VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY

Ho Chi Minh City University of Technology

Programming Intergration Project (CO3101)

Project

"Sentiment Analysis"

Instructor:	Trần Huy
Class:	CC11
Group:	1
Students:	Lê Nguyễn Gia Phúc - 2352931
	Tăng Công Thành - 2353104
	Nguyễn Hoàng Quốc - 2353027

Ho Chi Minh City, 2025

Table of Contents

List of Figures

List of Tables

I Introduction

In today's information-driven era, online platforms and social media have become spaces where users frequently share their opinions, emotions, and personal experiences. This massive amount of textual data not only reflects community perspectives on various topics, from products and services to social events, but also creates opportunities for research and practical applications. One of the most important approaches in this context is **Sentiment Analysis**, also known as opinion mining. As a natural language processing (NLP) task, sentiment analysis focuses on determining the emotional tone or subjective opinion expressed in text. It plays a crucial role in understanding public perception, customer satisfaction, and user-generated content across various domains such as product reviews, social media posts, and online discussions. By automatically classifying text as positive, negative, or neutral, sentiment analysis allows organizations and researchers to extract valuable insights from large volumes of unstructured data.

For example, streaming platforms like **Netflix** can apply sentiment analysis to audience reviews and social media discussions about movies and TV shows. By classifying feedback as positive or negative, Netflix can better understand viewer satisfaction, predict whether a show is likely to succeed, and make informed decisions about renewing series, recommending content, or adjusting marketing strategies. This not only enhances user experience but also supports data-driven content development and business growth.

Therefore, this study aims to develop a sentiment analysis system using the IMDB dataset in order to assess the effectiveness of deep learning methods. Specifically, we will apply and compare three approaches: Recurrent Neural Network (RNN), Long-short term memory (LSTM), and a transformer-based neural network model.The objective is not only to achieve high classification accuracy but also to highlight the advantages and limitations of each method, thereby identifying the most suitable approach for real-world applications in natural language processing

II Problem specification

Sentiment analysis is one of the key challenges in the field of natural language processing (NLP) and text data mining. It focuses on identifying and classifying the emotional tone expressed in textual data, such as reviews, comments, or social media posts.
In the context of streaming platforms such as Netflix, Amazon Prime, or IMDB, sentiment analysis plays an even more critical role. By analyzing audience reviews and feedback, these platforms can:

- Understand viewer satisfaction with specific movies or TV shows.
- Predict the potential success of newly released content.
- Improve recommendation systems by tailoring suggestions to user preferences.
- Support content development and investment decisions based on audience sentiment trends.

2.1 Inputs

The IMDB dataset provides the following inputs:

- Movie Reviews (Text Data): A collection of 50,000 movie reviews written by users. These reviews vary in length, vocabulary and writing style, making the dataset diverse and challenging for sentiment analysis.
- Sentiment Labels: Each review is annotated as either positive or negative. These labels serve as ground truth for training and evaluating sentiment classification models.

2.2 Outputs

Sentiment label: The classification result for each review, either Positive or Negative.
Confidence score: The probability value showing how confident the model is in its prediction.

III Related research works

3.1 Umang Gupta, Ankush Chatterjee, Radhakrishnan Srikanth and Puneet Agrawal "A Sentiment-and-Semantics-Based Approach for Emotion Detection in Textual Conversations"

Based on the research, the SS-LSTM (Sentiment & Semantic LSTM) model is designed to detect emotions in textual conversations by combining two crucial sources of information: sentiment and semantics.The model operates on a two-branch parallel architecture, where each branch processes the same input sentence but uses a different word embedding method.

3.1.1 The Sentiment-Focused Branch

- Word Embedding: this branch uses SSWE (Stanford Sentiment Word Embeddings). SSWE is a word embedding method specifically trained to represent not only the semantic meaning but also the sentiment of words. The SSWE vectors are capable of encoding information about the positive, negative or neutral tone of each word, helping the model to recognize emotional key points from the very beginning.

- LSTM Processing: The sentiment-rich SMWE vectors are then fed into a LSTM network. The LSTM processes the sequence of vectors, remembering the context and the relationships between words. This process helps the model to aggregate the emotional information from the entire sentence.

3.1.2 The Semantic-Focused Branch

- Word Embedding: This branch uses GloVe (Global Vectors for Word Representation). GloVe is another popular word embedding method, trained on the co-occurrence matrix of words in a large corpus. GloVe vectors reflect the semantic and grammatical relationships between words. For example, the vector for "king" minus "man" plus "woman" will be close to the vector for "queen".

- LSTM Processing: Similar to the GloVe vectors are passed through a separate LSTM network for processing. This LSTM branch capture the overall semantic meaning and structure of the sentence.

3.1.3 Combination and Classification

The two output vectors from the two branches (Sentiment Encoding and Semantic Encoding) are then concatenated to form a single vector. This combined vector is then fed into a fully connected neural network to perform the emotion classification task.

This approach allows the model to leverage the strengths of both word embedding methods, leading to a more accurate and comprehensive prediction of the text's emotion.

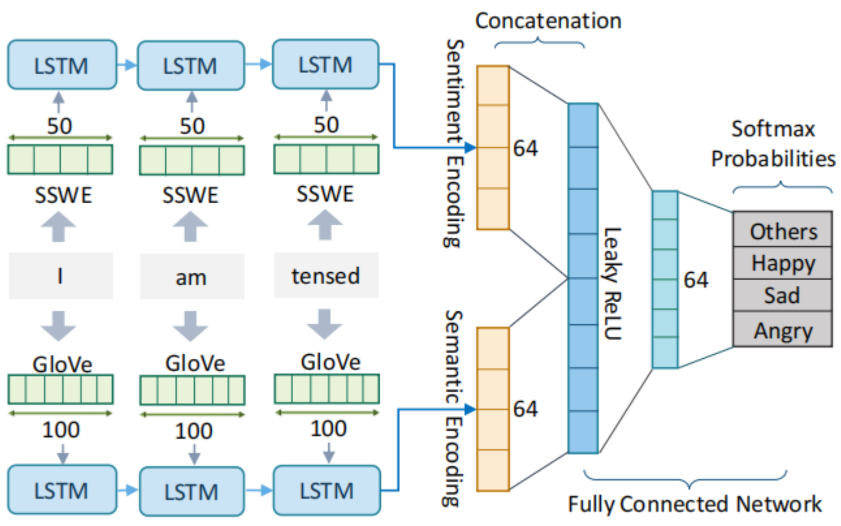
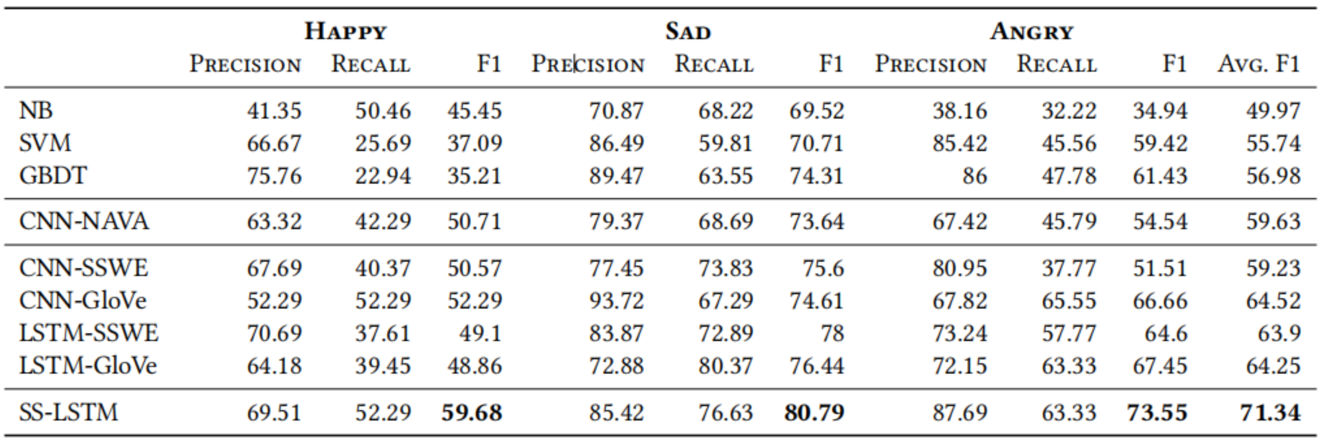


Figure 1: *The architecture of Sentiment and Semantic LSTM (SS-LSTM) model*

3.1.4 Performance

When all model are trained on the dataset of 17.62 million tweets, the SS-LSTM model achieves the best overall performance, clearly outperforming both traditional machine learning methods (NB, SVM, GBDT) and other deep learning approaches (CNN, LSTM with SSWE or GloVe). Notably, SS-LSTM shows strong effectiveness in detecting Sad and Angry emotions.This result demonstrates that SS-LSTM leverages both the richness of big data and the strengths of sequential modeling, making it more robust than existing baselines.



	Happy			Sad			Angry			
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Avg. F1
NB	41.35	50.46	45.45	70.87	68.22	69.52	38.16	32.22	34.94	49.97
SVM	66.67	25.69	37.09	86.49	59.81	70.71	85.42	45.56	59.42	55.74
GBDT	75.76	22.94	35.21	89.47	63.55	74.31	86	47.78	61.43	56.98
CNN-NAVA	63.32	42.29	50.71	79.37	68.69	73.64	67.42	45.79	54.54	59.63
CNN-SSWE	67.69	40.37	50.57	77.45	73.83	75.6	80.95	37.77	51.51	59.23
CNN-GloVe	52.29	52.29	52.29	93.72	67.29	74.61	67.82	65.55	66.66	64.52
LSTM-SSWE	70.69	37.61	49.1	83.87	72.89	78	73.24	57.77	64.6	63.9
LSTM-GloVe	64.18	39.45	48.86	72.88	80.37	76.44	72.15	63.33	67.45	64.25
SS-LSTM	69.51	52.29	**59.68**	85.42	76.63	**80.79**	87.69	63.33	**73.55**	**71.34**

Figure 2: *Performance of Sentiment and Semantic LSTM (SS-LSTM) model*

3.2 Ashish Vaswani et al. “Attention Is All You Need”

In this seminal work, Vaswani et al revolutionized the field of Natural Language Processing by introducing the **Transformer** architecture. Prior to this proposal, dominant sequence transduction models were based on complex Recurrent Neural Networks (RNNs) or Long Short Term Memory (LSTM), which process data sequentially. This sequential nature precluded parallelization within training examples and limited the model’s ability to learn dependencies between distant positions in a sequence.
To address these limitations, the authors proposed a network architecture based solely on **attention mechanisms**, dispensing with recurrence and convolutions entirely. The core innovation is the **Self-Attention** mechanism (specifically, Multi-Head Attention), which allows the model to relate different

positions of a single sequence in order to compute a representation of the sequence. This architecture offers two primary advantages relevant to our study:

1. **Parallelization:** Unlike RNNs, the Transformer processes the entire input sequence simultaneously, significantly reducing training time and allowing for the utilization of large-scale datasets.
2. **Global Dependency Modeling:** The path length between any two positions in the network is constant ($O(1)$), enabling the model to effectively capture long-range dependencies and complex semantic structures, which are critical for understanding context in Sentiment Analysis tasks.

