

5 Deep Sequential and Transformer Architectures

Modern Artificial Intelligence (AI) systems increasingly deal with *sequential data*, such as language, speech, time-series signals, and user interactions. Unlike static inputs, sequences exhibit temporal dependencies, contextual relationships, and evolving patterns over time. In such data, the meaning of an element is often influenced not only by its own value but also by its position and its relationship with preceding and succeeding elements. For example, in natural language, the interpretation of a word depends heavily on its surrounding words, while in time-series data, future values are influenced by past observations.

Traditional machine learning approaches struggled to effectively model such dependencies, particularly when long-range relationships were involved. This limitation led to the development of specialized neural architectures capable of capturing sequential dynamics. Early neural sequence models, such as Recurrent Neural Networks (RNNs), introduced the concept of maintaining a hidden state that evolves over time. However, challenges such as vanishing gradients and limited memory capacity restricted their ability to learn long-term dependencies. Subsequent advancements, including Long Short-Term Memory (LSTM) networks, addressed some of these issues through gating mechanisms that regulate information flow.

A major paradigm shift occurred with the introduction of the *attention mechanism*, which enabled models to selectively focus on relevant parts of the input sequence rather than relying solely on sequential processing. This innovation paved the way for the *transformer architecture*, which replaces recurrence with attention-based parallel processing, significantly improving scalability and performance. Transformers form the backbone of modern AI systems, including Large Language Models (LLMs) that demonstrate remarkable capabilities in understanding and generating human-like text.

This chapter introduces the evolution from classical sequence models to modern deep architectures, culminating in the *transformer paradigm*, which powers contemporary systems such as LLMs. The chapter builds conceptual clarity step-by-step, moving from recurrent structures to attention mechanisms and large-scale pretrained systems.

5.1 EVOLUTION OF SEQUENCE MODELING

Sequence modeling refers to the process of learning patterns from *ordered data*, where the arrangement of elements carries meaningful information. Unlike independent data points, sequences exhibit *temporal dependencies*, meaning that the current element depends on previous elements.

Typical examples include: (i) Natural language sentences (ii) Speech signals (iii) Time-series sensor data and (iv) User interaction logs.

Formally, a sequence is represented as:

$$X = (x_1, x_2, x_3, \ldots, x_T),$$

where T denotes the total length of the sequence.

Sequence X: The symbol X represents an *ordered collection of elements*, where the order is crucial. Unlike a set, where elements are unordered, a sequence preserves temporal or positional structure. For example, in a sentence, the meaning depends heavily on the order of words; similarly, in time-series data, values are meaningful only with respect to their position in time.

Elements x_t: Each x_t denotes the element at position (or time step) t in the sequence. The index t typically ranges from 1 to T , where T is the total number of elements. Depending on the application, x_t may represent a word (in language), a signal value (in time-series), or a feature vector (in machine learning).

Sequence length T: The parameter T defines the length of the sequence, i.e., the number of elements it contains. In real-world applications, T may vary across different sequences. For instance, sentences in natural language can have different lengths, and sensor readings may span different durations.

Dependency on context: The expression 'each element x_t depends on its preceding context' means that the current element is not independent; rather, it is influenced by the earlier elements in the sequence. This context is typically denoted as:

$$x_{<t} = (x_1, x_2, \ldots, x_{t-1}),$$

which represents all elements that occur before position t. This dependency captures the fundamental nature of sequential data, where past information influences future outcomes.

The central goal of sequence modeling is to estimate:

$$P(X) = \prod_{t=1}^{T} P\left(x_t \mid x_{<t}\right)$$

Sequence probability P(X): The term $P(X)$ represents the probability of observing the entire sequence X. It quantifies how likely the sequence is under a given model. In applications such as language modeling, a higher probability indicates that the sequence is more natural or plausible.

Chain rule decomposition: The product form:

$$\prod_{t=1}^{T} P\left(x_t \mid x_{<t}\right)$$

is derived from the **chain rule of probability**. It decomposes the joint probability of the entire sequence into a product of conditional probabilities. This allows us to model complex sequences step-by-step, rather than attempting to estimate the full joint probability directly.

Conditional probability $P(x_t \mid x_{<t})$: Each term $P(x_t \mid x_{<t})$ represents the probability of the current element x_t, given all previous elements in the sequence. This is the core component of sequence modeling, as it captures how past context influences future predictions. For example, in language, the probability of a word depends strongly on the words that precede it.

Thus, sequence modeling can be understood as a *step-by-step prediction process*. At each time step t, the model uses the past context $x_{<t}$ to predict the next element x_t. By multiplying these probabilities across all time steps, we obtain the overall likelihood of the sequence. This formulation forms the foundation for both classical models (such as n-grams and Hidden Markov Models (HMMs) and modern deep learning models (such as RNNs and transformers).

5.1.1 Classical Approaches

Before the emergence of deep learning, sequence modeling was dominated by *probabilistic models*, primarily:

- **n-gram models**
- **HMMs**

These models laid the mathematical foundation for modern sequence learning.

n-gram Models: Local Context Modeling

n-gram models approximate the probability of a sequence using a *fixed-size context window*. Instead of conditioning on the entire history, they rely on the *Markov assumption*:

$$P(x_t \mid x_1, x_2, \ldots, x_{t-1}) \approx P(x_t \mid x_{t-(n-1)}, \ldots, x_{t-1}).$$

Thus, only the last $n - 1$ tokens influence the prediction.

Classification view: n-gram models can be interpreted as **discriminative classifiers** over the vocabulary:

$$\hat{x}_t = \arg\max_{w \in V} P(w \mid \text{context}).$$

That is, given a context, the model selects the most probable next token.

Types of n-gram Models

1. **Unigram model ($n = 1$: No context):** The unigram model is the simplest form of language modeling, where each token is assumed to be *independent of all others*. The model ignores word order completely and relies only on how frequently each word appears in the corpus. Thus, the probability of a sequence is given by:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t).$$

Example: Consider a corpus with the following word counts:

$$C(\text{the}) = 5, C(\text{cat}) = 3, C(\text{sits}) = 2, T = 10.$$

Then:

$$P(\text{the}) = 0.5, P(\text{cat}) = 0.3, P(\text{sits}) = 0.2.$$

For the sentence:

'the cat sits',

$$P(\text{the cat sits}) = 0.5 \times 0.3 \times 0.2 = 0.03.$$

Even incorrect sequences like 'sits cat the' would receive the same probability, since order is ignored.

2. **Bigram model ($n = 2$: One-step context):** The bigram model assumes that each token depends only on the *previous token*. The model captures *local dependencies*, meaning it learns which words are likely to follow others:

$$P(x_t \mid x_1, \ldots, x_{t-1}) \approx P(x_t \mid x_{t-1}).$$

Thus, the sequence probability becomes:

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P\left(x_t \mid x_{t-1}\right).$$

Example: Consider the corpus:

$\langle s \rangle$ *the cat sits* $\langle /s \rangle$
$\langle s \rangle$ *the cat sleeps* $\langle /s \rangle$

Counts:

$$C(\text{the, cat}) = 2, C(\text{cat}) = 2, C(\text{cat, sits}) = 1.$$

Thus:

$$P(\text{cat} \mid \text{the}) = 1, \text{P}(\text{sits} \mid \text{cat}) = 0.5.$$

For the sentence:

'the cat sits',

$$P = P(\text{the} \mid \langle s \rangle) \times P(\text{cat} \mid \text{the}) \times P(\text{sits} \mid \text{cat}) = 1 \times 1 \times 0.5 = 0.5.$$

Unlike unigram, incorrect sequences like 'cat the sits' get probability 0, since such transitions are not observed.

3. **Trigram model (n = 3: Two-step context):** The trigram model assumes that each token depends on the *previous two tokens*. The trigram model captures richer context than bigram and can model short phrases more accurately:

$$P(x_t \mid x_1, \ldots, x_{t-1}) \approx P(x_t \mid x_{t-2}, x_{t-1}).$$

Thus:

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P\left(x_t \mid x_{t-2}, x_{t-1}\right).$$

Example: Consider the corpus:

$\langle s \rangle$ *I saw the dog* $\langle /s \rangle$
$\langle s \rangle$ *I saw the cat* $\langle /s \rangle$

Counts:

$$C(I, saw, the) = 2, C(saw, the) = 2, C(saw, the, dog) = 1.$$

Thus:

$$P(\text{the} \mid I, saw) = 1, P(\text{dog} \mid saw, the) = 0.5.$$

For the sentence:

'I saw the dog',

$$P = 1 \times 1 \times 1 \times 0.5 = 0.5.$$

The model can distinguish contexts like:
'I saw the' vs 'You saw the'
which is not possible in bigram models.

As summarized in Table 5.1, different n-gram models vary in their context size and dependency assumptions.

Strengths and Limitations

n-gram models are simple, interpretable, and computationally efficient, as they rely on frequency-based probability estimation from observed data. They are particularly effective in capturing *local sequential dependencies*, making them useful for short-context predictions. However, they suffer from several limitations—they cannot model *long-range dependencies*, their performance degrades due to *data sparsity* as n increases, and they exhibit *exponential growth in parameters*, making them less scalable for large vocabularies and higher-order contexts.

TABLE 5.1: Comparison of unigram, bigram, and trigram models

Models	Context Size	Dependency	Example
Unigram	0	Independent	$P(w)$
Bigram	1	Previous word	$P(w_i \mid w_{i-1})$
Trigram	2	Previous two words	$P(w_i \mid w_{i-2}, w_{i-1})$

Hidden Markov Models: Sequential Generative Modeling

The HMMs extend probabilistic modeling by introducing *hidden states*. The HMM separates *latent structure (states)* from *observed data (symbols)*, enabling modeling of complex temporal processes where the underlying mechanism is hidden.

An HMM consists of:

- **Hidden states:** $\Omega = \{\omega_1, \omega_2, \ldots, \omega_N\}$
- **Observable symbols:** $V = \{v_1, v_2, \ldots, v_M\}$
- **Transition probabilities:** a_{ij}
- **Emission probabilities:** b_{jk}

The model assumes:

- The system evolves through hidden states
- Each state emits an observable symbol

Generative process:

$$\omega(t) \to \omega(t+1),\ \omega(t) \to v(t)$$

Sequence probability:

$$P(V^T \mid \theta) = \sum_{\Omega^T} P(V^T \mid \Omega^T)\, P(\Omega^T).$$

Classification view: HMMs are widely used for *sequence classification*:

$$\hat{c} = \arg\max_c P(V^T \mid \theta_c),$$

where each class c has its own model θ_c.

Example (Speech recognition): Different words correspond to different HMMs:

- 'pin' $\to$ model θ_1
- 'nip' $\to$ model θ_2

The observed signal is classified based on maximum likelihood.

