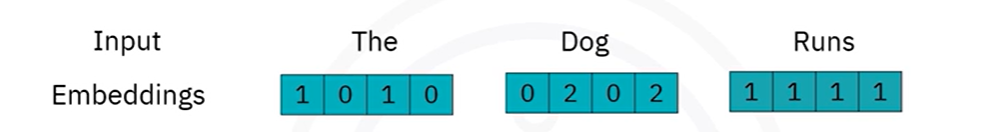
For the sentence -> **"The dog runs"**

* 1. Each word is first converted into an **embedding vector**.

**Input tokens** are first converted into **dense vectors**, also known as **embeddings**. These embeddings are numerical representations that capture the meaning and relationships between words.

Embeddings are **learned** during training via backpropagation, each word has a **unique embedding vector.**

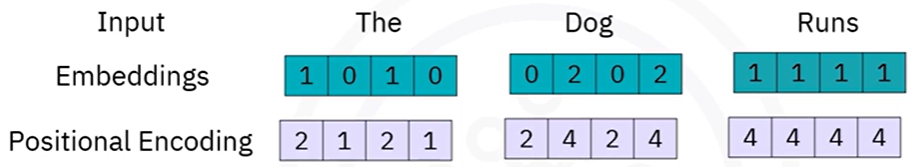
Each token is mapped to a fixed-size vector



* 1. **Positional encoding**

Since transformers don’t have recurrence, they don’t inherently know the **order** of tokens. To add order information, a **positional vector** is added to each token’s embedding.

Resulting vector = **Word embedding + Positional encoding**

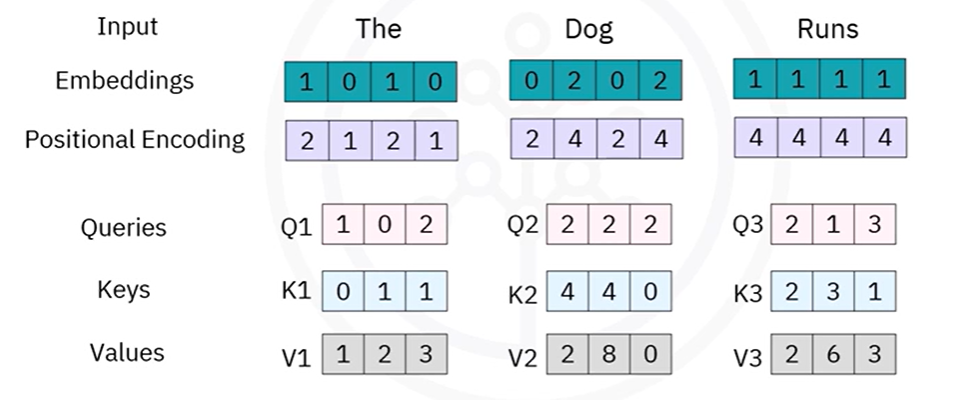


Position-aware input vectors:

* "The" = [1 0 1 0] + [2 1 2 1] = [3, 1, 3, 1]
* "Dog" = [0 2 0 2] + [2 4 2 4] = [2, 6, 2, 6]
* "Runs" = [1 1 1 1] + [4 4 4 4] = [5, 5, 5, 5]
  1. Q, K, and V vectors are generated from each embedding, by:

Applying **separate linear transformations** (matrices) to each token's embedding.

Each transformation is **learned during training**, specific to the attention head.



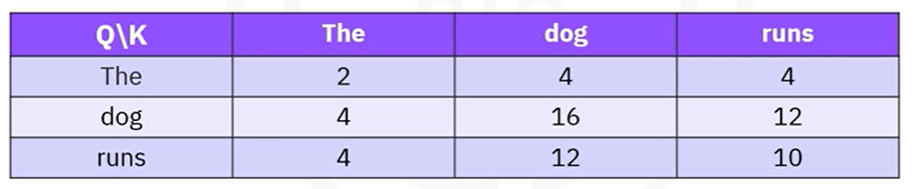
Each word's combined embedding (after positional encoding) is passed through **three different weight matrices** to produce:

* Query vector (Q) ->
* Key vector (K) ->
* Value vector (V) ->

Where:

* **E** is the [word + position] embedding
* are **learnable weight matrices** (different for Q, K, V)
* Output dimensions can be chosen (e.g., 4D → 4D)
  1. Attention scores are computed for each token pair using the **dot product** of Query and Key vectors ->

Each **row corresponds to a Query** and each **column to a Key**.



This table tells us:

1. The **query for "Dog"** gives **highest relevance to itself (16)**.
2. The **query for "Runs"** is most related to "Dog" (12), then "Runs" (10), then "The" (4).
3. These raw attention scores will be **normalized via softmax** in the next step.
4. Softmax normalization is applied to generate weights, Converting Attention Scores into probabilities

This produces a **probability distribution** over the tokens, summing to 1, for example:

* The: 0.05
* Dog: 0.65
* Runs: 0.30

These are called the **attention weights** — they determine *how much each token contributes to the current token’s final representation*.