Since the model contains no recurrence, the authors also introduced **Positional Encoding** to inject information about the relative or absolute position of the tokens in the sequence. This work lays the foundation for modern pre-trained language models (such as BERT and RoBERTa) utilized in this thesis.

3.3 Haisheng Deng and Ahmed Alkhayyat "Sentiment analysis using long short term memory and amended dwarf mongoose optimization algorithm"

This research proposes an improved deep learning model for sentiment analysis that combines Long Short-Term Memory (LSTM) with the Amended Dwarf Mongoose Optimization (ADMO) algorithm. The objective is to enhance hyperparameter tuning and improve performance on noisy and informal textual data such as social media posts and movie reviews.

3.3.1 Word Embedding Techniques

- **GloVe:** Captures global statistical information about word co-occurrences, representing semantic and syntactic relations between words.
- **Word2Vec:** Learns distributed vector representations based on local contextual windows, emphasizing semantic similarity.

3.3.2 The LSTM Model

- LSTM was selected as the main architecture due to its ability to capture long-term dependencies and mitigate the vanishing gradient problem inherent in standard RNNs.
- Compared with more complex models such as BERT or BiLSTM, LSTM provides a balance between performance and computational efficiency, making it more suitable for large-scale and noisy sentiment datasets.

3.3.3 Optimization with ADMO

- The proposed Amended Dwarf Mongoose Optimization (ADMO) is employed to automatically fine-tune the hyperparameters of the LSTM model.
- ADMO improves exploration and exploitation in the search space, reducing the need for manual hyperparameter adjustments.

3.3.4 Experimental Results

The model was evaluated on two benchmark datasets:

- **IMDB Dataset:** Achieved 97.74% accuracy with Word2Vec and 97.47% with GloVe.
- **SST-2 Dataset:** Achieved 97.84% accuracy with Word2Vec and 97.51% with GloVe.

The results demonstrate that the proposed LSTM–ADMO framework significantly outperforms other baseline models. The difference between GloVe and Word2Vec embeddings is minimal, indicating that the optimization algorithm is robust across embedding techniques.

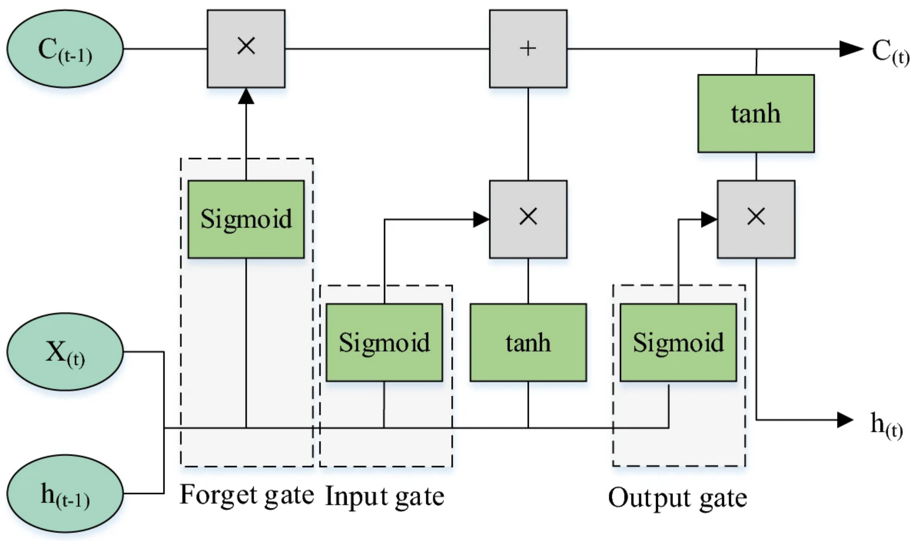


Figure 3: *The architecture of LSTM with Amended Dwarf Mongoose Optimization (ADMO).*

3.3.5 Performance Comparison

Dataset	Embedding	Accuracy (%)
IMDB	Word2Vec	97.74
IMDB	GloVe	97.47
SST-2	Word2Vec	97.84
SST-2	GloVe	97.51

Table 1: *Performance of LSTM–ADMO on IMDB and SST-2 datasets using different embeddings.*

The findings indicate that the integration of ADMO with LSTM offers an efficient and accurate sentiment classification framework. It reduces the reliance on manual hyperparameter tuning and is particularly effective in handling informal and noisy text data commonly found on social media platforms.

IV Applied Methods

4.1

4.2 Optimizer and Learning Rate Scheduler

For optimization, the **Adam optimizer** is employed to adaptively adjust the learning rate for each parameter based on the first and second moments of gradients. The update rules are defined as:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t, \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t}, \\ \theta_{t+1} &= \theta_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}, \end{aligned}$$

where g_t is the gradient at time t, α is the learning rate, β_1, β_2 are decay rates for the moment estimates, and ϵ prevents division by zero.
To ensure training stability and convergence, a **Step Learning Rate (StepLR)** scheduler is applied to decay the learning rate by n% every m epochs, following the rule:

$$\alpha_t = \alpha_0 \times \gamma^{\left\lfloor \frac{t}{\text{step\_size}} \right\rfloor},$$

where $\gamma = n$ and step_size $= m$.
This combination allows the model to learn effectively in the early stages while gradually reducing the learning rate to refine convergence.

4.3 RNN (Recurrent Neural Network)

4.3.1 Overview

A Simple Recurrent Neural Network (Vanilla RNN) is a class of artificial neural networks designed specifically for processing sequential data, such as time-series analysis, speech recognition, and Natural Language Processing (NLP). Unlike traditional Feed-Forward Neural Networks (FNNs) where inputs and outputs are independent of each other, an RNN relies on the concept of internal memory. It retains information from previous inputs in the sequence to influence the current output, making it capable of recognizing temporal dependencies.

However, RNNs suffer from serious drawbacks when applied to sentiment analysis on longer texts: they struggle to capture long-term dependencies due to the vanishing gradient problem, are also vulnerable to exploding gradients, and may fail to retain or effectively use information from distant earlier words. Additionally, because they process sequentially, training and inference can be slower, and they tend to underperform relative to more advanced architectures (LSTM, GRU, or attention-based models) when the sentiment depends on clauses far apart in the sequence or requires understanding complex syntactic or discourse structure.

4.3.2 Architecture

The defining feature of an RNN is its "looped" architecture, which allows information to persist. To understand this clearly, we often visualize the RNN in its Unrolled (or Unfolded) state across time steps.
Components:

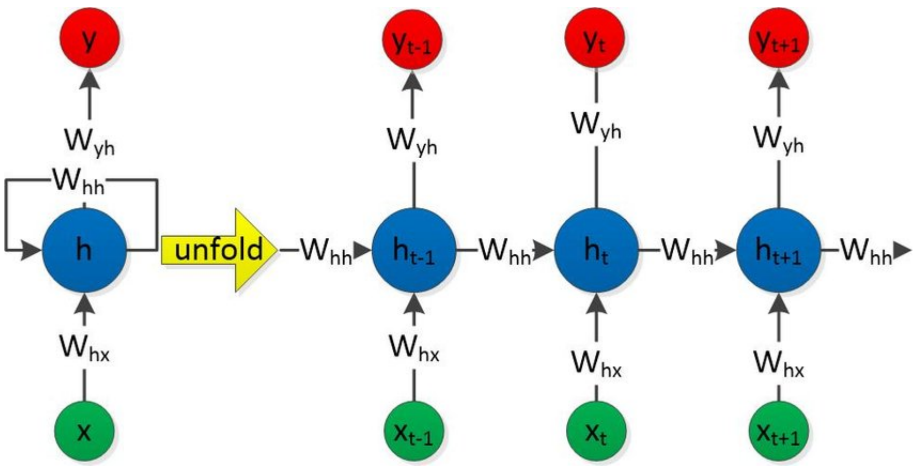


Figure 4: *Basic RNN architecture*

1. **Input** (x_t): The input vector at time step t (e.g., word embedding).
2. **Hidden State** (h_t): The "memory" of the network. It captures information from the current input and the previous context.
3. **Output** ($\hat{y}_t$): The prediction at time step t.
4. **Weights (Shared Parameters):**
   - W_{xh}: Weights connecting the Input layer to the Hidden layer.
   - W_{hh}: Weights connecting the previous Hidden state to the current Hidden state (Recurrent weights).

- W_{hy}: Weights connecting the Hidden layer to the Output layer.

5. **Biases:** b_h (hidden bias) and b_y (output bias).

4.3.3 Working Mechanism: Forward Propagation

In the context of Sentiment Analysis, the RNN typically operates under a "Many-to-One" architecture. The network processes the input sequence word by word to construct a final context vector, which is then used for classification.
Given an input sequence of vectors $X = \{x_1, x_2, \ldots, x_T\}$, where $x_t \in \mathbb{R}^d$ represents the word embedding at time step t, the forward propagation proceeds as follows:

4.3.3.1 Hidden State Update (Memory Accumulation) At each time step t, the hidden state h_t is updated based on the current input x_t and the previous hidden state h_{t-1}. This update step fuses new information with the historical context:

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \tag{1}$$

Where:

- $W_{xh} \in \mathbb{R}^{h\times d}$: Input-to-Hidden weight matrix.
- $W_{hh} \in \mathbb{R}^{h\times h}$: Hidden-to-Hidden (Recurrent) weight matrix.
- $b_h \in \mathbb{R}^h$: Bias vector for the hidden state.
- tanh: The hyperbolic tangent activation function, squashing values to $[-1, 1]$ to regulate information flow.

4.3.3.2 Final Prediction (Sentiment Classification) Unlike sequence labeling tasks, sentiment analysis relies on the final hidden state h_T (after processing the last word) as the summary representation of the entire review. This vector is passed through a fully connected layer followed by a Sigmoid (or Softmax) activation function to produce the probability of the sentiment:

$$\hat{y} = \sigma(W_{hy}h_T + b_y) = \frac{1}{1 + e^{-(W_{hy}h_T + b_y)}} \tag{2}$$

Where $\hat{y} \in (0, 1)$ represents the predicted probability.

4.3.4 The Vanishing and Exploding Gradient Problems

Training RNNs on long sequences presents a fundamental challenge known as the vanishing and exploding gradient problem. This issue arises directly from the mathematical nature of BPTT.

4.3.4.1 Mathematical Derivation Using the chain rule, the gradient of the hidden state at time t with respect to a previous state at time k ($k < t$) is a product of Jacobian matrices:

$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k+1}^{t} \left(W_{hh}^T \cdot \mathrm{diag}(\sigma'(z_i))\right) \tag{3}$$

where $\sigma'(z_i)$ is the derivative of the activation function with respect to its input at step i.

4.3.4.2 The Vanishing Gradient Problem The vanishing gradient problem occurs when the gradient signal diminishes exponentially as it backpropagates through time.

- **Cause:** This typically happens when the spectral radius (largest eigenvalue) of the weight matrix W_{hh} is less than 1 ($\rho(W_{hh}) < 1$) or when the derivative of the activation function is small (e.g., $|\sigma'(z)| < 1$ for tanh).