Exemplary Interpretation of Hidden Markov Model Structure

Figure 5.1 provides a complete representation of a HMM with:

- **Three regular hidden states:** $\omega_1, \omega_2, \omega_3$
- **One absorbing state:** ω_0
- **Observable symbols:** v_1, v_2, v_3, v_0

Hidden states: Each node (ω_i) represents a hidden internal condition of the system. These states are not directly observable.

Transition probabilities (a_{ij}): Directed edges between states represent transition probabilities:

- a_{11}, a_{22}, a_{33}: Self-loops (state persistence)
- a_{12}, a_{23}, a_{31}: Transitions between states
- a_{10}, a_{20}, a_{30}: Transitions to absorbing state

These satisfy:

$$\sum_j a_{ij} = 1.$$

Emission probabilities (b_{jk}): Each hidden state emits observable symbols:

- **From ω_1:** Emits v_1, v_2, v_3
- **From ω_2:** Emits v_1, v_2, v_3

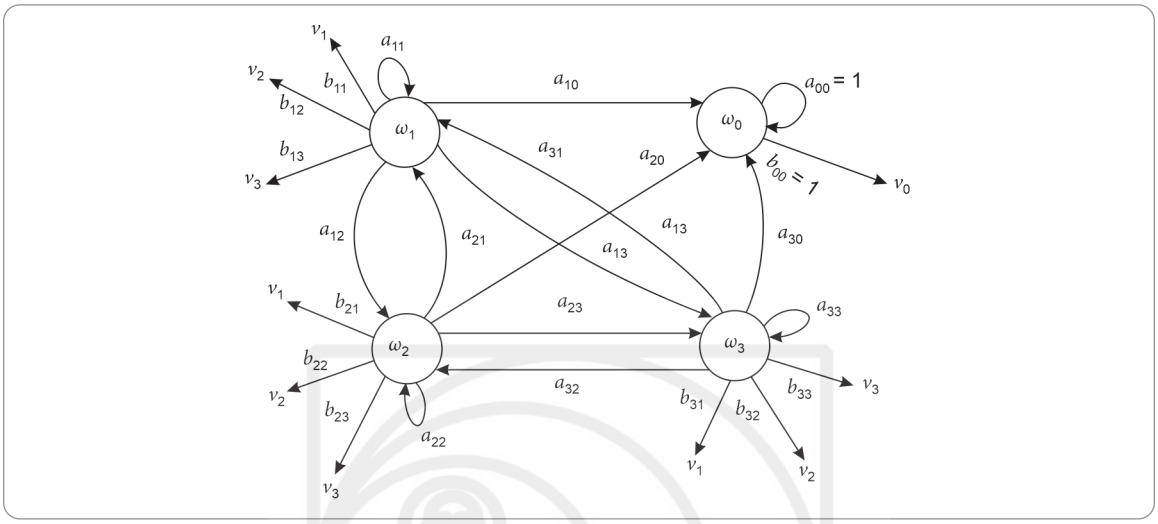


Fig. 5.1: Hidden Markov model with three hidden states and an absorbing state ω_0

- **From ω_3:** Emits v_1, v_2, v_3

$$b_{jk} = P(v_k \mid \omega_j)$$

Absorbing state (ω_0): This state represents sequence termination:

$$a_{00} = 1, a_{0i} = 0$$

- Once entered, the model cannot leave
- Emits a special symbol v_0

Temporal flow: The model evolves as:

$$\omega_1 \rightarrow \omega_2 \rightarrow \omega_3 \rightarrow \omega_0$$

At each step:

- A state transition occurs
- An observation is generated

As summarized in Table 5.2, classical sequence models such as n-grams and HMMs exhibit several fundamental limitations.

Despite their importance, both n-gram models and HMMs exhibit inherent limitations. They rely on *limited context* due to the fixed Markov assumption, making them incapable of capturing long-term dependencies in sequences. Additionally, they suffer from *data sparsity*, as many possible sequences are not observed in

TABLE 5.2: Key limitations of n-gram model and Hidden Markov model

Aspects	n-gram model	Hidden Markov model
Model type	Statistical (count-based)	Generative probabilistic
Context	Fixed window	State-based memory
Hidden structure	No	Yes
Dependencies	Short-range	Sequential dynamics
Learning	Frequency counts	EM (Baum-Welch)
Applications	Text prediction	Speech, gesture recognition

finite datasets, leading to unreliable probability estimates. Their *rigid assumptions* about independence and fixed-order dependencies further restrict their expressiveness. Moreover, these models face *poor scalability*, as the number of parameters grows exponentially with vocabulary size and context length. These challenges collectively motivated the transition toward more powerful deep learning-based sequence models.

Key Insight

Classical models rely on fixed context windows and explicit probability estimation, whereas deep models learn dynamic, distributed representations of context directly from data.

5.2 WORD REPRESENTATIONS AND EMBEDDINGS

In the previous section, we saw that classical sequence models such as n-grams and HMMs rely on explicit probability estimation and fixed contextual assumptions. Although these models are mathematically important, they are limited in their ability to represent rich semantic and contextual relationships among words. Modern neural models overcome this difficulty by learning representations directly from data. However, before a neural network can process language, words must first be converted into numerical form because neural networks operate only on numbers, not on raw text tokens. This gives rise to the important concept of *word representation*.

5.2.1 One-Hot Encoding

One of the earliest and simplest methods for representing words numerically is *one-hot encoding*. In this scheme, each word in the vocabulary is assigned a vector whose length equals the size of the vocabulary. All entries of the vector are 0, except for a single position corresponding to that word, which is set to 1. Thus, each word is represented uniquely, but without any notion of similarity to other words.

For example, suppose the vocabulary is:

$$\{\text{cat, dog, bird}\}.$$

Then the one-hot representations are:

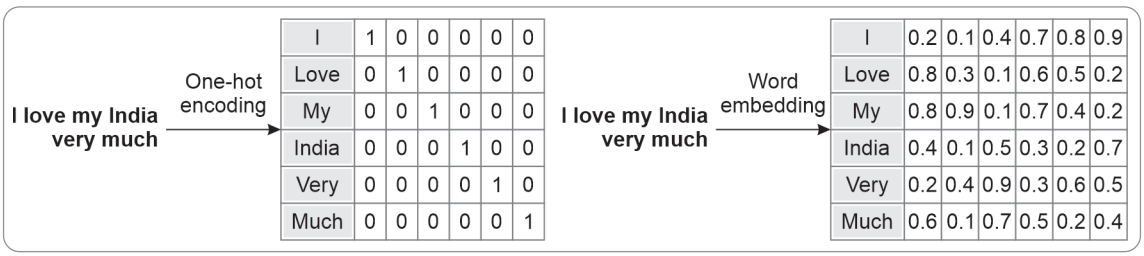
$$\text{cat} = [1, 0, 0],\ \text{dog} = [0, 1, 0],\ \text{bird} = [0, 0, 1].$$

This representation is straightforward and mathematically simple, which makes it useful for introductory understanding. However, it has important limitations. First, it is a *sparse representation* because almost all entries are zero, especially when the vocabulary is large. This leads to inefficient memory usage and computation. Second, one-hot vectors do not encode any semantic relationship between words. For instance, the vectors for 'cat' and 'dog' are just as far apart as the vectors for 'cat' and 'car', even though the first pair is much more semantically related. Hence, one-hot encoding identifies words, but it does not explain their meaning.

5.2.2 Dense Embeddings

To overcome the limitations of one-hot encoding, modern NLP systems use *dense embeddings*, commonly called *word embeddings*. Instead of assigning each word a sparse binary vector, embeddings map each word to a *low-dimensional continuous-valued vector*. These vectors are learned from data in such a way that words occurring in similar contexts tend to receive similar numerical representations.

As illustrated in Figure 5.2, one-hot encoding represents each word as a high-dimensional sparse identity vector, whereas word embedding transforms the same word into a compact dense vector of real numbers.



I	1	0	0	0	0	0
Love	0	1	0	0	0	0
My	0	0	1	0	0	0
India	0	0	0	1	0	0
Very	0	0	0	0	1	0
Much	0	0	0	0	0	1

I	0.2	0.1	0.4	0.7	0.8	0.9
Love	0.8	0.3	0.1	0.6	0.5	0.2
My	0.8	0.9	0.1	0.7	0.4	0.2
India	0.4	0.1	0.5	0.3	0.2	0.7
Very	0.2	0.4	0.9	0.3	0.6	0.5
Much	0.6	0.1	0.7	0.5	0.2	0.4

Fig. 5.2: Transformation from one-hot encoded word vectors to dense word embeddings

In an embedding space, the relative positions of words become meaningful. Words with similar meanings, related usage, or comparable contextual roles tend to lie closer to one another. Thus, the representation is no longer merely symbolic; it becomes semantically informative. This is one of the key reasons why embeddings are so powerful in neural language processing.

Example: Consider the sentence *'I Love My India Very Much'*. In a learned embedding space, words such as *Love*, *Very*, and *Much* tend to be located close to one another because they express similar semantic notions related to intensity and emotion. Similarly, words like *India* and other country names (not shown here) would occupy nearby regions in the embedding space, reflecting their conceptual similarity as geographical entities. Even functionally related words such as *My* and *I* may appear closer compared to unrelated words, as they frequently co-occur in similar linguistic contexts.

Thus, the embedding space captures both semantic and contextual relationships among words appearing in the sentence. Rather than treating each word as completely independent, the model learns meaningful geometric structures where related words are grouped together. This organization enables neural networks to better understand context, capture relationships between words, and generalize effectively to new sentences.

Important Concept

Embeddings transform discrete tokens into continuous vector spaces, enabling neural networks to capture semantic similarity, contextual relationships, and meaningful linguistic patterns.

5.3 SEQUENCE MODELING WITH NEURAL NETWORKS

In classical sequence modeling, the next element of a sequence is predicted using fixed probabilistic assumptions, such as a limited Markov context. Although such approaches are mathematically elegant, they are not sufficiently expressive for modeling complex patterns in language, speech, and temporal signals. Neural sequence models address this limitation by learning representations directly from data. Among the earliest and most influential neural architectures for sequential data are the *RNNs* and their improved variant, the *LSTM* network.

5.3.1 Recurrent Neural Networks

An RNN processes a sequence *step-by-step* and maintains an internal *hidden state* that acts as a memory of previous inputs. Unlike feedforward neural networks, which treat each input independently, an RNN explicitly models temporal dependence by passing information from one-time step to the next. This makes it suitable for tasks such as next-word prediction, language modeling, speech recognition, and sequence classification.