1. Weighted sums of the Value (V) vectors with the **attention weights**,addingcontext-aware embeddings.

This gives us the **output vector for the current word**, now enriched with contextual information from other tokens.

ℹ️ In **self-attention**, each token (or word) in the input sequence generates its **own probability distribution** over **all tokens**, including itself.

Example, using token “**dog**”:

* + - * Vectors are:
        + The -> [1, 2, 3]
        + Dog -> [2, 8, 0]
        + Runs -> [2, 6, 3]
      * Probability distribution for “dog” token -> [0.05, 0.65, 0.3]
      * Compute the new embedding:

This new vector is the **contextual embedding for the word "Dog"** — it captures not just the word itself, but also the influence of surrounding words like "The" and "Runs".

### 🔹 Cross-Attention Mechanism (for Text-to-Image)

Transformers are also extensively used for **text-to-image generation**, which is made possible through another form of attention known as the **cross-attention mechanism**.

While self-attention focuses on understanding relationships within a **single sequence** (like a sentence), **cross-attention allows one type of input to influence another**, such as using a text prompt to guide image generation.

Cross-attention works in **three main phases**, building on the contextual understanding created during self-attention.

* 1. Learning Contextualized Embeddings with Self-Attention.

Given a natural language prompt such as:

*"A two-story house with a red roof and a garden in front"*

The transformer model first uses a **self-attention mechanism** to learn contextualized embeddings from the entire sentence. Each word in the prompt is processed in the context of the other words, resulting in a **sequence of embeddings** that represent the full semantic meaning of the sentence.

These embeddings are passed through a **transformer encoder**, which produces a set of **Query vectors (Q)** representing the textual input.

* 1. Applying Cross-Attention for Image Generation.

Next, the transformer model responsible for **image generation** (e.g., **DALL·E**) takes over. It uses the **Query vectors from the text encoder** and performs a **cross-attention operation** setup to determine **how the text prompt should influence the visual output**., with its own internal image tokens.

In this mechanism:

* The **Query (Q)** comes from the **textual input**.
* The **Keys (K)** and **Values (V)** are derived from **image representations** or partially generated image data.
  1. Applying Cross-Attention for Image Generation.

The image is generated **sequentially** using an **auto-regressive approach**:

* At each generation step, the model predicts the **next part of the image**.
* The prediction is based on:
  + The **text prompt (via Q from cross-attention)**
  + The **previously generated image parts**

This allows the model to **compose an image piece-by-piece**, guided by the meaning of the original text.

The output image is **synthesized from scratch** based on the model’s learned understanding of text-image relationships.

This enables the model to:

* Combine **concepts that may not exist in the real world**
* Create **whimsical or creative combinations**
* Generate **multiple variations** of the same prompt, allowing for creative exploration

### 🔹 Transformers vs RNNs

While both **Transformers** and **Recurrent Neural Networks (RNNs)** are used for processing sequential data, they differ significantly in how they handle sequences — especially in terms of performance, scalability, and ability to capture long-range dependencies.

| **Feature** | **RNNs** | **Transformers** |
| --- | --- | --- |
| Data Processing | Sequential (step-by-step) | Parallel (entire sequence at once) |
| Training Speed | Slower (not parallelizable) | Faster (fully parallelizable) |
| Dependency Modeling | Short-term dependencies | Long-range dependencies |
| Vanishing Gradient Risk | High (especially in long sequences) | Mitigated via attention |
| Suitability | Simple or short-sequence tasks | Complex, long-context tasks (NLP, vision) |

### 🔹 Limitations of Transformers

Despite their capabilities, transformers also present certain challenges:

* **Data Hungry**: Require large volumes of training data to generalize well.
* **Bias Amplification**: Because learning is purely data-driven, **biases present in training data** can be embedded and perpetuated.
* **Resource Intensive**: Training and inference with large transformer models demand **significant computational resources**.

These limitations have motivated ongoing research into more efficient and responsible variants of transformer models.

### ✅ Takeaways

✅ Transformers are a type of neural network architecture designed to handle sequential data using attention mechanisms instead of recurrence.

✅ The **self-attention mechanism** allows each token in a sequence to focus on other tokens, capturing long-range dependencies more effectively than RNNs.

✅ Transformers process data **in parallel**, enabling significantly faster training compared to RNNs, which handle data sequentially.

✅ The **cross-attention mechanism** enables multimodal generation tasks, such as producing images based on text descriptions.

✅ Unlike models that retrieve stored content, transformers can synthesize **entirely new outputs** based on their understanding of the input (e.g., text-to-image generation).

✅ Despite their strengths, transformers require **large datasets** and can **inherit biases** from the data they are trained on.

✅ Transformers are now the backbone of state-of-the-art systems in NLP and generative AI, powering models like ChatGPT, BERT, and DALL·E.

📌 💡 ✅ ❌ 🚫 ⚠️ **🔍** ℹ️ **🧠** ⚙️ 🔧 **🧰** 🛠 🧩