- **Mechanism:** As shown in Equation (6), multiplying many terms smaller than 1 causes the gradient to approach zero:

$$\lim_{(t-k)\to\infty} \left\| \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} \right\| \approx 0 \tag{4}$$

- **Consequence:** The weights capturing long-term dependencies (e.g., a sentiment word at the start of a long review) are not updated. The model "forgets" early information.

4.3.4.3 The Exploding Gradient Problem Conversely, the exploding gradient problem refers to the scenario where the gradients grow exponentially.

- **Cause:** This occurs when the spectral radius of the recurrent weight matrix is greater than 1 ($\rho(W_{hh}) > 1$).
- **Mechanism:** The product term in Equation (6) grows exponentially with the sequence length.
- **Consequence:** The gradient values become extremely large, causing drastic weight updates, numerical instability (NaN), and divergence. This is mitigated using **Gradient Clipping**.

4.4 LSTM (Long-Short Term Memory)

4.4.1 Overview

LSTM networks extend vanilla RNNs by introducing gating mechanisms that resolve vanishing and exploding gradient problems, enabling the modeling of long-range dependencies in sequential data. Traditional RNNs face significant limitations when learning long-range dependencies as the network backpropagates errors through many time steps, the gradient values can diminish exponentially or grow without bound, effectively "forgetting" contributions from far earlier steps. The vanishing and exploding gradient problems mean standard RNNs have difficulty learning the influence of words that appear long before the point of prediction (e.g. a crucial early sentence in a long review), often resulting to failure upon capturing long-term context in sentiment analysis. This limitation motivated the development of improved recurrent architectures that can maintain memory over longer sequences.

To overcome the shortcomings of standard RNNs, Long Short-Term Memory (LSTM) networks were introduced by Hochreiter and Schmidhuber in 1997. The key innovation of LSTMs is the introduction of a special memory unit – often called a cell – equipped with gating mechanisms that regulate the flow of information. By augmenting the recurrent network with an internal memory cell and gating signals, LSTMs can learn what to remember, what to forget, and what to output at each time step. This design allows important information to persist over long sequence distances with stable gradients to propagate more steadily, which is crucial for tasks like sentiment analysis on lengthy movie reviews.

4.4.2 Working mechanism

An LSTM unit augments the standard RNN with a cell state (the memory) and three primary gates that control information flow: a forget gate, an input gate, and an output gate. These components work together to decide at each step which information is retained, which is discarded, and what will be produced as output.

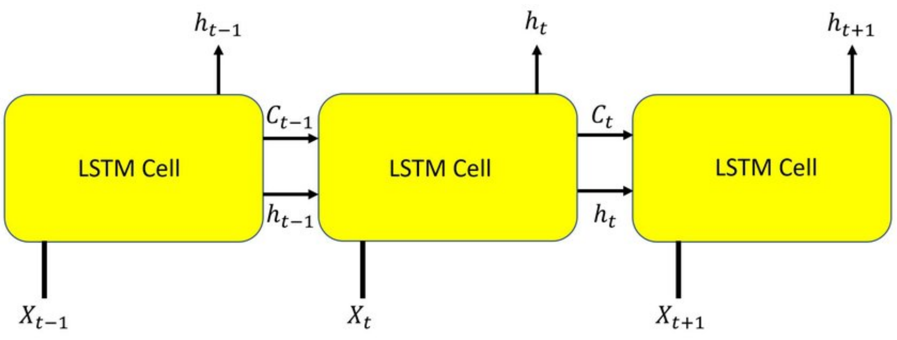


Figure 5: *Layout of the LSTM with cell connections*

As shown in Figure 5, the internal structure of the LSTM cell features a horizontal cell state pathway modulated by vertical gates, enabling selective memory management. This design underpins the recurrent architecture depicted in Figure 6.

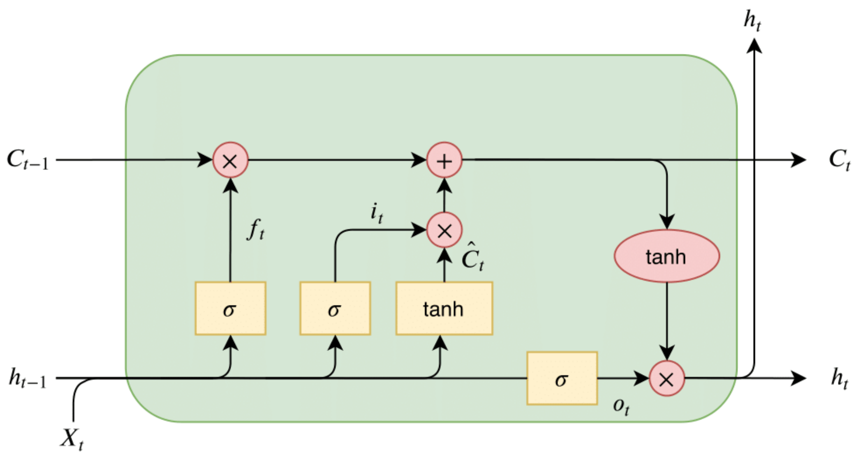


Figure 6: *The basic architecture of LSTM*

X_t: input, h_t: output
f_t, i_t, o_t: forget, input, output gates
C_t: cell state, $\hat{C}_t$: candidate cell state

4.4.3 Cell State and Overall Information Flow

The cell state C_t acts as the core memory pathway running through the top of the diagram. The flow integrates short-term (hidden state) and long-term (cell state) memories, allowing LSTMs to handle dependencies in sentiment analysis.
Mathematically, the cell state update balances retention and addition:

$$C_t = f_t \odot C_{t-1} + i_t \odot \hat{C}_t,$$

where $\odot$ denotes element-wise multiplication. This additive structure, combined with gates, maintains a Jacobian determinant near 1, preserving error signals during Backpropagation Through Time (BPTT). Specifically, the partial derivative of C_t with respect to C_{t-1} is f_t, a value between 0 and 1 modulated by the forget gate, which can be trained to sustain gradients across hundreds or thousands of time steps without decay.
In the domain of sentiment analysis, the cell state's robust information flow is instrumental in modeling intricate textual dependencies. Because C_t is a sum of contributions, important information can accumulate over time, and unimportant information can gradually fade away. The cell state thus carries forward a refined memory that reflects the sequence history up to time t, selectively remembering what

the model has deemed important for the task at hand. For example, in processing a review such as "The movie started strong but ultimately disappointed," the cell state can retain the positive connotation from "strong" across the sequence to inform the negative shift at "disappointed," enabling accurate polarity detection even in the presence of negations, sarcasm, or contextual reversals.

4.4.4 Forget Gate: Selective Discarding

The forget gate f_t determines which portions of the prior cell state C_{t-1} to retain or discard, addressing RNN's overwriting issue. It computes:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f),$$

where σ denotes the sigmoid activation function, Weight matrix W_f and Bias vector b_f are learnable parameters, and $[h_{t-1}, x_t]$ denotes concatenation of previous hidden state and current input. This gate produces a vector of retention probabilities for each dimension of the cell state.
During training, the parameters W_f and b_f are optimized using gradient-based methods, such as SGD, Adam or RMSprop, to minimize the overall loss function (e.g., cross-entropy for sentiment classification tasks). The multiplicative nature of the gate facilitates stable gradient flow during backpropagation through time (BPTT). To illustrate the gate's impact, consider the gradient propagation - the partial derivative $\partial C_t / \partial C_{t-1} = f_t$, which, when chained over multiple steps, yields:

$$\partial C_T / \partial C_t = \prod_{k=t+1}^{T} f_k$$

Training adapts f_k to values near 1 for preserved features, mitigating vanishing gradients. In sentiment analysis applications, the forget gate learns to discard irrelevant contextual information while preserving sentiment-bearing features across long distances: A value of zero means “let nothing through” while a value of one means “let everything through”. This means that by the time the LSTM reaches the end of the sentence, it still “remembers” the important sentiment information from the beginning or middle.

4.4.5 Input Gate and Candidate Cell Update: Selective Addition

The input gate i_t regulates new information addition to the cell state while a vector of new candidate values $\hat{C}_t$ suggests what could be added to the state, providing a mechanism for selective addition that complements the forget gate's discarding function. They are computed as:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i),$$

$$\hat{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

The tanh activation function introduces a nonlinear transformation bounded between -1 and 1. The update $i_t \odot \hat{C}_t$ scales and adds relevant new data, allowing LSTMs to incorporate fresh sentiment cues (e.g., intensifiers like "very") without overwhelming existing memory. This selective addition is crucial for maintaining bounded cell state values and mitigating exploding gradients, as the tanh function normalizes inputs while the sigmoid gate filters relevance. During optimization, parameters W_i, b_i, W_C, b_C are tuned to prioritize salient features, often with techniques like layer normalization or dropout to enhance generalization. For the IMDB dataset, this mechanism proves essential in accumulating sentiment evidence throughout the text while filtering noise from neutral passages.

4.4.6 Output Gate and the Hidden State

The output gate o_t governs the selective exposure of the updated cell state C_t to the hidden state h_t, determining which aspects of the internalized memory are propagated forward to influence predictions or subsequent time steps. This gate ensures that only relevant synthesized information is revealed, protecting the cell state from unnecessary perturbations while enabling the network to output contextually appropriate representations.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

Values near 1 permit full passage of corresponding cell state elements, while those near 0 suppress them, providing strong control over information flow. Specifically, a memory cell can accrue information across many time steps without impacting the rest of the network as long as the output gate takes values close to 0, and then suddenly impact the network at a subsequent time step as soon as the output gate flips from values close to 0 to values close to 1. This nearly binary-like thresholding, though probabilistic due to the sigmoid, enables the LSTM to decouple long-term memory accumulation from immediate output requirements, a feature absent in vanilla RNNs where states are always fully exposed. This gate works in conjunction with the activated cell state to produce the hidden state:

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

where $\odot$ signifies element-wise multiplication, and tanh() activation function normalizes the cell state to [-1,1], introducing bounded nonlinearity for activation stability. In vectorized form for batched processing, this extends to $\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t)$, where matrices accommodate multiple sequences, enabling efficient parallel computation in frameworks like PyTorch or TensorFlow.
The hidden state h_t is the externally visible, short-term working representation produced at time t. It serves three roles simultaneously:

- **Interface to downstream computation:** h_t is the signal consumed by subsequent layers (e.g., classification heads, attention blocks, or additional recurrent layers).
- **Recurrent carrier:** h_t is fed back into the next step's gates, shaping how the model will retain, write, and expose information at time $t + 1$.
- **Bounded, filtered view of memory:** Because h_t is formed as $h_t = o_t \odot \tanh(C_t)$, it is both range-limited and selectively exposed. The tanh nonlinearity prevents unbounded activations; the output gate o_t masks irrelevant coordinates.

Functionally, h_t operates as a versatile short-term memory reservoir, responsive to recent inputs x_t and antecedent states, while serving as the LSTM cell's external output for immediate utilization. This dual role positions it as the "working memory" holding fleeting details for on-the-fly processing, in contrast to the cell state's role as a more long-term archive.