Let the input sequence be represented as:

$$X = (x_1, x_2, x_3, \ldots, x_t),$$

where x_t denotes the input at time step t. The hidden state at time t is denoted by h_t, and the corresponding output is denoted by y_t. The recurrent computation is expressed as:

$$h_t = \phi\,(Ux_t + Wh_{t-1} + b_h),$$
$$y_t = \psi(Vh_t + b_y),$$

where:

- x_t is the input vector at time step t.
- h_{t-1} is the hidden state from the previous time step.
- h_t is the current hidden state.
- y_t is the output at time step t.
- U is the input-to-hidden weight matrix.
- W is the hidden-to-hidden recurrent weight matrix.
- V is the hidden-to-output weight matrix.
- b_h and b_y are bias vectors.
- $\phi(\cdot)$ is a nonlinear activation function such as tanh or Rectified Linear Unit (ReLU).
- $\psi(\cdot)$ is an output activation function, often softmax for prediction tasks.

Thus, at each time step, the hidden state h_t combines the *current input* x_t with the *past memory* h_{t-1}. This recurrent mechanism allows the network to accumulate contextual information over time.

Figure 5.3 shows the structure of a *single recurrent cell*. The input x_t enters the cell through the weight matrix U, the previous hidden state influences the current computation through the recurrent connection W, and the resulting hidden state h_t produces the output y_t through the weight matrix V. This compact diagram captures the essential recurrence mechanism of an RNN.

Figure 5.4 presents the *unfolded form* of the same RNN across multiple time steps. Here, the repeated application of the same recurrent cell is shown explicitly for time steps 1 through T. The same parameter matrices U, W, and V are reused at every time step. This property is known as *parameter sharing across time*, and it is one of the defining features of recurrent networks.

In the single-cell diagram (Fig. 5.3), x_t denotes the current input, h_t denotes the hidden representation, and y_t denotes the output generated from that hidden representation. The arrow labeled U connects the input

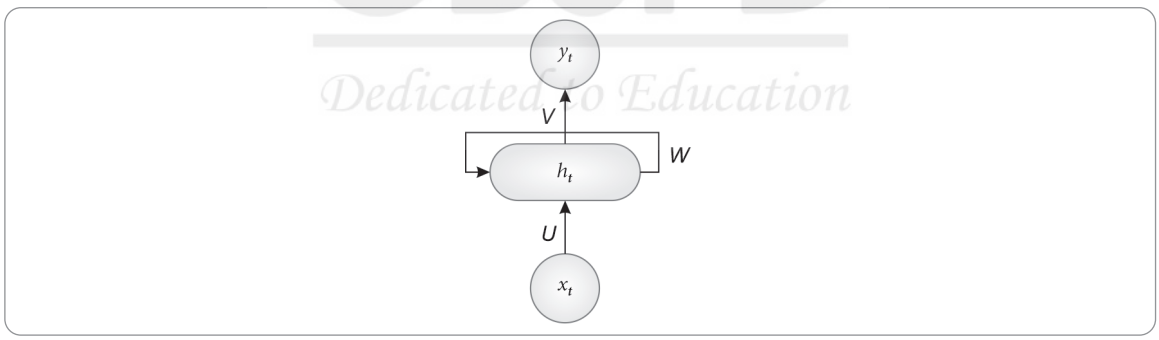


Fig. 5.3: Single recurrent neural network cell before unfolding in time. The hidden state h_t is updated recursively using the current input x_t through the input weight matrix U and the previous hidden state h_{t-1} through the recurrent weight matrix W. Formally, $h_t = \phi\,(U_{xt} + Wh_{t-1} + b_h)$. The updated hidden state then produces the output y_t *via* the output weight matrix V, i.e., $y_t = \psi(Vh_t + b_y)$. This compact representation highlights the recursive dependency of the hidden state across time steps

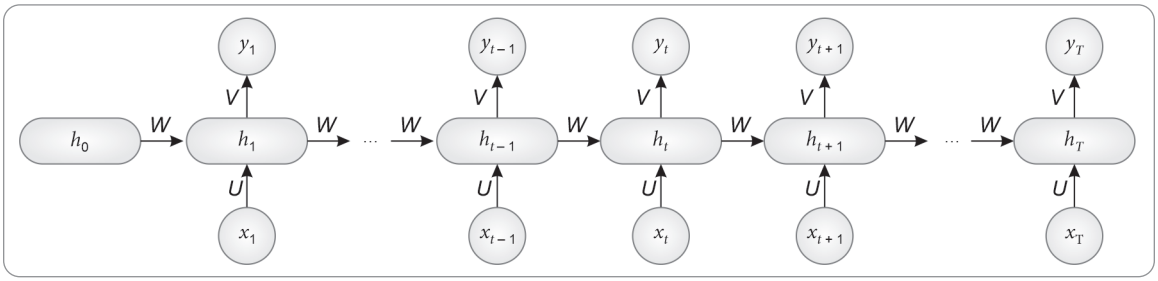


Fig. 5.4: Unrolled representation of a recurrent neural network from time step $t = 1$ to $t = T$: Each recurrent cell processes the current input x_t together with the previous hidden state h_{t-1} to update its internal memory h_t through the input and recurrent weight matrices U and W. The corresponding output y_t is generated via the output weight matrix V. All recurrent cells share the same parameter matrices (U, W, V), illustrating temporal weight sharing across the sequence. The initial hidden state h_0 is externally initialized, and the sequence evolves sequentially up to h_T, beyond which no further propagation occurs

x_t to the hidden unit, indicating the input-to-hidden transformation. The recurrent arrow labeled W indicates the hidden-to-hidden transition, through which information from the previous hidden state is transferred forward. The arrow labeled V maps the hidden state to the output. In the unfolded diagram, the sequence of hidden states $h_0, h_1, \ldots, h_T$ explicitly shows how the model evolves over time, while the outputs $y_1, y_2, \ldots, y_T$ are generated at successive steps from their corresponding hidden states.

Illustrative Example: Forward Computation in a Simple Recurrent Neural Network

To understand how an RNN processes sequences, let us consider a small language modeling example. The goal of the model is to predict the next word in a sequence based on previously seen words.

Vocabulary: Let the vocabulary be:

$$\mathscr{V} = \{\text{'rain', 'falls', 'softly'}\}.$$

We assign:

$$\omega_1 = \text{'rain'}, \omega_2 = \text{'falls'}, \omega_3 = \text{'softly'}.$$

Sequence: Consider the sequence:

$$\text{'rain'} \rightarrow \text{'falls'} \rightarrow \text{'softly'}.$$

Thus, the RNN performs computations for:

$$T = 3.$$

Input encoding: Each word is represented as a one-hot vector:

$$\mathbf{x}_1 = [1, 0, 0]^T, \mathbf{x}_2 = [0, 1, 0]^T, \mathbf{x}_3 = [0, 0, 1]^T.$$

Network parameters: We assume the following simple weights:

$$\mathbf{U} = \begin{bmatrix} 0.4 & -0.2 & 0.1 \\ 0.3 & 0.5 & -0.4 \end{bmatrix}, \mathbf{W} = \begin{bmatrix} 0.2 & 0.6 \\ -0.3 & 0.1 \end{bmatrix}, \mathbf{V} = \begin{bmatrix} 0.5 & 0.1 \\ -0.2 & 0.3 \\ 0.1 & 0.4 \end{bmatrix}.$$

Initial hidden state:

$$\mathbf{h}_0 = [0, 0]^T.$$

Activation functions:

$$\mathbf{h}_t = \tanh(\mathbf{U}\mathbf{x}_t + \mathbf{W}\mathbf{h}_{t-1}), \mathbf{y}_t = \text{Softmax}(\mathbf{V}\mathbf{h}_t).$$

Step 1. $t = 1$ **(Input: 'rain'):**

$$\mathbf{a}_1 = \mathbf{U}\mathbf{x}_1 = \begin{bmatrix} 0.4 \\ 0.3 \end{bmatrix}$$

$$\mathbf{h}_1 = \tanh(\mathbf{a}_1) = \begin{bmatrix} 0.3799 \\ 0.2913 \end{bmatrix}$$

$$\mathbf{o}_1 = \mathbf{V}\mathbf{h}_1 = \begin{bmatrix} 0.5\,(0.3799) + 0.1\ \ (0.2913) \\ -0.2\,(0.3799) + 0.3\,(0.2913) \\ 0.1\,(0.3799) + 0.4\,(0.2913) \end{bmatrix} = \begin{bmatrix} 0.2181 \\ 0.0344 \\ 0.1555 \end{bmatrix}$$

Softmax (approx):

$$\mathbf{y}_1 \approx [0.37, 0.30, 0.33].$$

Interpretation: The model slightly favors 'falls' as the next word.

Step 2. $t = 2$ **(Input: 'falls'):**

$$\mathbf{a}_2 = \mathbf{U}\mathbf{x}_2 + \mathbf{W}\mathbf{h}_1$$

$$\mathbf{U}\mathbf{x}_2 = \begin{bmatrix} -0.2 \\ 0.5 \end{bmatrix}, \mathbf{W}\mathbf{h}_1 = \begin{bmatrix} 0.2\,(0.3799) + 0.6\,(0.2913) \\ -0.3\,(0.3799) + 0.1\,(0.2913) \end{bmatrix} = \begin{bmatrix} 0.2500 \\ -0.0838 \end{bmatrix}$$

$$\mathbf{a}_2 = \begin{bmatrix} 0.3461 \\ -0.3760 \end{bmatrix}$$

$$\mathbf{h}_2 = \tanh(\mathbf{a}_2) = \begin{bmatrix} 0.0499 \\ 0.3935 \end{bmatrix}$$

$$\mathbf{o}_2 = \mathbf{V}\mathbf{h}_2 = \begin{bmatrix} 0.0793 \\ 0.1081 \\ 0.1624 \end{bmatrix}$$

$$\mathbf{y}_2 \approx [0.31, 0.32, 0.37].$$

Interpretation: The model now favors 'softly', consistent with the sequence.

Step 3. $t = 3$ **(Input: 'softly'):**

$$\mathbf{a}_3 = \mathbf{U}\mathbf{x}_3 + \mathbf{W}\mathbf{h}_2$$

$$\mathbf{U}\mathbf{x}_3 = \begin{bmatrix} 0.1 \\ -0.4 \end{bmatrix}, \mathbf{W}\mathbf{h}_2 = \begin{bmatrix} 0.2\,(0.0499) + 0.6\,(0.3935) \\ -0.3\,(0.0499) + 0.1\,(0.3935) \end{bmatrix} = \begin{bmatrix} 0.2461 \\ 0.0240 \end{bmatrix}$$

$$\mathbf{a}_3 = \begin{bmatrix} 0.3461 \\ -0.3760 \end{bmatrix}$$

$$\mathbf{h}_3 = \tanh(\mathbf{a}_3) = \begin{bmatrix} 0.3327 \\ -0.3594 \end{bmatrix}$$

$$\mathbf{o}_3 = \begin{bmatrix} 0.1300 \\ -0.1565 \\ -0.1100 \end{bmatrix}$$

$$\mathbf{y}_3 \approx [0.38, 0.30, 0.32].$$

Interpretation: The model cycles back and predicts 'rain' with highest probability.

Summary

t	Input	Predicted next word
1	rain	falls
2	falls	softly
3	softly	rain

Key Insight

At each time step, the hidden state $\boldsymbol{h}_t$ combines the current input and past memory. This allows the RNN to model sequential dependencies and predict the next element in a sequence.

Backpropagation Through Time

Training RNNs is performed using *Backpropagation Through Time (BPTT)*, which extends standard backpropagation to sequential data. In this approach, the RNN is *unrolled across time steps*, and the loss is computed at each step. The total loss is expressed as:

$$L = \sum_{t=1}^{T} L_t,$$

where L_t denotes the loss at time step t.

The key idea of BPTT is to compute gradients using the chain rule across time. For the hidden state, the gradient at an earlier time step k depends on all future time steps:

$$\frac{\partial L}{\partial h_k} = \sum_{t=k}^{T} \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_k}.$$

This indicates that each hidden state contributes not only to its immediate output but also influences subsequent predictions.

Since the hidden states are recursively defined as:

$$h_t = \phi\,(Ux_t + Wh_{t-1}),$$

the gradient propagation involves repeated multiplication across time:

$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} \left(W^T \cdot \phi'(a_i)\right).$$

If the norm of W and the derivative $\phi'(a_i)$ is less than 1, then:

$$\left\| \frac{\partial h_t}{\partial h_k} \right\| \to 0,$$

leading to the *vanishing gradient problem*. Conversely, if it exceeds 1, gradients may grow uncontrollably, causing the *exploding gradient problem*. This repeated multiplicative structure is the fundamental reason why standard RNNs struggle to learn long-range dependencies. Importantly, the same gradient propagation principle is used in LSTMs; however, LSTMs mitigate this issue through *gated additive memory updates*, which stabilize gradient flow over long sequences.

Limitation of Basic Recurrent Neural Networks

Due to the above gradient behavior, standard RNNs face significant difficulty in learning *long-range dependencies*. As gradients diminish during backpropagation, the contribution of earlier time steps becomes negligible, causing the network to rely primarily on recent inputs. This results in poor retention of distant contextual information and limits the model's effectiveness in tasks where earlier elements in the sequence are crucial for later predictions. Consequently, basic RNNs are often inadequate for modeling long sequences, motivating the development of improved architectures such as LSTMs.

5.3.2 Long Short-Term Memory Networks

The LSTM networks were introduced to overcome the memory limitations of basic RNNs. Unlike standard RNNs, which struggle to retain information over long sequences, LSTMs are designed to selectively remember and forget information through a structured internal mechanism. This is achieved using a *cell state* c_t, which acts as a long-term memory, along with a set of *gates* that regulate the flow of information.

At each time step t, the LSTM processes the current input x_t and the previous hidden state h_{t-1} to update both the hidden state h_t and the cell state c_t. The computations are governed by the following equations:

$$i_t = \sigma(U_i x_t + W_i h_{t-1} + b_i) \quad \text{(Input gate)}$$
$$f_t = \sigma(U_f x_t + W_f h_{t-1} + b_f) \quad \text{(Forget gate)}$$
$$o_t = \sigma(U_o x_t + W_o h_{t-1} + b_o) \quad \text{(Output gate)}$$
$$\tilde{c}_t = \tanh(U_c x_t + W_c h_{t-1} + b_c) \quad \text{(Candidate cell state)}$$

Cell and hidden state updates: After computing the gates and the candidate memory, the LSTM updates its internal states through the following equations:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t,$$
$$h_t = o_t \odot \tanh(c_t).$$

The cell state c_t represents the *long-term memory* of the network. It is updated by combining two components—the retained past memory $f_t \odot c_{t-1}$, controlled by the forget gate f_t, and the newly added information $i_t \odot \tilde{c}_t$, regulated by the input gate i_t. This formulation allows the LSTM to selectively preserve useful historical information while incorporating relevant new content.

The hidden state h_t acts as the *short-term, observable representation* of the cell. It is computed by applying a nonlinear transformation $\tanh(c_t)$ to the cell state and then modulating it through the output gate o_t. This ensures that only the relevant portion of the internal memory is exposed to the next time step or subsequent layers.

The operator $\odot$ denotes *element-wise multiplication* (Hadamard product), which allows each dimension of the memory vector to be independently controlled. This fine-grained control is crucial for learning complex temporal dependencies.

The activation functions used in LSTM play complementary roles. The $\sigma(\cdot)$ (sigmoid) function acts as a *gate controller*, producing values between 0 and 1 to decide how much information should be passed or suppressed. In contrast, the $\tanh(\cdot)$ function acts as a *content transformer*, shaping the information within the memory to lie in the range $[-1, 1]$. Together, these mechanisms enable the LSTM to effectively regulate information flow and maintain stable learning over long sequences.

The LSTM uses three gates to control memory flow:

1. The **forget gate** f_t determines how much of the previous memory c_{t-1} should be retained or discarded.
2. The **input gate** i_t controls how much new information $\tilde{c}_t$ should be added to the memory.
3. The **output gate** o_t decides how much of the internal memory should be exposed as the hidden state h_t.

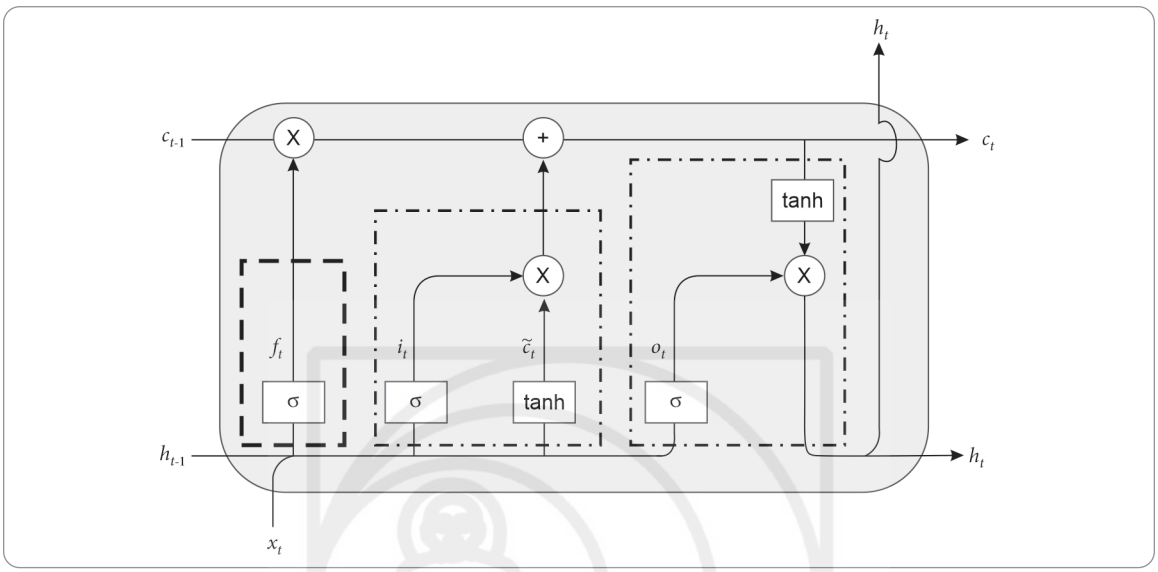


Fig. 5.5: Structure of a long short-term memory cell illustrating gated control of information flow: The forget gate f_t, input gate i_t, and output gate o_t regulate how information is retained, updated, and exposed through the cell state c_t and hidden state h_t, enabling effective modeling of long-term dependencies

Here, $\sigma(\cdot)$ denotes the sigmoid activation function, which outputs values between 0 and 1, acting as a soft gating mechanism. The function $\tanh(\cdot)$ produces values in the range $[-1, 1]$, allowing both positive and negative information flow. The symbol $\odot$ denotes element-wise multiplication.

Unlike standard RNNs, which repeatedly overwrite their hidden state, LSTMs maintain a *persistent cell state* that flows through time with controlled updates. This allows important information to be preserved over long sequences while irrelevant information is gradually forgotten.

As shown in Figure 5.5, the LSTM cell integrates gating mechanisms with a persistent memory pathway. The cell state c_t acts as a highway for information flow, while the gates dynamically control what to remember, update, and output at each time step. This design enables LSTMs to capture long-range dependencies that are difficult for standard RNNs to learn.

Why LSTMs are important: The main contribution of LSTMs lies in their ability to mitigate the vanishing gradient problem. By allowing information to flow through the cell state with controlled updates, LSTMs can preserve relevant context over longer ranges than standard RNNs. This makes them highly effective in language modeling, machine translation, speech processing, handwriting recognition, and many other sequential learning tasks.

Worked Example: LSTM Forward Computation for the Sequence 'I Love India'

To understand how an LSTM processes a sequence, let us consider the short sentence:

'I Love India'.

The goal is to see how the LSTM updates its *cell state* and *hidden state* at each time step while reading the words one after another.

Vocabulary and input representation: Assume the vocabulary is:

$\mathscr{V}$ = {'I', 'Love', 'India'}.

Using one-hot encoding, the input vectors are:

$$x_1 = \begin{bmatrix}1\\0\\0\end{bmatrix}, \qquad x_2 = \begin{bmatrix}0\\1\\0\end{bmatrix}, \qquad x_3 = \begin{bmatrix}0\\0\\1\end{bmatrix}.$$

Initial states: Before reading the sequence, we initialize the hidden state and cell state as:

$$h_0 = \begin{bmatrix}0\\0\end{bmatrix}, \qquad c_0 = \begin{bmatrix}0\\0\end{bmatrix}.$$

For simplicity, instead of computing the gate activations from weight matrices, we assume that the LSTM has already produced the following gate values at each step. These values are chosen only for illustration.