4.4.7 Forward Pass and Training Dynamics

The forward pass unfolds sequentially, with notations as follows for clarity:

Variable	Description
$x_t \in \mathbb{R}^d$	Input vector to LSTM unit
$f_t \in (0,1)^h$	Forget gate activation vector
$i_t \in (0,1)^h$	Input/update gate activation vector
$o_t \in (0,1)^h$	Output gate activation vector
$h_t \in (-1,1)^h$	Hidden state vector (output)
$\hat{C}_t \in (-1,1)^h$	Candidate cell state vector
$C_t \in \mathbb{R}^h$	Cell state vector
$z_t \in \mathbb{R}^{d+h}$	Concatenated input $[x_t, h_{t-1}]$
$W \in \mathbb{R}^{h\times d}, U \in \mathbb{R}^{h\times h}, b \in \mathbb{R}^h$	Weight matrices and bias parameters (learnable)

Table 2: *Notation for LSTM Forward Pass Variables*

Training is performed using Backpropagation Through Time (BPTT). The network is unrolled across the sequence, and gradients are propagated backward from the final loss $\mathcal{L}$ (typically cross-entropy on the last hidden state).
Although all three gates use the sigmoid activation, they learn distinct roles because each has **independent parameters** (W_f, W_i, W_o, etc.) and receives **different error signals**: The key gradient path through the cell state is:

$$\partial C_T / \partial C_t = \prod_{k=t+1}^{T} f_k$$

When the network needs long-term memory, it learns to keep $f_k \approx 1$ along the path, preventing vanishing gradients.

Through the hidden state (short-term output):

$$h_t = o_t \odot \tanh(C_t) \qquad \Rightarrow \qquad \partial h_t / \partial C_t = o_t \odot (1 - \tanh^2(C_t))$$

The model learns to open the output gate ($o_t \approx 1$) when the stored memory should influence the prediction. All parameters (W_f, b_f, etc.) are updated via Adam using these gradients by the chain rule in the usual way. This mechanism allows the LSTM to automatically learn long-range dependencies, such as remembering an early negation in a long movie review, resulting in superior performance on sentiment analysis tasks.

- **Forget gate** learns to output ≈ 1 when old memory should be preserved (gradient flows through f_t).
- **Input gate** learns to output ≈ 1 when new information is important.
- **Output gate** learns to output ≈ 1 when current memory should influence predictions.

4.5 Transformer

4.5.1 Overview

Transformer is a deep learning architecture which is first introduced by Vaswani (2017) in the paper “Attention Is All You Need”.Unlike previous sequence modeling approaches such as RNN and LSTM networks, Transfomer rely entirely on a mechanism called self-attention to capture relationships between elements in a sequence. This allows it to model long-range dependencies more effectively and in parallel, overcoming the limitations of sequential computation in RNN-based models.

4.5.2 Working Mechanism

In general, the Transformer processes input sequences through a series of key steps. First, input tokens are converted into vector representations with embeddings and enhanced with positional encoding to retain order information. Next, the model applies the self attention mechanism and multi-head attention to capture contextual relationships between tokens. The results are passed through a feed-forward network with residual connections and normalization to improve stability and avoid gradient vanishing. Finally, in the encoder-decoder architecture, the encoder transforms the input into contextual representations, while the decoder generates the output sequence by attending both to encoder outputs and previously generated tokens.

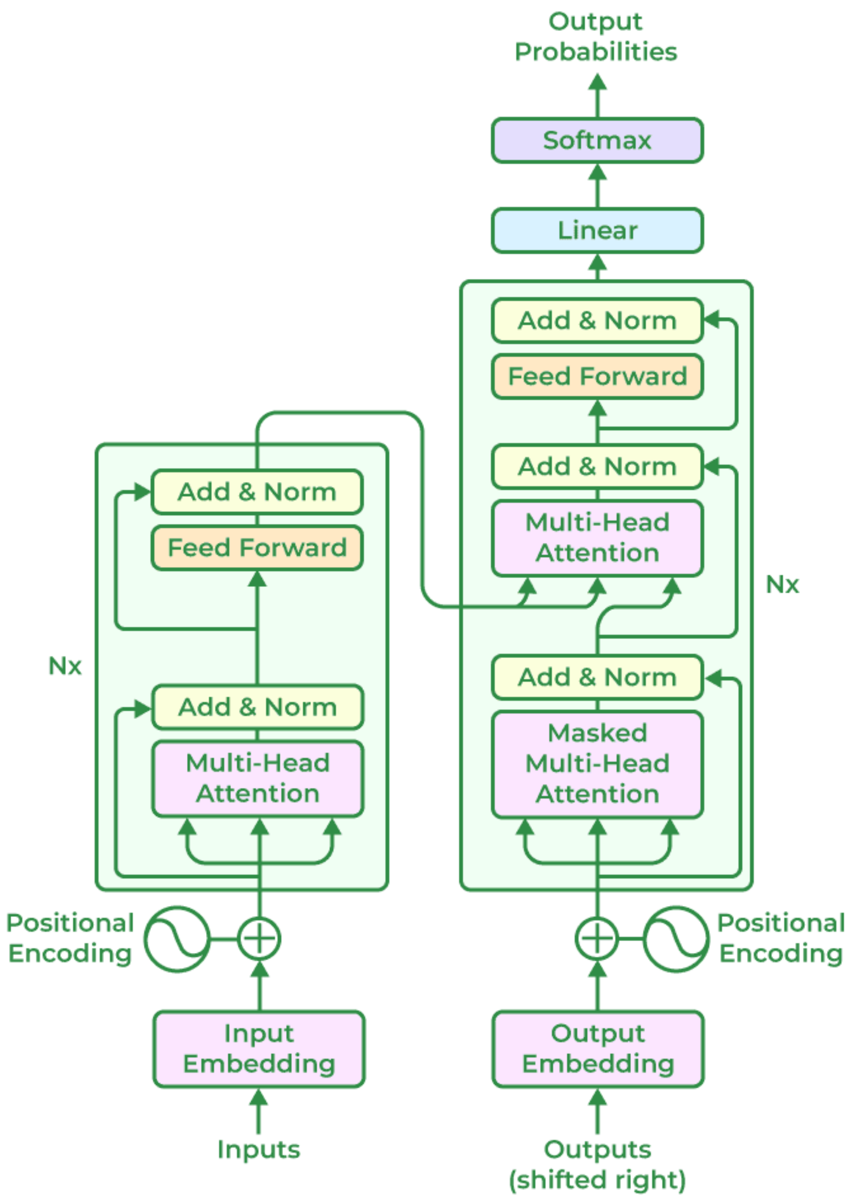


Figure 7: *Transformer's architecture*

The original Transformer model, introduced in the paper "Attention Is All You Need," was designed to address sequence-to-sequence tasks such as text generation, and it outperformed all previous approaches at the time. However, sentiment analysis is a classification problem, so only the encoder component is required. The output matrix from the encoder is passed through an average pooling layer to obtain a single vector, which is then fed into one or more dense layers. Finally, the outputs are passed through a SoftMax activation function to produce the classification results.

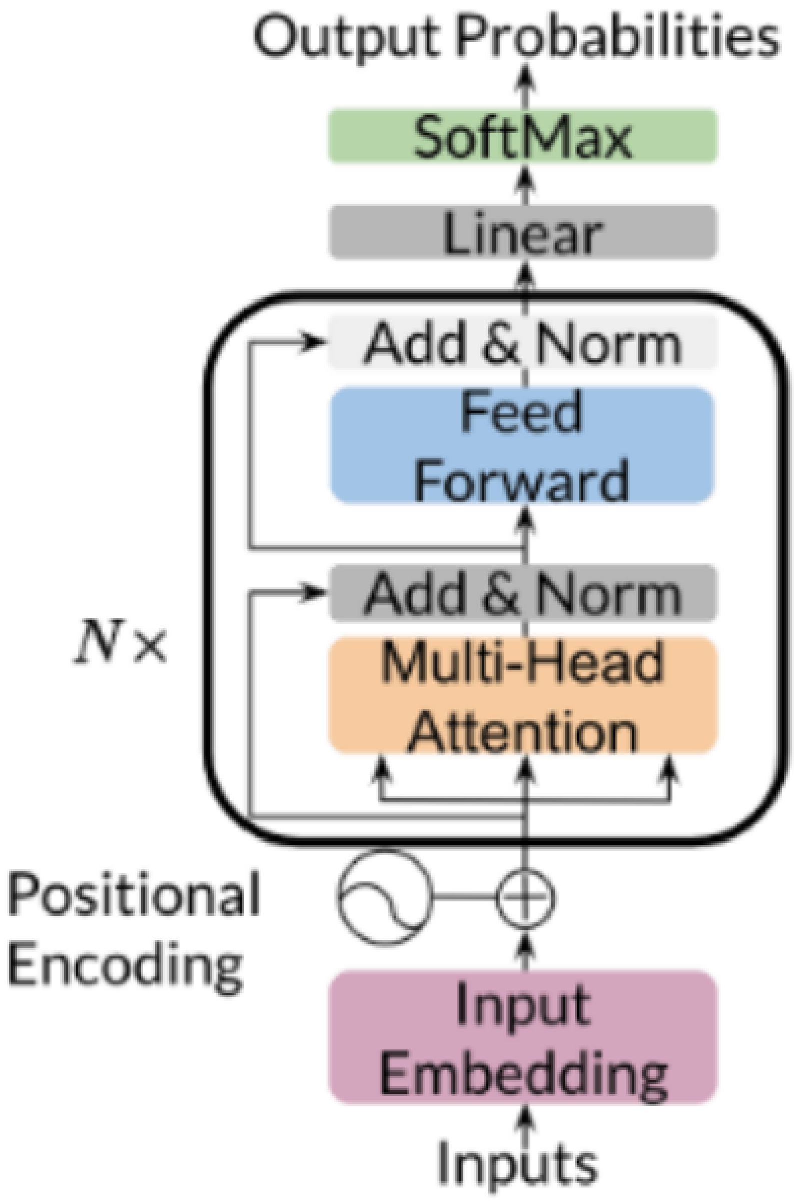


Figure 8: *Encoder-based classifier architecture*

After giving an overview of the Transformer architecture, we now proceed to a detailed explanation of its components.

4.5.3 Input embedding

Raw input tokens, such as words or subwords, cannot be processed directly by the Transformer since they are symbolic and discrete. Therefore, the first step is to convert them into continuous vector representations, known as embeddings. Each token is mapped to a vector of fixed dimension through a learnable embedding matrix. This transformation allows the model to represent tokens in a numerical form that preserves semantic relationships.
The embedding process ensures that tokens with similar meaning have similar representations in the vector space. For example, the words "happy" and "joyful" will have embeddings that are closer to each other compared to "happy" and "cat". This property enables the model to capture semantic and syntactic relationships between words.
In practice, *embeddings* can either be:

- **Trained from scratch** during model training, where the embedding matrix is updated through *backpropagation.*
- **Initialized from *pre-trained embeddings*** (Word2Vec, GloVe, SSWE, or embeddings from large pre-trained Transformer models like BERT), which can significantly improve performance by leveraging prior knowledge.

According to the research,the embedding vectors are then scaled by the square root of their dimension to maintain numerical stability. After this step, positional encoding is added to provide information about token order before the sequence is passed into the attention layers of the Transformer.