Step 1. Processing the word 'I': Assume the LSTM computes:

$$i_1 = \begin{bmatrix}0.7\\0.6\end{bmatrix}, \qquad f_1 = \begin{bmatrix}0.2\\0.3\end{bmatrix}, \qquad o_1 = \begin{bmatrix}0.8\\0.7\end{bmatrix},$$

and the candidate cell content is:

$$\tilde{c}_1 = \begin{bmatrix}0.5\\0.4\end{bmatrix}.$$

The cell-state update equation is:

$$c_1 = f_1 \odot c_0 + i_1 \odot \tilde{c}_1.$$

Since $c_0 = \begin{bmatrix}0\\0\end{bmatrix}$, we first compute:

$$f_1 \odot c_0 = \begin{bmatrix}0.2\\0.3\end{bmatrix} \odot \begin{bmatrix}0\\0\end{bmatrix} = \begin{bmatrix}0\\0\end{bmatrix}.$$

Next:

$$i_1 \odot \tilde{c}_1 = \begin{bmatrix}0.7\\0.6\end{bmatrix} \odot \begin{bmatrix}0.5\\0.4\end{bmatrix} = \begin{bmatrix}0.35\\0.24\end{bmatrix}.$$

Therefore,

$$c_1 = \begin{bmatrix}0\\0\end{bmatrix} + \begin{bmatrix}0.35\\0.24\end{bmatrix} = \begin{bmatrix}0.35\\0.24\end{bmatrix}.$$

Now compute the hidden state:

$$h_1 = o_1 \odot \tanh(c_1).$$

Using approximate values:

$$\tanh(0.35) \approx 0.336, \qquad \tanh(0.24) \approx 0.235,$$

so,

$$\tanh(c_1) \approx \begin{bmatrix}0.336\\0.235\end{bmatrix}.$$

Hence,

$$h_1 = \begin{bmatrix}0.8\\0.7\end{bmatrix} \odot \begin{bmatrix}0.336\\0.235\end{bmatrix} = \begin{bmatrix}0.2688\\0.1645\end{bmatrix}.$$

CBSD Dedicated for education

Thus, after reading 'I':

$$c_1 = \begin{bmatrix} 0.35 \\ 0.24 \end{bmatrix}, \quad h_1 = \begin{bmatrix} 0.2688 \\ 0.1645 \end{bmatrix}.$$

Interpretation: The LSTM starts building an internal representation of the sequence. Since this is the first word, the model mainly initializes its memory around the subject of the sentence.

Step 2. Processing the word 'Love': Assume the gates at the second time step are:

$$i_2 = \begin{bmatrix} 0.9 \\ 0.8 \end{bmatrix}, \quad f_2 = \begin{bmatrix} 0.5 \\ 0.4 \end{bmatrix}, \quad o_2 = \begin{bmatrix} 0.85 \\ 0.75 \end{bmatrix},$$

and the candidate memory is:

$$\tilde{c}_2 = \begin{bmatrix} 0.7 \\ 0.6 \end{bmatrix}.$$

Now update the cell state:

$$c_2 = f_2 \odot c_1 + i_2 \odot \tilde{c}_2.$$

First compute the retained memory:

$$f_2 \odot c_1 = \begin{bmatrix} 0.5 \\ 0.4 \end{bmatrix} \odot \begin{bmatrix} 0.35 \\ 0.24 \end{bmatrix} = \begin{bmatrix} 0.175 \\ 0.096 \end{bmatrix}.$$

Next compute the new information to be added:

$$i_2 \odot \tilde{c}_2 = \begin{bmatrix} 0.9 \\ 0.8 \end{bmatrix} \odot \begin{bmatrix} 0.7 \\ 0.6 \end{bmatrix} = \begin{bmatrix} 0.63 \\ 0.48 \end{bmatrix}.$$

Thus,

$$c_2 = \begin{bmatrix} 0.175 \\ 0.096 \end{bmatrix} + \begin{bmatrix} 0.63 \\ 0.48 \end{bmatrix} = \begin{bmatrix} 0.805 \\ 0.576 \end{bmatrix}.$$

Now compute:

$$h_2 = o_2 \odot \tanh(c_2).$$

Using approximate values:

$$\tanh(0.805) \approx 0.667, \qquad \tanh(0.576) \approx 0.520,$$

so,

$$\tanh(c_2) \approx \begin{bmatrix} 0.667 \\ 0.520 \end{bmatrix}.$$

Hence,

$$h_2 = \begin{bmatrix} 0.85 \\ 0.75 \end{bmatrix} \odot \begin{bmatrix} 0.667 \\ 0.520 \end{bmatrix} = \begin{bmatrix} 0.5670 \\ 0.3900 \end{bmatrix}.$$

Therefore, after reading 'Love':

$$c_2 = \begin{bmatrix} 0.805 \\ 0.576 \end{bmatrix}, \quad h_2 = \begin{bmatrix} 0.5670 \\ 0.3900 \end{bmatrix}.$$

CBSPD Dedicated to Education

Interpretation: The word 'Love' adds strong semantic content to the sequence. The cell state increases because the LSTM decides to retain some previous information and also add important new emotional content.

Step 3. Processing the word 'India': Assume:

$$i_3 = \begin{bmatrix} 0.6 \\ 0.5 \end{bmatrix}, \quad f_3 = \begin{bmatrix} 0.6 \\ 0.5 \end{bmatrix}, \quad o_3 = \begin{bmatrix} 0.9 \\ 0.85 \end{bmatrix},$$

and

$$\tilde{c}_3 = \begin{bmatrix} 0.6 \\ 0.7 \end{bmatrix}.$$

Now:

$$c_3 = f_3 \odot c_2 + i_3 \odot \tilde{c}_3.$$

First compute:

$$f_3 \odot c_2 = \begin{bmatrix} 0.6 \\ 0.5 \end{bmatrix} \odot \begin{bmatrix} 0.805 \\ 0.576 \end{bmatrix} = \begin{bmatrix} 0.483 \\ 0.288 \end{bmatrix}.$$

Next:

$$i_3 \odot \tilde{c}_3 = \begin{bmatrix} 0.8 \\ 0.9 \end{bmatrix} \odot \begin{bmatrix} 0.6 \\ 0.7 \end{bmatrix} = \begin{bmatrix} 0.48 \\ 0.63 \end{bmatrix}.$$

Thus,

$$c_3 = \begin{bmatrix} 0.483 \\ 0.288 \end{bmatrix} + \begin{bmatrix} 0.48 \\ 0.63 \end{bmatrix} = \begin{bmatrix} 0.963 \\ 0.918 \end{bmatrix}.$$

Now compute the hidden state:

$$h_3 = o_3 \odot \tanh(c_3).$$

Using approximate values:

$$\tanh(0.963) \approx 0.746, \qquad \tanh(0.918) \approx 0.725,$$

so,

$$\tanh(c_3) \approx \begin{bmatrix} 0.746 \\ 0.725 \end{bmatrix}.$$

Hence,

$$h_3 = \begin{bmatrix} 0.9 \\ 0.85 \end{bmatrix} \odot \begin{bmatrix} 0.746 \\ 0.725 \end{bmatrix} = \begin{bmatrix} 0.6714 \\ 0.6163 \end{bmatrix}.$$

Therefore, after reading 'India':

$$c_3 = \begin{bmatrix} 0.963 \\ 0.918 \end{bmatrix}, \quad h_3 = \begin{bmatrix} 0.6714 \\ 0.6163 \end{bmatrix}.$$

Interpretation: At this stage, the LSTM has combined the information from all three words and formed a strong internal representation of the complete phrase 'I Love India'.

CBSPD Dedicated to Education

Summary of State Evolution

t	Input word	c_t	h_t
1	I	$\begin{bmatrix}0.35\\0.24\end{bmatrix}$	$\begin{bmatrix}0.2688\\0.1645\end{bmatrix}$
2	Love	$\begin{bmatrix}0.805\\0.576\end{bmatrix}$	$\begin{bmatrix}0.5670\\0.3900\end{bmatrix}$
3	India	$\begin{bmatrix}0.963\\0.918\end{bmatrix}$	$\begin{bmatrix}0.6714\\0.6163\end{bmatrix}$

Gradient Flow in Long Short-Term Memory Networks

The LSTM networks are specifically designed to improve gradient propagation across time. Similar to RNNs, training in LSTMs also uses *BPTT*, where the network is unrolled and gradients are computed across time steps. However, unlike standard RNNs, the key difference lies in the *cell state dynamics*, which follow an additive update rule:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t.$$

When computing gradients with respect to an earlier cell state c_k, we obtain:

$$\frac{\partial c_t}{\partial c_k} = \prod_{i=k+1}^{t} f_i,$$

where f_i are the forget gate values lying in the range [0, 1].

Since the forget gate f_t can be close to 1, the product:

$$\prod_{i=k+1}^{t} f_i \approx 1,$$

allowing gradients to flow backward through time without vanishing rapidly. This is in contrast to standard RNNs, where gradients involve repeated multiplication by weight matrices and nonlinear derivatives.

Furthermore, the hidden state is computed as:

$$h_t = o_t \odot \tanh(c_t),$$

which separates the *internal memory* c_t from the *exposed output* h_t. This separation ensures that long-term information is preserved in the cell state, while the output gate o_t controls how much of it is revealed at each time step.

Thus, the additive structure of the cell state, combined with gated control, enables LSTMs to maintain stable gradients and effectively learn long-range dependencies, overcoming the limitations of basic RNNs.

Limitations of Long Short-Term Memory Networks

Although LSTMs significantly improve the ability to model long-range dependencies compared to basic RNNs, they still have several practical limitations. The architecture is *computationally expensive*, as it involves multiple gates and parameter matrices, leading to increased training time and memory usage. LSTMs also require careful *hyperparameter tuning* and large amounts of training data to achieve good generalization. Despite mitigating vanishing gradients, they may still struggle with *very long sequences* where information needs to be retained over hundreds or thousands of time steps. Additionally, their inherently *sequential computation* limits parallelization, making them slower to train compared to modern architectures such as transformers. Finally, the internal gating mechanisms, while powerful, can reduce interpretability, making it difficult to understand how specific decisions are made within the network.

5.4 ATTENTION MECHANISM AND CONTEXT MODELING

In earlier sequence models such as RNNs and LSTMs, information is compressed into a fixed-size hidden state. While effective for short sequences, this compression often leads to the loss of important contextual information when dealing with longer inputs. The *attention mechanism* addresses this limitation by allowing the model to dynamically access and weigh different parts of the input sequence during decoding, rather than relying on a single fixed summary.

Motivation: The central idea behind attention is that not all elements of an input sequence contribute equally to generating an output. The relevance of each input token depends on the current decoding context. Instead of using only the final encoder state, attention computes a *context vector* as a weighted combination of all encoder hidden states:

$$c_t = \sum_{j=1}^{T} \alpha_{t,j} h_j,$$

where:

- h_j represents the hidden state of the encoder at position j.
- $\alpha_{t,j}$ denotes the attention weight assigned to h_j at decoding step t.
- $\sum_{j=1}^{T} \alpha_{t,j} = 1.$

Here, $\alpha_{t,j}$ quantifies how much attention the model pays to each hidden state h_j when generating the output at time t. Thus, the context vector c_t becomes a *linear combination of encoder hidden states*, dynamically adapted for each decoding step.

As illustrated in Figure 5.6, the encoder processes the input sequence $x_1, x_2, \ldots, x_T$ and produces a sequence of hidden states $h_1, h_2, \ldots, h_T$. During decoding, the previous decoder state s_{t-1} interacts with these encoder states to compute attention weights $\alpha_{t,j}$. These weights are then used to form the context vector c_t, which is fed into the decoder along with s_{t-1} to produce the next state s_t.

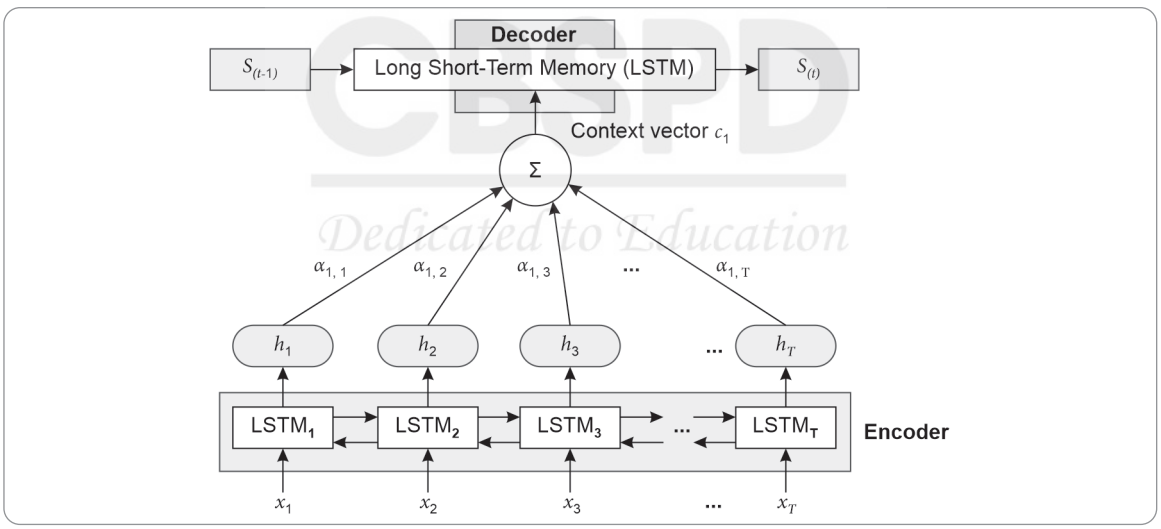


Fig. 5.6: Attention-based encoder–decoder architecture using Long Short-Term Memory: The encoder generates hidden states $h_1, h_2, \ldots, h_T$, and the decoder computes a context vector $c_t = \sum_{j=1}^{T} \alpha_{t,j} h_j$ as a weighted combination of these states

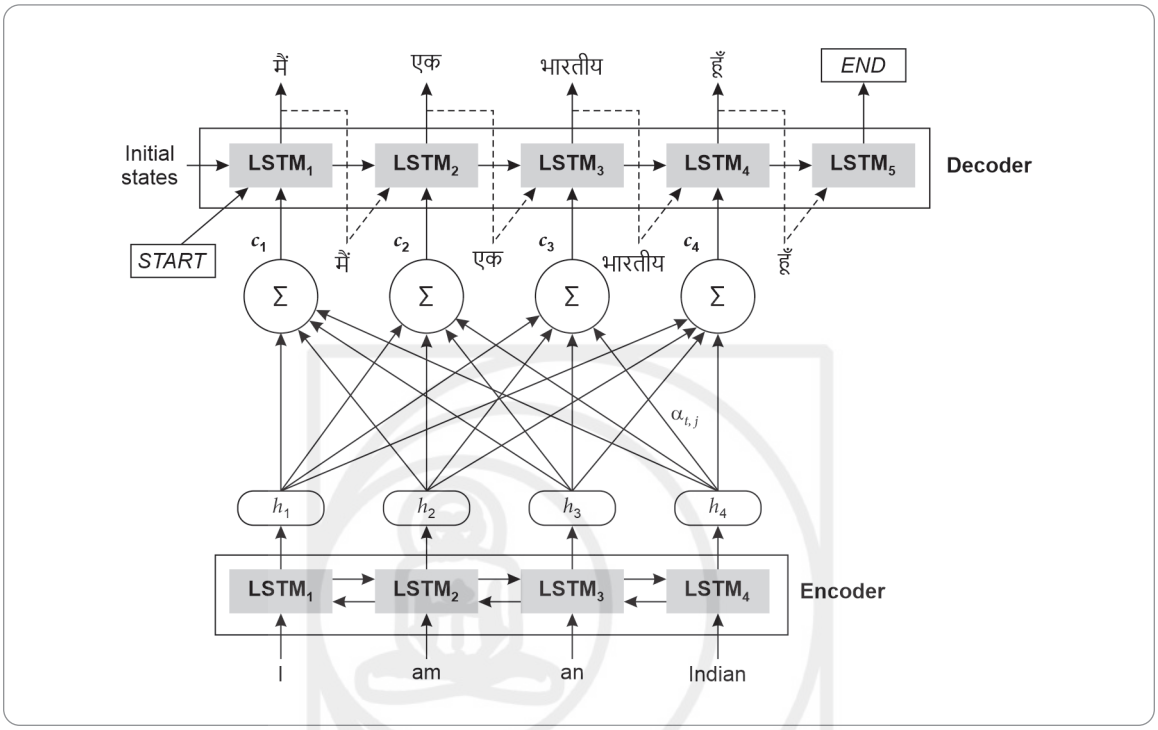


Fig. 5.7: Attention-based sequence-to-sequence model using a bidirectional long short-term memory network for English-to-Hindi translation: The encoder processes the input sequence in both forward and backward directions to produce context-rich hidden states $h_1, h_2, \ldots, h_T$. At each decoding step, the attention mechanism computes weights $\alpha_{t,j}$ over all encoder states to form the context vector $c_t = \sum_{j=1}^{T} \alpha_{t,j} h_j$. This context vector is combined with the previous decoder state to generate the next target word, enabling dynamic alignment between source and target sequences

Abbreviation: LSTM, Long Short-Term Memory

Example: Attention-based bidirectional long short-term memory network for translation

Figure 5.7 illustrates an attention-based encoder–decoder architecture using a *bidirectional LSTM* for sequence-to-sequence translation, specifically from English to Hindi.

In this architecture, the encoder consists of a *bidirectional LSTM*, which processes the input sequence in both forward and backward directions. This allows each hidden state h_j to capture information from both past and future context, leading to richer representations compared to unidirectional models.

Let the input sentence be:

'I am an Indian'.

The encoder produces hidden states:

$$h_1, h_2, h_3, h_4,$$

each encoding contextual information from the entire sequence.

At each decoding step t, the model computes attention weights $\alpha_{t,j}$ over all encoder hidden states:

$$c_t = \sum_{j=1}^{T} \alpha_{t,j} h_j.$$

5.5 TRANSFORMER ARCHITECTURE

The transformer architecture represents a major shift in sequence modeling by eliminating recurrence entirely and relying solely on attention mechanisms. Instead of processing tokens sequentially, transformers process all tokens in parallel, enabling efficient computation and improved modeling of long-range dependencies.

5.5.1 Advantages

Transformers offer several key advantages over RNN-based models:

- **Parallel computation:** All tokens are processed simultaneously, leading to faster training.
- **Better long-range dependency modeling:** Direct connections between all tokens avoid information bottlenecks.
- **Scalability:** Easily extended to large models and datasets, forming the basis of modern LLMs.

5.5.2 Self-Attention

The core component of the transformer architecture is the *self-attention mechanism*, which enables each token in a sequence to interact directly with every other token. Unlike recurrent models that process tokens sequentially, self-attention computes *pairwise relationships* across all tokens simultaneously, allowing the model to capture both local and global dependencies efficiently.

Let the input sequence be represented by an embedding matrix:

$$X \in \mathbb{R}^{T \times d},$$

where T is the sequence length and d is the embedding dimension.
Self-attention projects the input embeddings into three distinct spaces:

$$Q = XW_q, \qquad K = XW_k, \qquad V = XW_v,$$

where:

- Q (queries) represent what each token is looking for.
- K (keys) represent what each token offers.
- V (values) contain the actual information to be aggregated.
- $W_q, W_k, W_v \in \mathbb{R}^{d \times d_k}$ are learnable weight matrices.

The attention scores are computed using the dot product between queries and keys:

$$\text{Score}(Q, K) = QK^{\top}.$$

These scores are scaled to prevent large values:

$$\frac{QK^{\top}}{\sqrt{d_k}}.$$

Applying the softmax function yields the *attention weight matrix*:

$$A = \text{softmax}\left(\frac{QK^{\top}}{\sqrt{d_k}}\right),$$

where each row of A sums to 1, representing the distribution of attention weights for a given token.
Finally, the output of self-attention is obtained as:

$$\text{Attention}(Q, K, V) = AV.$$

As shown in Figure 5.8, each token (e.g., 'I', 'Love', 'India') is first converted into an embedding vector. These embeddings are then linearly transformed into queries, keys, and values. The interaction between queries and keys produces an *attention matrix*, which determines how much each token should

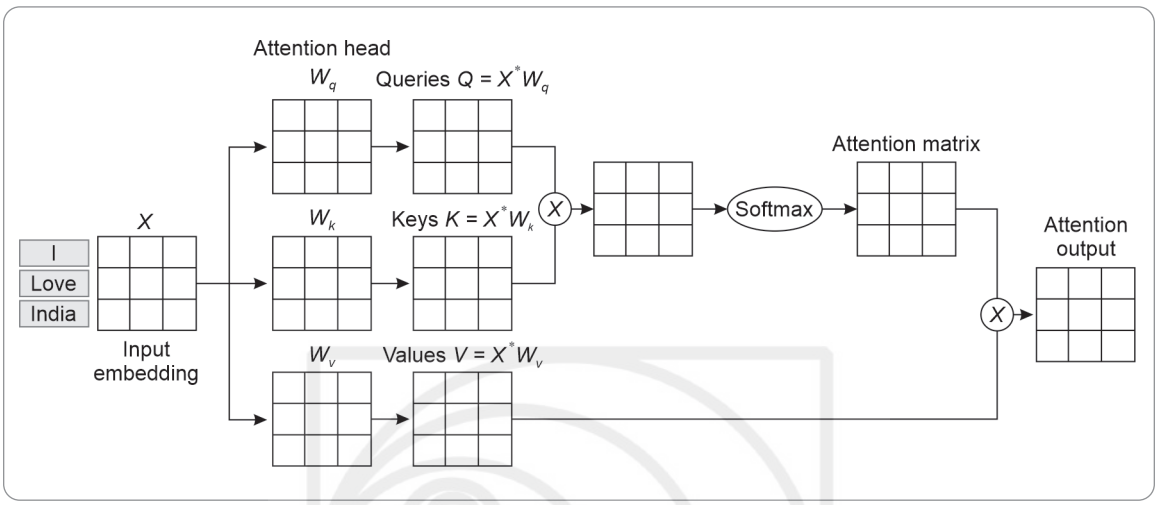


Fig. 5.8: Self-attention mechanism illustrating the transformation of input embeddings X into queries Q, keys K, and values V using learned projection matrices W_q, W_k, W_v. The attention matrix is computed via scaled dot-product followed by softmax, and the final output is obtained by weighting the value vectors accordingly

attend to others. This matrix is then used to compute a weighted combination of value vectors, resulting in context-aware representations.