4.5.4 Positional Encoding

Unlike recurrent or convolutional models, the Transformer has no inherent mechanism to capture the order of tokens in a sequence, since it processes all tokens in parallel. To overcome this limitation, positional encoding is introduced to provide information about the relative and absolute positions of tokens.
In practice, positional encodings are vectors of the same dimension as the input embeddings, allowing them to be added directly to the token embeddings before entering the encoder. This combination enables the model to retain both semantic information (from embeddings) and sequential order (from positional encodings). The original Transformer paper proposed a deterministic approach based on sine and cosine functions of different frequencies:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

Where:

- ***pos***: the position of the token in the sequence,
- ***i***: the dimension index,
- d_{model}: the embedding dimension

The use of sine and cosine function for positional encoding in the original Transformer offers several advantages:

- First,since the formulation is continuous, the model can generalize to sequences longer than those seen during training
- Second,The varying wavelengths across dimensions capture positional information at different levels of detail, ranging from small-scale details to broad-level patterns item
- Most importantly, for any fixed offset k, the encoding at position $pos + k$ can be expressed as a linear function of the encoding at position pos:

$$\sin(pos + k) = \sin(pos)\cos(k) + \cos(pos)\sin(k)$$
$$\cos(pos + k) = \cos(pos)\cos(k) - \sin(pos)\sin(k)$$

  This property makes it easier for the model to learn to attend to tokens based on their relative positions, which is crucial for capturing contextual dependencies in natural language.

In conclusion, by combining token embeddings with positional encodings, the Transformer effectively integrates both semantic and sequential information, enabling the self-attention mechanism to operate with awareness of word order.

4.5.5 Attention Mechanism

The Attention mechanism is the central component of the Transformer, enabling the model to focus on the most revelant parts of the input sequence when processing a specific token. The core idea of Attention is to compute the degree of relevance between the current token and all other tokens in the sequence, and then use this information to weight their contributions.
In the Transformer, Attention is implemented through three vectors derived from the input embeddings (or from the outputs of the previous layer): Query (Q), Key (K) and Value (V).

- Query (Q): represents the token currently being processed
- Key (K): represents identifiers for all tokens in the sequence
- Value (V): contains the actual semantic content of each token

The Attention process follows three main steps:

- **Step 1**: Compute similarity scores: for each token, the dot product between its Query and all Keys is calculated. The higher the score, the more relevant that Key is to the Query.
- **Step 2**: Normalization with Softmax. These similarity scores are then passed through a Softmax function to obtain a probability distribution, ensuring that the weights are positive and sum to one.
- **Step 3**: Each Value vector is multiplied by its corresponding weight to form the output representation for the token.

The overall Scaled Dot-Product Attention can be expressed as

$$Attention(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Where:

- d_k: the dimensionality of the Key vectors, which usually equals the embedding dimension. This is used as a scaling factor to prevent excessively large dot product values that could saturate the *Softmax*.
- QK^T: measures the pairwise similarity between queries and keys.
- The *Softmax* distribution is then applied to weight the Value vectors accordingly.

By leveraging this mechanism, the Transformer effectively captures both short-range and long-range dependencies within a sequence, which traditional RNN-based architectures struggle with due to vanishing gradients and sequential processing constraints.

4.5.6 Multi-head Attention

While a single Attention mechanism can already capture dependencies between tokens, it is limited to representing relationships in one subspace of the embedding. To overcome this limitation, the Transformer employs Multi-head Attention, which allows the model to attend to information from multiple subspaces simultaneously.
Instead of performing one attention operation with d_{model}-dimensional Queries, Keys, and Values, the Transformer linearly projects them h times using different, learnable weight matrices. Specifically, given the original Q, K, V (all with dimension d_{model}), each head i computes:

$$Q_i = QW_i^Q, \quad K_i = KW_i^K, \quad V_i = VW_i^V$$

where $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ are projection matrices, and d_k, d_v are the reduced dimensions of Keys and Values.
Each head then applies the Scaled Dot-Product Attention independently:

$$\text{head}_i = Attention(Q_i, K_i, V_i) = \text{Softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right)V_i$$

The outputs of all heads are concatenated and linearly transformed using an output matrix W^O:

$$MultiHead(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

This design provides several advantages:

- Each head learns to capture different types of dependencies, such as syntactic relations or semantic associations.
- By working in multiple subspaces, the model learns richer contextual representations than a single-head mechanism could provide.
- Parallel processing across heads enable the model to integrate diverse information efficiently

In practice,the dimensions are chosen so that the concatenated outputs of all heads match the original model dimension: h * dv = dmodel . Since the outputs preserve the same dimensionality as the inputs, the model can stack multiple layers of Multi-Head Attention in sequence without requiring additional dimensional adjustments. This stacking allows the network to progressively refine contextual representations, enabling deeper layers to capture increasingly complex relationships before the information is passed to subsequent feed-forward layers.

4.5.7 Position-wise Feed-Forward Networks

Following the Multi-Head Attention meachnism, each position in the sequence is processed independently through a Position-wise Feed-Forward Network (FFN). Unlike recurrent or convolutional layers, which consider temporal or spatial dependencies, the FFN operate identiacally and separately on every position, ensuring that the contextualized information obtained from the attention mechanism is transformed in a uniform way across the sequence.
The Position-wise FFN consists of two fully connected linear layers with a non-linear activation function (usually RELU) in between. Normally, for an input vector x:

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{5}$$

Where:

- $W_1 \in \mathbb{R}^{d_{model} \times d_{ff}}$, $b_1 \in \mathbb{R}^{d_{ff}}$ are the parameters of the first linear transformation.
- $W_2 \in \mathbb{R}^{d_{ff} \times d_{model}}$, $b_2 \in \mathbb{R}^{d_{model}}$ are the parameters of the second linear transformation.
- d_{ff} is typically larger than d_{model} (e.g., $d_{ff} = 2048$ and $d_{model} = 512$ in the original Transformer), allowing the network to project into a higher-dimensional space before compressing back to the model dimension.

This design serves two main purposes:

- **Non-linear transformation:** The activation function introduces non-linearity, enabling the model to learn more complex mappings than would be possible with only attention and linear operations.
- **Dimensional expansion and compression:** By temporarily increasing dimensionality to dff , the FFN allows the model to capture more expressive intermediate representations before projecting back to d_{model}.

Since the FFN is applied independently to each position, it does not direcly model relationships between tokens. Instead, its role is to refine and transform the features that already contextualized by the Multi-head Attention mechanism, contributing to the overall depth and representational power of the Transformer.

4.5.8 Residual Connections and Layer Normalization

In the Transformer architecture, both the Multi-head Attention and the Position-wise Feed-Forward Network are wrapped with residual connections followed by layer normalization. Specifically, the output of each sub-layer (either Multi-head Attention or FFN) is added to its original input through a residual connection, and the sum is then normalized across the embedding dimension.
This design helps stabilize training and prevents the network from degrading as it grows deeper. The residual connection ensures that the original input signal is preserved, while the normalization step controls the variance of activation across the sequence, making the model more robust.
Formally, for a given sub-layer function Sublayer(x) (such as attention or feed-forward), the output is computed as:

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

LayerNorm operates on each individual sample,which is token in this case, normalizing across the feature dimension within that sample. This makes it paricularly well-suited for sequence models such as Transformer, where sequence lengths may vary and batch sizes can be small Formally, given an input vector

$$x = (x_1, x_2, \dots, x_d)$$

with hidden dimension d, LayerNorm is computed as:

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

where:

$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i \quad \text{is the } \textbf{mean} \text{ across all features,}$$

$$\sigma^2 = \frac{1}{d}\sum_{i=1}^{d} (x_i - \mu)^2 \quad \text{is the } \textbf{variance}\text{,}$$

$$\epsilon \quad \text{is a small constant added for numerical stability.}$$

The normalized values $\hat{x}_i$ are then rescaled and shifted using two learnable parameters:

$$y_i = \gamma \hat{x}_i + \beta$$

where γ (scale) and β (shift) are trainable parameters that allow the model to adjust or even undo normalization if beneficial.

4.5.9 Forward and Backward Propagation

To understand how the Transformer learns to classify sentiment, we mathematically formulate the flow of data during the training process, divided into the Forward Pass (inference) and Backward Pass (learning).

4.5.9.1 Forward Propagation The forward propagation transforms input token indices into a probability distribution over the sentiment classes. Let $X = \{x_1, x_2, \dots, x_T\}$ be the sequence of input token indices of length T.

1. **Embedding and Positional Encoding:** The input indices are converted into dense vectors and summed with positional encodings to retain sequential order:

$$H^{(0)} = \text{Embedding}(X) \times \sqrt{d_{model}} + \text{PE} \tag{6}$$

   where $H^{(0)} \in \mathbb{R}^{T \times d_{model}}$ represents the initial hidden states.

2. **Encoder Layers (Stacked):** The data passes through N stacked encoder blocks. For each layer $l \in \{1, \dots, N\}$, the output $H^{(l)}$ is computed as:

$$Z^{(l)} = \text{LayerNorm}(H^{(l-1)} + \text{MultiHeadAttn}(H^{(l-1)})) \tag{7}$$

$$H^{(l)} = \text{LayerNorm}(Z^{(l)} + \text{FFN}(Z^{(l)})) \tag{8}$$

3. **Sequence Aggregation (Mean Pooling):** Unlike RNNs which typically use the last hidden state, our Transformer implementation aggregates information from all tokens to form a fixed-size sentence representation. We compute the mean of the encoder outputs, accounting for padding masks:
$$h_{avg} = \frac{1}{T_{valid}} \sum_{t=1}^{T} H_t^{(N)} \cdot \mathbb{I}(x_t \neq \text{PAD}) \tag{9}$$
where $h_{avg} \in \mathbb{R}^{d_{model}}$ is the global context vector and $\mathbb{I}$ is the indicator function for non-padding tokens.

4. **Classification Head:** The aggregated vector is passed through a feed-forward classifier to produce the final logits z:
$$z = W_2(\text{ReLU}(W_1 h_{avg} + b_1)) + b_2 \tag{10}$$
Finally, the Softmax function produces the predicted probabilities $\hat{y}$:
$$\hat{y} = \text{Softmax}(z) \tag{11}$$

4.5.9.2 Backward Propagation The backward pass calculates the gradients required to update the model parameters θ (weights and biases) to minimize the error.

1. **Loss Calculation:** We utilize the Categorical Cross-Entropy Loss function. Given the true one-hot encoded label y and predicted probability $\hat{y}$:
$$\mathcal{L}(\theta) = -\sum_{c=1}^{C} y_c \log(\hat{y}_c) \tag{12}$$

2. **Gradient Computation (Backpropagation):** Using the Chain Rule of calculus, we compute the gradient of the loss with respect to every parameter θ in the network. For the Transformer, the error signal propagates back through the classification head, the mean pooling operation, and through the layers of Attention and FFNs:
$$\nabla_\theta \mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \dots \frac{\partial H^{(l)}}{\partial \theta} \right) \tag{13}$$
Crucially, the gradient flows through the Softmax function in the Attention mechanism, allowing the model to learn which tokens to focus on.