Example: Consider the sentence:

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

To correctly interpret the pronoun *'it'*, the model must identify its referent. Through self-attention, the token 'it' computes similarity scores with all other tokens and assigns higher attention weights to semantically relevant words such as *animal*. As a result, the model correctly resolves the reference by focusing on the appropriate context.

Key Insight

Self-attention enables each token to dynamically aggregate information from the entire sequence, making it possible to capture long-range dependencies without relying on sequential processing. This forms the foundation of modern transformer-based architectures.

CBSPD Dedicated to Education

5.5.3 Positional Encoding

Unlike RNNs, transformers do not inherently process tokens in sequence order. Therefore, *positional encoding* is introduced to inject information about the position of each token in the sequence.

A common formulation uses sinusoidal functions:

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d}}\right), \quad PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d}}\right),$$

where pos denotes the position and i is the dimension index.

Worked Example: Positional Encoding for 'I love India very much'

Consider the sentence:

'I love India very much'.

This sentence has five tokens:

$$(x_1, x_2, x_3, x_4, x_5) = (\text{I, love, India, very, much}).$$

We assume embedding dimension $d = 4$. The positional encoding is defined as:

$$PE(p, 2i) = \sin\left(\frac{p}{10000^{2i/d}}\right). \; PE(p, 2i+1) = \cos\left(\frac{p}{10000^{2i/d}}\right).$$

Since $d = 4$, we consider $i = 0, 1$.

Position 1. (Token: I):

$$PE(1) = \left[\sin\left(\frac{p}{10000^{2i/d}}\right), \cos\left(\frac{p}{10000^{2i/d}}\right), \sin\left(\frac{p}{10000^{2i/d}}\right), \cos\left(\frac{p}{10000^{2i/d}}\right)\right]$$

$$= \left[\sin\left(\frac{1}{10000^{2(0)/4}}\right), \cos\left(\frac{1}{10000^{2(0)/4}}\right), \sin\left(\frac{1}{10000^{2(1)/4}}\right), \cos\left(\frac{1}{10000^{2(1)/4}}\right)\right]$$

$$= \left[\sin(1), \cos(1), \sin\left(\frac{1}{100}\right), \cos\left(\frac{1}{100}\right)\right]$$

$$\approx [0.84, 0.54, 0.01, 0.9999].$$

Position 2. (Token: love):

$$PE(2) = \left[\sin\left(\frac{p}{10000^{2i/d}}\right), \cos\left(\frac{p}{10000^{2i/d}}\right), \sin\left(\frac{p}{10000^{2i/d}}\right), \cos\left(\frac{p}{10000^{2i/d}}\right)\right]$$

$$= \left[\sin\left(\frac{2}{10000^{2(0)/4}}\right), \cos\left(\frac{2}{10000^{2(0)/4}}\right), \sin\left(\frac{2}{10000^{2(1)/4}}\right), \cos\left(\frac{2}{10000^{2(1)/4}}\right)\right]$$

$$= \left[\sin(2), \cos(2), \sin\left(\frac{2}{100}\right), \cos\left(\frac{2}{100}\right)\right]$$

$$\approx [0.91, -0.42, 0.02, 0.9998].$$

Position 3. (Token: India):

$$PE(3) = \left[\sin\left(\frac{p}{10000^{2i/d}}\right), \cos\left(\frac{p}{10000^{2i/d}}\right), \sin\left(\frac{p}{10000^{2i/d}}\right), \cos\left(\frac{p}{10000^{2i/d}}\right)\right]$$

$$= \left[\sin\left(\frac{3}{10000^{2(0)/4}}\right), \cos\left(\frac{3}{10000^{2(0)/4}}\right), \sin\left(\frac{3}{10000^{2(1)/4}}\right), \cos\left(\frac{3}{10000^{2(1)/4}}\right)\right]$$

$$= \left[\sin(3), \cos(3), \sin\left(\frac{3}{100}\right), \cos\left(\frac{3}{100}\right)\right]$$

$$\approx [0.14, -0.99, 0.03, 0.9996].$$

BSPD Dedicated to Education

Position 4. (Token: very):

$$PE\,(4) = \left[\sin\left(\frac{p}{10000^{2i/d}}\right), \cos\left(\frac{p}{10000^{2i/d}}\right), \sin\left(\frac{p}{10000^{2i/d}}\right), \cos\left(\frac{p}{10000^{2i/d}}\right)\right]$$

$$= \left[\sin\left(\frac{4}{10000^{2(0)/4}}\right), \cos\left(\frac{4}{10000^{2(0)/4}}\right), \sin\left(\frac{4}{10000^{2(1)/4}}\right), \cos\left(\frac{4}{10000^{2(1)/4}}\right)\right]$$

$$= \left[\sin(4), \cos(4), \sin\left(\frac{4}{100}\right), \cos\left(\frac{4}{100}\right)\right]$$

$$\approx [-0.76, -0.65, 0.04, 0.9992].$$

Position 5. (Token: much):

$$PE\,(5) = \left[\sin\left(\frac{p}{10000^{2i/d}}\right), \cos\left(\frac{p}{10000^{2i/d}}\right), \sin\left(\frac{p}{10000^{2i/d}}\right), \cos\left(\frac{p}{10000^{2i/d}}\right)\right]$$

$$= \left[\sin\left(\frac{5}{10000^{2(0)/4}}\right), \cos\left(\frac{5}{10000^{2(0)/4}}\right), \sin\left(\frac{5}{10000^{2(1)/4}}\right), \cos\left(\frac{5}{10000^{2(1)/4}}\right)\right]$$

$$= \left[\sin(5), \cos(5), \sin\left(\frac{5}{100}\right), \cos\left(\frac{5}{100}\right)\right]$$

$$\approx [-0.96, 0.28, 0.05, 0.9987].$$

Final step:

$$z_t = \text{Embedding}(x_t) + PE(t)$$

Using the computed values, the positional encoding matrix is:

$$PE = \begin{bmatrix} 0.84 & 0.54 & 0.01 & 0.9999 \\ 0.91 & -0.42 & 0.02 & 0.9998 \\ 0.14 & -0.99 & 0.03 & 0.9996 \\ -0.76 & -0.65 & 0.04 & 0.9992 \\ -0.96 & 0.28 & 0.05 & 0.9987 \end{bmatrix} \in \mathbb{R}^{5\times 4}$$

Each row corresponds to $PE(t)$ for token position $t = 1, \ldots, 5$.
Assume the embedding vectors for each word are:

$$X = \begin{bmatrix} 0.20 & 0.10 & 0.50 & 0.30 \\ 0.60 & 0.80 & 0.20 & 0.10 \\ 0.90 & 0.70 & 0.40 & 0.60 \\ 0.30 & 0.50 & 0.80 & 0.20 \\ 0.40 & 0.60 & 0.30 & 0.70 \end{bmatrix} \in \mathbb{R}^{5\times 4}$$

The final input to the transformer is obtained by element-wise addition:

$$Z = X + PE$$

$$Z = \begin{bmatrix} 1.04 & 0.64 & 0.51 & 1.2999 \\ 1.51 & 0.38 & 0.22 & 1.0998 \\ 1.04 & -0.29 & 0.43 & 1.5996 \\ -0.46 & -0.15 & 0.84 & 1.1992 \\ -0.56 & 0.88 & 0.35 & 1.6987 \end{bmatrix}$$

5.5.4 Encoder and Decoder Structures

Key Insight

Each positional encoding component is first defined using the general formula in terms of *p*, *i*,*d*, then instantiated with actual values, and finally evaluated numerically, ensuring both theoretical clarity and computational understanding.

The transformer architecture is composed of two primary components—the *encoder* and the *decoder*—which work together to transform an input sequence into an output sequence. Unlike recurrent architectures, both components operate using attention mechanisms and fully connected layers, enabling efficient parallel computation.

As illustrated in Figure 5.9, both encoder and decoder are composed of multiple identical layers stacked N times, allowing the model to learn hierarchical and progressively richer representations. Each encoder layer consists of multihead self-attention and feedforward networks, while each decoder layer includes masked self-attention, encoder-decoder attention, and feedforward layers, along with residual connections and normalization.

Encoder Structure

The *encoder* processes the input sequence X and produces a sequence of contextual representations:

$$Z = \text{Encoder}(X).$$

Each encoder layer consists of two main sublayers:

1. **Multihead self-attention:** Each token attends to all other tokens in the input sequence, capturing global dependencies.
2. **Feedforward neural network:** A position-wise fully connected network applied independently to each token:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2.$$

Each sublayer is followed by:

- **Residual connection:** Adds the input of the sublayer to its output
- **Layer normalization:** Stabilizes training and improves convergence

Thus, each encoder layer can be expressed as:

$$\text{Output} = \text{Layernorm}(x + \text{sublayer}(x)).$$

Decoder Structure

The *decoder* generates the output sequence one token at a time using both its own previous outputs and the encoder representations.

Each decoder layer consists of three sublayers:

1. **Masked multihead self-attention:** Prevents the decoder from accessing future tokens by applying a mask, ensuring autoregressive generation