3. **Parameter Update:** The parameters are updated using the Adam optimizer. The weights are adjusted based on the calculated gradients and adaptive moment estimates:
$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{14}$$

V Evaluation Metrics

5.1 Accuracy

Accuracy measures the proportion of correctly predicted samples over the total number of samples:
$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$
Where:

- **TP (True Positive):** The number of positive samples that are correctly predicted as positive
- **FP (False Positive):** The number of positive samples that are incorrectly predicted as negative
- **TN (True Negative):** The number of negative samples that are correctly predicted as negative

- **TN (False Negative):** The number of negative samples that are incorrectly predicted as positive

Pros:

- Simple to calculate and easy to understand
- Works well when classes are balanced

Cons:

- Can be misleading with imbalanced datasets (90% positive, 10% negative)
- May overestimate model performance if it always predicts the majority class

5.2 Precision

Precision quantifies how many of the samples predicted as positive are actually positive

$$Precision = \frac{TP}{TP + FP}$$

Pros:

- Measures the quality of predictions for a specific class
- Important when the cost of false positives is high. For example, labeling a negative review as positive could lead to overlooking the real issues and damaging customer trust

Cons:

- Does not reflect the model's ability to find all true instances (recall might be low).
- Focusing only on precision can lead to ignoring many relevant examples

5.3 Sensitivity(Recall)

Also known as True Positive Rate (TPR), recall measures the proportion of actual positive samples that are correctly identified

$$Sensitivity = \frac{TP}{TP + FN}$$

Pros:

- Reflects the ability to "capture" all important examples.
- Useful when missing an instance has a high cost

Cons:

- High recall does not guarantee correctness (precision may be low).
- Focusing only on recall may generate many false positive

5.4 F1-score

The F1-score is the harmonic mean of precision and recall.It balances the trade-off between precision and recall, making it particularly usefull when the dataset is imbalanced

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

Pros:

- Balances precision and recall
- Useful for imbalanced datasets

Cons:

- Less intuitive than accuracy
- Macro F1 (average the F1 of each class equally) may not account for the proportion of each class

5.5 Cross-Entropy Loss

It is crucial to distinguish between the metrics used for final model evaluation and the objective function used during the training process. While the model's ultimate performance is typically benchmarked using interpretive metrics such as Accuracy, F1-Score, Recall, and Precision, the optimization algorithm focuses exclusively on minimizing the Loss function. Evaluation metrics like accuracy are often discrete and non-differentiable, making them unsuitable for gradient-based optimization methods. In contrast, the Cross-Entropy Loss provides a smooth, differentiable landscape that allows the optimizer to calculate gradients and adjust weights via backpropagation. Therefore, minimizing the loss serves as the fundamental mathematical proxy for "learning," driving the model toward states that eventually yield higher performance on the final evaluation metrics.
The core principle of Cross-Entropy Loss is to quantify the "distance" between the predicted probability distribution and the true distribution. It heavily penalizes the model for predictions that are both confident and incorrect. For instance, If a review is truly "Positive" but the model confidently predicts it is "Negative" with a high probability, the loss value will be very large. Conversely, if the model correctly predicts the sentiment with high confidence, the loss will be very small, close to zero.

For this project, while the IMDB dataset represents a binary classification task ('Positive' and 'Negative'), we have chosen to implement the Categorical Cross-Entropy loss function instead of Binary Cross-Entropy. This decision was made strategically to build a flexible model architecture. This approach ensures that the model can be seamlessly extended to handle multi-class sentiment analysis problems in the future (e.g., classifying sentiments into 'Very Positive', 'Positive', 'Neutral', and 'Negative') without requiring fundamental changes to the loss function or model structure.

The final output layer of the model uses the Softmax activation function. Softmax transforms the model's raw output scores (logits) into a probability distribution, where each element represents the probability of a specific class, and the sum of all elements equals 1.

The mathematical formula for Categorical Cross-Entropy for a single sample is:

$$\text{Loss} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

Where:

- C: the total number of classes (in this case, $C = 2$).
- y_i: the true value for class i from the one-hot vector (it is 1 for the correct class and 0 for all others).
- $\hat{y}_i$: the model's predicted probability for class i.

VI Data Preprocessing

6.1 Dataset Introduction

The dataset was introduced by Stanford researchers and its specifically designed to be large enough to train and evaluate deep learning models effectively. It consists of 50.000 movie reviews written in English, sourced from the Internet Movie Database (IMDB). These reviews are highly polarized, meaning they express strong positive or negative opinions, which makes them ideal for sentiment classification.

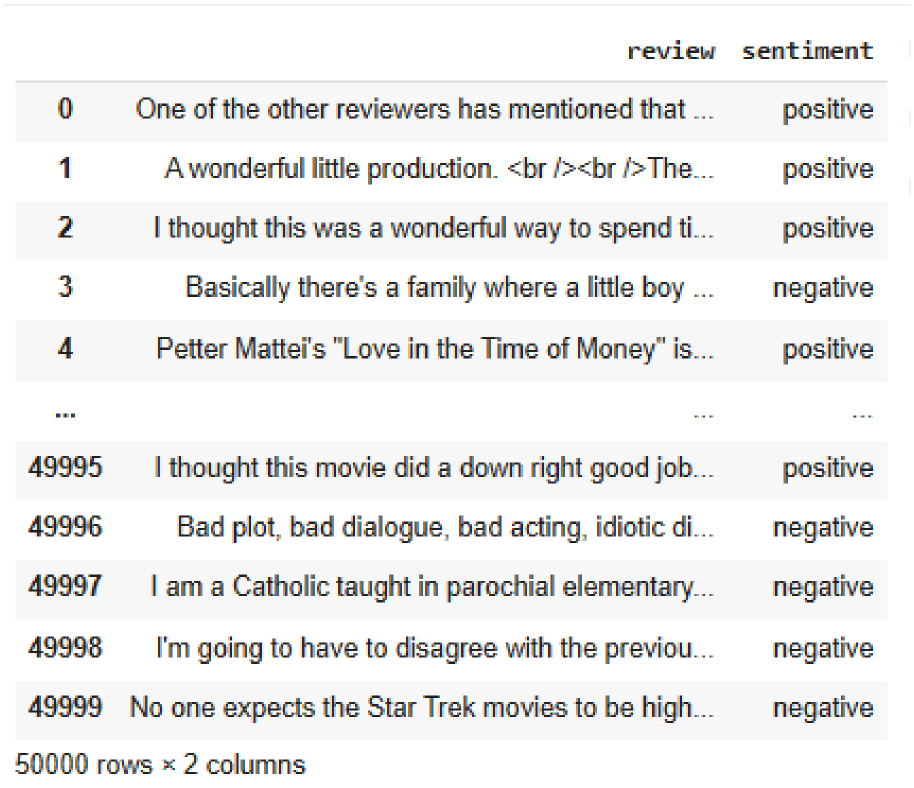
A crucial feature of this dataset is its balanced class distribution, the dataset contains 25.000 positive reviews and 25.000 negative reviews. This balance prevents the model from developing a bias towards the majority class and allows for a more accurate assessment of its performance.

Each sample in the dataset consists of two main fields: the raw text of the movie review and a corresponding label (e.g., 0 for negative and 1 for positive)

6.2 Data Preparation

6.2.1 Read the dataset

```python
df = pd.read_csv('./IMDB Dataset.csv')
print(f'Number of Null value: {df.isnull().sum().values.sum()}')
df
```



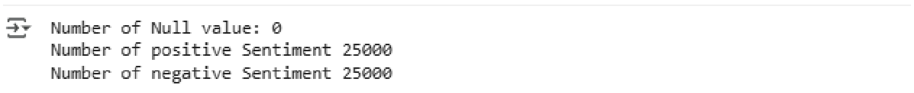
	review	sentiment
0	One of the other reviewers has mentioned that ...	positive
1	A wonderful little production. <br /><br />The...	positive
2	I thought this was a wonderful way to spend ti...	positive
3	Basically there's a family where a little boy ...	negative
4	Petter Mattei's "Love in the Time of Money" is...	positive
...	...	...
49995	I thought this movie did a down right good job...	positive
49996	Bad plot, bad dialogue, bad acting, idiotic di...	negative
49997	I am a Catholic taught in parochial elementary...	negative
49998	I'm going to have to disagree with the previou...	negative
49999	No one expects the Star Trek movies to be high...	negative

50000 rows × 2 columns

Figure 9: *IMDB Dataset*

6.2.2 Data Overview

```python
print(f'Number of Null value: {df.isnull().sum().values.sum()}')
print(f'Number of positive Sentiment {sum(df['sentiment'] == 'positive')}')
print(f'Number of negative Sentiment {sum(df['sentiment'] == 'negative')}')
```



```
Number of Null value: 0
Number of positive Sentiment 25000
Number of negative Sentiment 25000
```

Figure 10: *Data Overview*

6.2.3 Data spliting

```python
X = df['review'].values
y = df['sentiment'].values
y = [1 if label == 'positive' else 0 for label in y]
n_classes = len(set(y)) # n_classes = 2
```

6.3 Data Preprocessing for RNN and LSTM

Historically, recurrent architectures such as RNNs and LSTMs predominantly utilized **word-level tokenization**. The primary rationale was to minimize the sequence length of the input data; since these models process data sequentially, excessive sequence lengths exacerbate the vanishing gradient problem and impede temporal dependency learning. However, this approach necessitates a vast vocabulary size to cover the language distribution, resulting in high memory usage and a limited ability to handle Out-Of-Vocabulary (OOV) terms.

6.3.1 Data Denoising and Tokenization

Raw text data, such as the reviews from the IMDB dataset, is unstructured and contains a significant amount of noise. Before this text can be fed into a model either machine learning or deep learning, it must undergo a series of preprocessing steps. The goal of this pipeline is to clean, normalize and convert the text into a structured format of tokens (words or sub-words) that the model can understand and learn from. The following steps were systematically applied to each review in the dataset

Change to Lowercase

The first step in the normalization process is to convert the entire text to lowercase. This is a fundamental step that ensures uniformity. By converting all characters to their lowercase equivalents, we prevent the model from treating the same word with different capitalization as distinct tokens (e.g., treating *Movie*, *MOVIE*, and *movie* as the same word). This significantly reduces the vocabulary size and the complexity of the learning task, helping the model to generalize better.

Unicode Normalization

Text data sourced from the internet often contains various Unicode characters that may look similar but have different underlying representations (e.g., different types of quotes, dashes, or special characters). Unicode normalization is the process of converting all text into a standard, canonical form. This step ensures that words like *naïve* and *naive* are treated as the same token, preventing the model from learning redundant representations and reducing the overall vocabulary size.

Contraction Expansion

English text frequently uses contractions, such as *don't*, *I've*, or *can't*. These forms can be problematic for tokenization and sentiment analysis. For example, the word *don't* contains the negation *not*, which is critical for understanding sentiment. By expanding contractions (e.g., converting *don't* to *do not*), we make the sentiment-bearing words explicit and standardized. This process ensures that the model can correctly interpret the semantic meaning of the sentence without being confused by the abbreviated forms.

Remove HTML tags

Since the IMDB reviews are scraped from a website, they often contain residual HTML markup, such as `<br />` for line breaks or `<i>` for italics. These tags provide no semantic value for determining the sentiment of the text and act as noise for the model. This step involves using regular expressions libraries to parse the text and strip out all HTML tags, leaving only the clean, human-written content of the review.

Remove Punctuation

Punctuation marks like commas, periods, exclamation points, and question marks generally do not contribute to identifying which words are present in the text. While they can sometimes imply sentiment (e.g., `!!!`), they also create unnecessary variations of words (e.g., *movie* vs. *movie.*). For many models, removing punctuation helps to standardize the vocabulary. This is typically achieved by iterating through the text and removing any character that is classified as punctuation.

Remove Stopwords

Stopwords are extremely common words that appear frequently in a language but typically carry little semantic weight. Examples in English include *the*, *a*, *is*, *in*, and *on*. By removing these words, we allow the model to focus on the more meaningful words that are stronger indicators of sentiment (e.g., *amazing*, *terrible*, *boring*). However, this process must be handled with care in sentiment analysis. A standard stopword list might include negations like *not*, which are crucial for sentiment. Therefore, the stopword list used was curated to exclude such important negative contractions and words.

Stemming

Stemming is a text normalization process that reduces words to their root or base form, known as the "stem." For example, the words *running*, *runs*, and *ran* can all be reduced to the stem *run*. The purpose of stemming is to treat different inflections of a word as a single concept, which significantly reduces the vocabulary size and helps the model recognize the core meaning across different word forms. For this project, a standard algorithm like the Porter Stemmer was used to perform this transformation on each word after the preceding cleaning steps.

Tokenization

Tokenization is the final and crucial step in the preprocessing pipeline. After the text has been thoroughly cleaned and normalized, it is broken down from a single string into a sequence of individual units, or "tokens." In this project, tokenization was performed by splitting the cleaned text string by whitespace, resulting in a list of individual words (or stems). This process transforms the unstructured text into a structured format—a list of tokens—which is the required input for the subsequent vectorization stage, where the tokens are converted into numerical representations for the model.

```python
english_stop_words = stopwords.words('english')
english_stop_words.remove('not')
stemmer = PorterStemmer()
def text_tokenization(text):
  text = text.lower()
  #Unicode Normalization
  text = unidecode.unidecode(text)
  text = text.strip()
  #Contraction Expansion
  text = contractions.fix(text)
  #Remove HTML tags
  text = re.sub(r'<.*?>','',text)
  #Remove punctuation
  text = text.translate(str.maketrans('','',string.punctuation))
  tokens = text.split()
  #Remove stopwords
  text = ' '.join([word for word in tokens if word not in english_stop_words])
  #Stemming
  text = ' '.join([stemmer.stem(word) for word in text.split(' ')])
  #Tokenization
  tokens = text.split(' ')
  return tokens
corpus = [text_tokenization(text) for text in X]
```

6.3.2 Vocabulary Creation

After preprocessing and tokenizing the entire text corpus, the next critical step is to construct a vocabulary. A vocabulary is a fixed-size set of unique tokens that the model will recognize. This process is essential for converting the textual data into a numerical format, as the model can only process numbers. The vocabulary acts as a dictionary, mapping each unique token to a specific integer index.

```python
def create_vocabulary(corpus):
  vocab_freq = defaultdict(int)
  for tokens in corpus:
    for token in tokens:
      vocab_freq[token] +=1
  print(len(vocab_freq))
  sorted_vocab = sorted(vocab_freq.items(), key=lambda x: x[1], reverse=True)
  vocab = [word[0] for word in sorted_vocab[:10000]]
  vocab.append('<UNK>')
  vocab.append('<PAD>')
  return vocab
vocab = create_vocabulary(corpus)
```

The function `create_vocabulary` was implemented to perform this task systematically. Its primary goals are to identify the most relevant tokens in the corpus, manage the vocabulary size to balance performance and computational efficiency, and handle words that are not part of this set.
The process is broken down as follows:

1. **Token Frequency Counting**: The function first iterates through every review in the preprocessed corpus. For each review, it goes through every token and counts its occurrences. The result is a frequency distribution dictionary, where each token is a key and its total count across the entire dataset is the value.

2. **Sorting by Frequency**: To identify the most significant words, the tokens are sorted in descending order based on their frequency. The most common words (e.g., *film*, *movi*, *great*) will appear at the top of the list, while the least common words will be at the bottom.

3. **Vocabulary Pruning**: It is impractical and inefficient to include every single unique token in the vocabulary. Many tokens may have only appeared once or twice and are unlikely to provide generalizable information. Therefore, the vocabulary is pruned by selecting only the **top 10,000 most frequent tokens**. This is a common practice that helps reduce the dimensionality of the input data and allows the model to focus on the most impactful words.

4. **Adding Special Tokens**: Two special tokens are appended to the vocabulary to handle specific requirements of sequence modeling:

   - `<UNK>` **(Unknown)**: This token serves as a placeholder for any word that is encountered during training or inference but is not present in our top 10,000-word vocabulary. This ensures that the model can process any input text without errors, even if it contains out-of-vocabulary (OOV) words.
   - `<PAD>` **(Padding)**: Neural network models require input sequences to have a fixed, uniform length for batch processing. Since movie reviews naturally have varying lengths, this token is used to "pad" shorter sequences until they match the required length.

6.3.3 Text Vectorization

After creating a curated vocabulary, the final step in data preparation is vectorization. This is the process of converting the corpus of token lists into a numerical format that a neural network can process. Each review, currently a list of word strings, will be transformed into a list of integers of a fixed length. This was accomplished through a pipeline involving three key stages: establishing a sequence length, mapping tokens to integers, and standardizing the length of each sequence.

```
max_seq_len = int(np.percentile([len(tokens) for tokens in corpus],95) + 1)
word_2_idx = {word:idx for idx,word in enumerate(vocab)}
vocab_size = len(vocab)
def Corpus_vectorization(corpus,max_seq_len,word_2_idx):
  vectorize_corpus = []
  for tokens in corpus:
    vectorize_tokens = []
    for token in tokens:
      if len(vectorize_tokens) == max_seq_len:
        break
      if token not in word_2_idx:
        vectorize_tokens.append(word_2_idx['<UNK>'])
      else:
        vectorize_tokens.append(word_2_idx[token])
    while len(vectorize_tokens) < max_seq_len:
      vectorize_tokens.append(word_2_idx['<PAD>'])
    vectorize_corpus.append(vectorize_tokens)
  return vectorize_corpus
vectorize_corpus = Corpus_vectorization(corpus,max_seq_len,word_2_idx)
```

Determining Maximum Sequence Length

Neural networks require inputs to have a consistent shape for efficient batch processing. However, movie reviews have variable lengths. To address this, a fixed `max_seq_len` (maximum sequence length) must be defined.
A naive approach would be to find the longest review in the entire dataset and use its length. However, this is often inefficient, as a few extremely long reviews (outliers) would force the vast majority of shorter reviews to be padded excessively. A more pragmatic approach was used:

```
max_num_tokens = 0
for tokens in corpus:
  if max_num_tokens < len(tokens):
    max_num_tokens = len(tokens)
print(max_num_tokens) # 1428
max_seq_len = int(np.percentile([len(tokens) for tokens in corpus], 95) + 1)
print(max_seq_len) #  310
```

From this line of code, we can see that the 95th percentile of review lengths is 310 . This means that 95% of the reviews in the corpus are shorter than or equal to 310, while the longest review lengths is 1428. By choosing this value:

- We retain the complete contain for the vast majority (95%) of the reviews.
- We only truncate the longest 5% of reviews, minimizing information loss.
- We avoid the computational inefficiency of excessive padding

Word-to-Index Mapping

With the vocabulary established, a *word_2_idx* dictionary was created to serve as a lookup table. This dictionary maps each of the 10,002 unique tokens (including $<UNK>$ and $<PAD>$) to a unique integer index, from 0 to 10,001. This mapping is the core of the vectorization process, allowing for a direct conversion from a word to its numerical representation.

Corpus Vectorization and Standardization The *Corpus_vectorization* function iterates through each tokenized review in the corpus and performs the following operations to create the final numerical sequence:

1. **Token-to-Index Conversion:** Each token in the review is looked up in the *word_2_idx* dictionary. Its corresponding integer index is appended to a new list.
2. **Handling Out-of-Vocabulary (OOV) Words:** If a token is not found in the *word_2_idx* dictionary (meaning it was not one of the top 10,000 most frequent words), it is mapped to the index of the special $<UNK>$ token. This ensures that every word can be processed without error.
3. **Truncation and Padding:** The function enforces the *max_seq_len* on every review:
   - **Truncation:** If a review is longer than *max_seq_len*, the conversion process stops once the list of integers reaches this maximum length. Any remaining tokens in that review are discarded.
   - **Padding:** If a review is shorter than *max_seq_len*, the list of integers is padded by appending the index of the $<PAD>$ token until it reaches the required length.

The final output is a *vectorize_corpus*, which is a list of lists, where each inner list is a numerical representation of a movie review, perfectly standardized to the same length and ready to be used as input for the model.

6.4 Data Preprocessing for Transformer

In contrast, the Transformer architecture, which relies on parallelized Self-Attention mechanisms, is more robust to longer sequence lengths. Consequently, Transformer-based models adopt **subword-level tokenization** techniques, such as Byte Pair Encoding (BPE). This approach offers a superior trade-off: it significantly reduces the embedding matrix size by maintaining a compact vocabulary while effectively resolving the OOV issue. By decomposing rare or complex words into subword units, the model can better capture morphological nuances—such as sentiment-bearing prefixes and suffixes—which are critical for accurate Sentiment Analysis.

6.4.1 Byte Pair Encoding (BPE)

A critical challenge in Sentiment Analysis involves handling rare or out-of-vocabulary (OOV) words, which frequently contain significant emotional signals. In many cases, these OOV terms are morphological variations of common words, such as negated adjectives or complex adverbs. If these are treated as a generic `<UNK>` token, the model fails to capture the sentiment polarity, potentially leading to misclassification. To address this limitation, instead of using standard word-level tokenization, we employ Byte Pair Encoding (BPE). BPE is a subword segmentation algorithm based on the frequency of adjacent character pairs. It allows the model to decompose the vocabulary into meaningful subword units, thereby preserving semantic information even for unseen words. The core process is as follows:

1. Initialize the vocabulary by treating each word as a sequence of characters. For example: `unimpressed` → `[u, n, i, m, p, r, e, s, s, e, d]`.
2. Iteratively count the frequency of all adjacent character pairs across the training corpus.
3. Merge the most frequent pairs into new subword units. For instance, frequently occurring morphemes like `u` + `n` merge into the prefix `un`, and `e` + `d` merge into the suffix `ed`.
4. Repeat this process until a fixed vocabulary size is achieved.