CBSPD Dedicated to Education

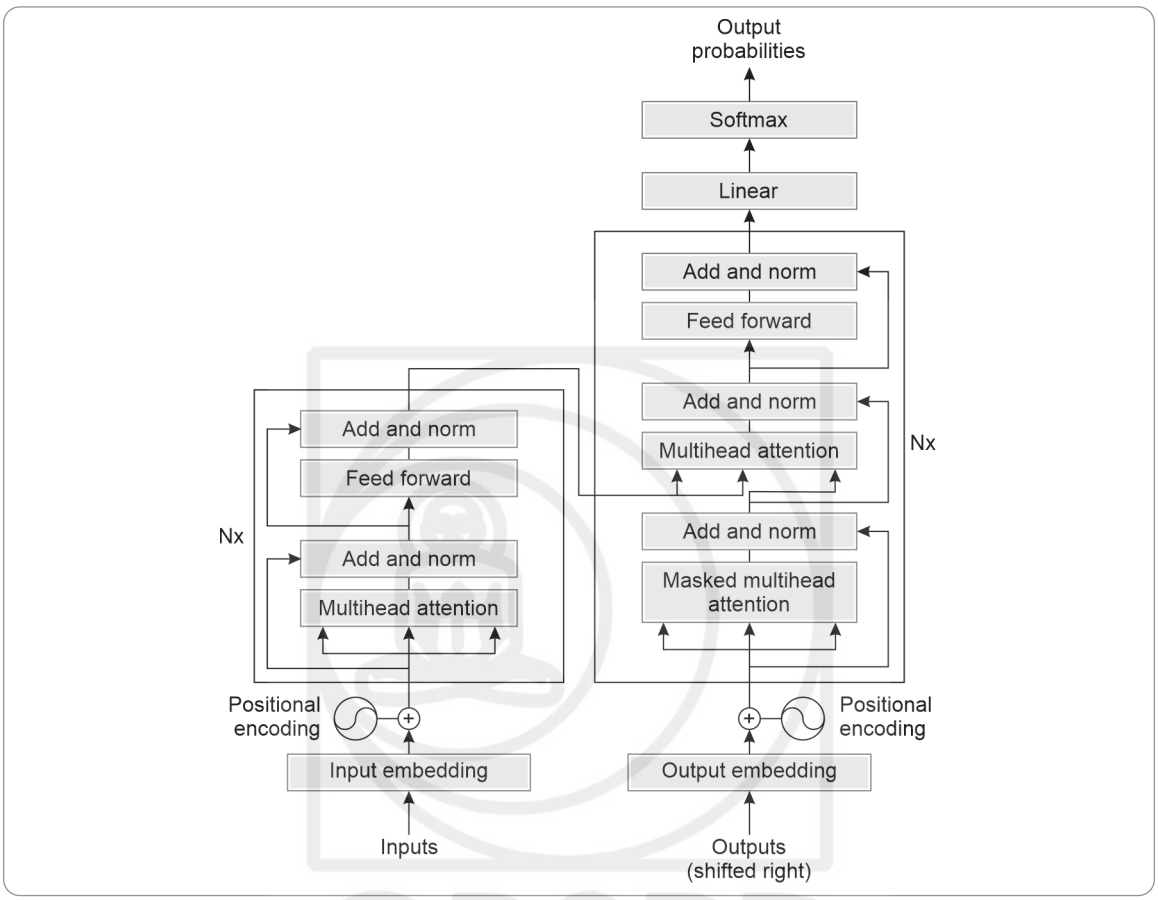


Fig. 5.9: Transformer architecture showing stacked encoder and decoder layers

2. **Encoder–decoder attention:** Allows the decoder to attend to the encoder output Z, aligning input and output sequences
3. **Feedforward neural network: Same structure as in the encoder.**

The decoder operates on shifted outputs:

$$Y = (y_1, y_2, \ldots, y_T),$$

where each prediction depends only on previously generated tokens.

Input and Output Processing

Before entering the encoder and decoder, tokens are transformed as:

$$Z_0 = X + PE,$$

where:

- X is the embedding matrix.
- PE is the positional encoding.

At the output stage, the decoder representation is passed through:

$$\text{Output probabilities} = \text{softmax}(ZW_o),$$

which produces a probability distribution over the vocabulary.

The encoder builds a deep contextual understanding of the input sequence, while the decoder uses this information along with previously generated tokens to produce the output. The interaction between encoder and decoder through attention enables precise alignment between input and output elements.

5.6 LARGE LANGUAGE MODELS

The LLMs extend the idea of pretraining to a much larger scale by training deep neural networks on massive text corpora. In practice, most modern LLMs are based on the transformer architecture and use self-attention to model relationships among tokens across long contexts. Their objective is typically next-token prediction, so that for a sequence $X = (x_1, x_2, \ldots, x_T)$, the model estimates:

$$P(X) = \prod_{t=1}^{T} P(x_t \mid x_{<t}) .$$

By learning this distribution over very large datasets, LLMs acquire broad linguistic, semantic, and task-general capabilities.

Characteristics: The LLMs are characterized by three major properties. First, they operate at very large scale, often using millions, billions, or even larger effective parameter counts, which gives them high representational capacity. Second, they perform *context-aware generation*, meaning that the output depends strongly on the prompt and the preceding textual context. Third, many of them exhibit *zero-shot, one-shot, or few-shot* behavior, where they can perform a new task from instructions or a small number of examples without full retraining. These properties make LLMs useful as general-purpose language systems rather than narrow task-specific models.

From an ecosystem perspective, it is useful to distinguish between *closed-source (proprietary)* models and *open-weight* models. As of 2026, well-known proprietary systems include *ChatGPT* and the *GPT-4.1* family from OpenAI, *Claude* models such as *Claude Sonnet 4.6* from Anthropic, and Google's *Gemini* family. On the open side, the field is strongly shaped by open-weight model families such as *Llama, Mistral, Qwen, DeepSeek, and NVIDIA Nemotron*. In modern literature and practice, these open-weight systems are often informally called 'open-source LLMs', even though the exact degree of openness may differ across model families, licenses, training recipes, and data disclosures.

Examples: Examples of proprietary LLM systems include *ChatGPT*, which offers conversational access to OpenAI models, and advanced assistant systems such as *Claude Sonnet 4.6*. These models are widely used for question answering, writing support, coding help, long-context reasoning, and knowledge work. Examples of open-weight systems include *Qwen* models, *Mistral Large 3, Llama* models, and *DeepSeek-R1*, which have become important for research, customization, self-hosting, and enterprise deployment where local control is needed.

In practical terms, LLMs can answer factual or conceptual questions, generate essays and reports, summarize documents, translate between languages, write or explain code, and assist with reasoning-oriented workflows. Their usefulness comes from the fact that a single pretrained model can often be adapted to many tasks merely by changing the prompt or by performing lightweight task-specific fine-tuning.

Important Observation

Large language models act as highly general language engines—the same model may support dialogue, summarization, coding, translation, and reasoning-style prompting. However, the quality of their outputs depends strongly on training data, alignment strategy, inference settings, and prompt design.

5.7 CAPABILITIES, LIMITATIONS, AND ENGINEERING IMPLICATIONS

5.7.1 Capabilities

Modern LLMs demonstrate strong capabilities in *natural language understanding, text generation, translation, summarization, code assistance*, and increasingly *multimodal processing* when language is combined with image, audio, or other modalities. Many systems also support instruction following, tool use, retrieval-augmented generation, and agent-like task execution when placed within larger software pipelines. These abilities arise from large-scale pretraining, attention-based contextual modeling, and post-training procedures such as instruction tuning and preference optimization.

5.7.2 Limitations

Despite their power, LLMs have important limitations, including *hallucinations* (generation of plausible but incorrect outputs), *bias in training data, lack of factual grounding, overconfidence in responses, context-length constraints, prompt sensitivity, vulnerability to prompt injection attacks, limited reasoning consistency, data privacy concerns, and high computational and energy cost*. Formally, hallucination can be viewed as a mismatch between the model-selected response:

$$\hat{y} = \arg\max_{y} P(y \mid x)$$

and the true or grounded answer required by the task, especially when external knowledge or verification mechanisms are absent.

5.7.3 Engineering Implications

Because of these capabilities and limitations, real-world deployment of LLMs is fundamentally an engineering problem as much as a modeling problem. Efficient deployment often requires *quantization, pruning, distillation*, batching strategies, memory-aware serving, and optimized inference stacks. Open-weight models are especially important here because they can be compressed, adapted, and deployed locally on private infrastructure. At the same time, developers must design systems that address latency, throughput, privacy, cost, safety, and reliability. Thus, in production environments, the best model is not always the largest one; rather, it is the one that achieves the best trade-off among accuracy, speed, memory footprint, and operational constraints.

Engineering Insight

Real-world large language model systems must balance intelligence with efficiency. In practice, system designers must jointly consider model quality, latency, memory usage, energy demand, deployment platform, safety, and maintainability.

STUDENT ASSIGNMENT

LONG ANSWER QUESTIONS

Answer the following questions with clear explanations, equations, and diagrams wherever appropriate.

1. Derive the chain rule decomposition for sequence probability and explain its significance in sequence modeling.

2. Compare n-gram models and hidden Markov models in terms of structure, assumptions, and limitations.
3. Explain the architecture and working of a Recurrent Neural Network (RNN) with suitable equations and diagrams.
4. Describe the Long Short-term Memory (LSTM) architecture in detail. Explain how gating mechanisms help overcome vanishing gradients.
5. Explain the attention mechanism mathematically and illustrate how context vectors are computed.
6. Derive the self-attention mechanism used in transformers and explain the role of query, key, and value matrices.
7. Explain positional encoding with its mathematical formulation and significance.
8. Describe the transformer encoder–decoder architecture with diagrams.
9. Discuss pretrained models and explain how fine-tuning works.
10. Critically analyze the capabilities and limitations of Large Language Models in real-world systems.

SHORT ANSWER QUESTIONS

Answer each question briefly and precisely.

1. What is sequence modeling? Give two real-world examples.
2. Write about the Markov assumption used in n-gram models.
3. What is the limitation of one-hot encoding?
4. Define word embeddings and state their advantage.
5. What is the role of hidden state in an RNN?
6. Why do RNNs suffer from the vanishing gradient problem?
7. What are the main components of an LSTM cell?
8. What is the intuition behind the attention mechanism?
9. Why do transformers not require recurrence?
10. What is positional encoding and why is it required?

TRUE/FALSE QUESTIONS

State whether each statement is True or False.

1. n-gram models can capture long-range dependencies effectively.
2. One-hot encoding captures semantic similarity between words.
3. RNNs process sequences using shared parameters across time steps.
4. LSTMs use gating mechanisms to control information flow.
5. Transformers rely on recurrence to process sequences.

FILL IN THE BLANKS

Fill in the blanks with the most appropriate term or expression.

1. The probability of a sequence is decomposed using the _____rule.
2. In an RNN, the hidden state at time t depends on _____ and the current input.
3. LSTM uses _____ gates to regulate memory.
4. The attention mechanism computes a _____combination of hidden states.
5. Transformers use _____encoding to capture sequence order.

MULTIPLE CHOICE QUESTIONS

Choose the most appropriate answer.

1. **Which model assumes independence between words?**
   a. Bigram b. Trigram
   c. Unigram d. LSTM
2. **What is the main purpose of the forget gate in LSTM?**
   a. Add new information b. Remove irrelevant information
   c. Generate output d. Normalize data
3. **Which component enables transformers to capture global dependencies?**
   a. Convolution b. Recurrence
   c. Self-attention d. Pooling
4. **What is the role of positional encoding?**
   a. Reduce model size b. Encode sequence order
   c. Improve optimization d. Normalize embeddings
5. **Which of the following is a limitation of LLMs?**
   a. Perfect reasoning b. No computational cost
   c. Hallucination d. Deterministic output

APPLICATION AND PROJECT-BASED QUESTIONS

These problems emphasize modeling, reasoning, and implementation.

1. Implement a simple bigram language model using a small text corpus and compute sequence probabilities.
2. Design and implement an RNN-based model for next-word prediction using a toy dataset.
3. Compare RNN and LSTM performance on a sequence prediction task using accuracy and loss metrics.
4. Implement a basic attention mechanism and visualize attention weights for a sentence.
5. Build a small transformer-based model (or use a pretrained model) for text classification and analyze its performance.

CBSPD
Dedicated to Education

ANSWER KEY

True/False Questions

1. False **2.** False **3.** True **4.** True **5.** False

Fill in the Blanks

1. chain rule **2.** previous hidden state **3.** three (or gating) **4.** weighted
5. positional

Multiple Choice Questions

1. c **2.** b **3.** c **4.** b **5.** c