The result is a subword vocabulary that allows the model to understand the composition of complex words. For example, the word `unimpressed`—which might be OOV in a word-level model—can be tokenized into `un` (negation prefix) + `impress` (root) + `ed`. This segmentation is crucial for sentiment analysis because it enables the model to recognize that the prefix `un-` reverses the positive polarity of the root *impress*, correctly identifying the term as negative.

```python
def text_clean(text):

    text = str(text)
    text = html.unescape(text)
    text = re.sub(r'<.*?>', ' ', text)
    text = unidecode.unidecode(text)
    text = contractions.fix(text)
    text = text.lower()
    text = re.sub(r'([.,!?])', r' \1 ', text)
    text = re.sub(r'[^a-z0-9\s.,!?\'-]', '', text)
    text = re.sub(r'\s+', ' ', text).strip()
    return text

X_clean = [text_clean(text) for text in X]

tokenizer = Tokenizer(BPE(unk_token="<UNK>"))
tokenizer.pre_tokenizer = Whitespace()

trainer = BpeTrainer(
    vocab_size=25000,
    min_frequency=2,
    special_tokens=["<PAD>", "<UNK>"]
)

tokenizer.train_from_iterator(X_clean, trainer)
```

```python
max_seq_len = 512
tokenizer.enable_padding(pad_id=pad_token_id, pad_token="<PAD>", length=
    max_seq_len)
tokenizer.enable_truncation(max_length=max_seq_len)
X_vec = [e.ids for e in tokenizer.encode_batch(X_clean)]
X_vec = np.array(X_vec)
```

VII Model Implementation

7.1 Recurrent Neural Network (RNN)

```python
class RNNClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_layers, n_classes,
     dropout):
        super(RNNClassifier, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.rnn = nn.RNN(input_size=embed_dim,
                          hidden_size=hidden_dim,
                          num_layers=num_layers,
                          batch_first=True,
                          dropout=dropout if num_layers > 1 else 0)
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(hidden_dim, n_classes)

    def forward(self, x):
        embedded = self.embedding(x)
        rnn_out, hidden = self.rnn(embedded)
        last_hidden = hidden[-1]
        out = self.dropout(last_hidden)
        logits = self.fc(out)
        return logits
```

The `RNNClassifier` class implements a standard Recurrent Neural Network (Vanilla RNN) for sentiment classification. The architecture processes sequential text data to produce a binary prediction.

- **Embedding Layer:** The `nn.Embedding` layer serves as the entry point, transforming integer token indices from the vocabulary into dense, continuous vector representations of size `embed_dim`. This captures the semantic meaning of words in a learnable feature space.
- **RNN Layer:** The core of the model is the `nn.RNN` module. We set `batch_first=True` to handle input tensors of shape `(batch, seq, feature)`. Unlike the Transformer which uses attention mechanisms, the RNN processes the sequence step-by-step, maintaining a hidden state that acts as memory.
- **Hidden State Extraction:** The `forward` method retrieves the final hidden state (`hidden[-1]`) after the RNN has processed the entire sequence. This vector summarizes the semantic content of the review.
- **Classification Head:** The final representation is passed through a `nn.Dropout` layer to prevent overfitting, followed by a fully connected linear layer (`nn.Linear`) that projects the hidden features to the output class logits (Positive/Negative).

7.2 Long Short Term Memory (LSTM)

7.2.1 LSTM Analysis

```python
class LSTMClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_layers, dropout):
        super(LSTMClassifier,self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.lstm = nn.LSTM(embed_dim, hidden_dim, num_layers, batch_first=True,
 dropout=dropout)
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(hidden_dim, 2)
    def forward(self, x):
        embedded = self.embedding(x)
        lstm_out, (hidden, cell) = self.lstm(embedded)
        last_hidden = hidden[-1]
        dropped = self.dropout(last_hidden)
        output = self.fc(dropped)
        return output
```

The **LSTMClassifier** integrates the core LSTM mechanism with embedding and classification layers to form a complete model for sentiment classification tasks. It processes tokenized input sequences into binary predictions (e.g., positive/negative sentiment).

- **Embedding Layer:** The `nn.Embedding` converts integer token indices (from vocabulary size `vocab_size`) into dense vectors of dimension `embed_dim`, providing a continuous representation suitable for recurrent processing. This step maps discrete words to a learned feature space, capturing semantic similarities.
- **LSTM Layers:** The `nn.LSTM` module stacks `num_layers` LSTM units, with input size `embed_dim` and hidden size `hidden_dim`. The `batch_first=True` ensures input shape (batch, sequence, features). Dropout between layers (`dropout`) regularizes to prevent overfitting. The output includes sequence hidden states `lstm_out` and final states (`hidden, cell`), where `hidden[-1]` captures the aggregated sequence representation from the last layer.
- **Dropout and Classification Head:** An additional `nn.Dropout` layer applies regularization to the last hidden state, followed by a linear layer (`nn.Linear`) projecting to 2 output classes (e.g., binary sentiment). This head transforms the fixed-size representation into logits for softmax or cross-entropy loss.
- **Forward Pass:** The input `x` (token indices) flows through embedding, LSTM processing (yielding final hidden), dropout, and linear classification. No activation is applied post-linear, assuming logits for loss computation.

This architecture leverages LSTM's memory capabilities for sequential data, making it effective for sentiment analysis where context spans varying lengths. Stacking layers (`num_layers > 1`) allows hierarchical feature extraction, while dropout enhances generalization on noisy text datasets like IMDb.

7.3 Transformer

In this section, we present the implementation of a simplified Transformer Encoder model using `PyTorch`. The architecture consists of stacked `EncoderBlock` modules that apply self-attention and feed-forward layers, combined with positional encodings to capture sequential information.

7.3.1 Encoder Block Analysis

```
class EncoderBlock(nn.Module):
  def __init__(self,embed_dim,num_heads,dropout):
    super().__init__()
    self.attn = nn.MultiheadAttention(embed_dim,num_heads,dropout,batch_first=
 True)
    self.norm1 = nn.LayerNorm(embed_dim)
    self.dropout = nn.Dropout(dropout)
    self.norm2 = nn.LayerNorm(embed_dim)
    self.ffwd = nn.Sequential(
            nn.Linear(embed_dim, embed_dim * 2),
            nn.ReLU(),
            nn.Linear(embed_dim * 2, embed_dim)
        )
  def forward(self,x,src_key_padding_mask=None):
    skip = x
    x,_ = self.attn(x,x,x, key_padding_mask=src_key_padding_mask) #output,
   weight
    x = self.norm1(skip+self.dropout(x))
    skip = x
    x = self.ffwd(x)
    x = self.norm2(skip+self.dropout(x))
    return x
```

The **EncoderBlock** is the fundamental computational unit of the Transformer. Each block performs two key operations: *multi-head self-attention* and a *feed-forward network (FFN)*, both equipped with residual connections and layer normalization.

- **Multi-Head Attention:** Implemented using `nn.MultiheadAttention`, this layer enables each token to attend to other tokens in the sequence, capturing long-range dependencies and contextual meaning.
- **Residual Connection and Normalization:** The residual (skip) connection (`skip + x`) helps mitigate vanishing gradients and allows information to flow directly through the network. Layer normalization (`nn.LayerNorm`) stabilizes training by maintaining consistent input distributions.
- **Feed-Forward Network:** A simple linear layer (`nn.Linear`) acts as the feed-forward transformation applied independently to each token representation. This step enhances the model's capacity to learn complex feature interactions.
- **Block Output:** The output of the block is again normalized and passed on to the next layer, ensuring stable learning across multiple stacked layers.

7.3.2 Transformer Encoder Analysis

```
class TransformerEncoder(nn.Module):
  def __init__(self,vocab_size,max_seq_len,embed_dim,num_heads,num_blocks,
   dropout,n_classes, pad_idx):
    super().__init__()
    self.pad_idx = pad_idx
    self.embed_matrix = nn.Embedding(num_embeddings=vocab_size,embedding_dim=
   embed_dim, padding_idx=pad_idx)

    self.register_buffer('positional_encoding', self._create_pos_encoding(
   max_seq_len, embed_dim))
    self.dropout = nn.Dropout(dropout)
    self.encoder_layers = nn.ModuleList([])
    for i in range(num_blocks):
      self.encoder_layers.append(EncoderBlock(embed_dim,num_heads,dropout))
    self.ffwd = nn.Linear(embed_dim,embed_dim*2)
    self.relu = nn.ReLU()
    self.out = nn.Linear(embed_dim*2,n_classes)

  def _create_pos_encoding(self, max_len, embed_dim):
    pe = torch.zeros(max_len, embed_dim)
    position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
    div_term = torch.exp(torch.arange(0, embed_dim, 2).float() * (-math.log
   (10000.0) / embed_dim))
    pe[:, 0::2] = torch.sin(position * div_term)
    pe[:, 1::2] = torch.cos(position * div_term)
    return pe.unsqueeze(0)

  def forward(self,x):
    # BxS
    padding_mask = (x == self.pad_idx)
    seq_len = x.size(1)
    x = self.embed_matrix(x)* math.sqrt(self.embed_matrix.embedding_dim)
    x = x + self.positional_encoding[:, :seq_len, :]
    x = self.dropout(x)
    for layer in self.encoder_layers:
      x = layer(x, src_key_padding_mask=padding_mask)
    mask_float = (~padding_mask).float().unsqueeze(-1)
    x = x*mask_float
    sum_embedings = torch.sum(x, dim=1)
    count_tokens = torch.sum(mask_float, dim=1).clamp(min=1)
    mean_pooling = sum_embedings/ count_tokens
    x = self.ffwd(mean_pooling)
    x = self.relu(x)
    return self.out(x)
```

The **TransformerEncoder** combines multiple encoder blocks to form the full model architecture. It adds embedding, positional encoding, and classification layers to make the model suitable for text processing tasks.

- **Embedding Layer:** The `nn.Embedding` layer converts discrete token indices into dense continuous vectors (`embed_matrix`), scaled by the square root of the embedding dimension to maintain variance stability.
- **Positional Encoding:** Since the Transformer architecture has no inherent notion of order, sinusoidal positional encodings are added to embeddings. These encodings use sine and cosine functions at different frequencies, allowing the model to infer token order and relative distance.
- **Stacked Encoder Layers:** The model constructs a sequence of `EncoderBlock` instances (as defined above) using `nn.ModuleList`, where each block refines the feature representation of the input sequence.
- **Sequence Aggregation:** After processing through all encoder layers, token representations are averaged along the sequence dimension (`mean(dim=1)`). This operation creates a fixed-size vector summarizing the entire sequence.
- **Classification Head:** A two-layer feed-forward network (`Linear → ReLU → Linear`) is used to transform the aggregated representation into the final class logits (`n_classes` output units).

VIII Model Training

8.1 Training and Evaluation Function

```
def evaluate(model, val_loader, criterion, device):
    model.eval()
    batch_val_loss = []
    all_preds = []
    all_targets = []

    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            batch_val_loss.append(criterion(outputs, labels).item())
            preds = torch.argmax(outputs, dim=1)
            all_preds.extend(preds.cpu().numpy())
            all_targets.extend(labels.cpu().numpy())
    epoch_val_loss = sum(batch_val_loss) / len(batch_val_loss)
    val_acc = np.mean(np.array(all_preds) == np.array(all_targets))
    val_f1 = f1_score(all_targets, all_preds, average='binary')
    val_precision = precision_score(all_targets, all_preds, average='binary')
    val_recall = recall_score(all_targets, all_preds, average='binary')

```

```python
    return epoch_val_loss, val_acc, val_f1, val_precision, val_recall

def fit(model, criterion, optimizer, train_loader, val_loader, num_epochs, device, save_path="/content/drive/MyDrive/Colab Notebooks/ ATH -IMDB"):
    epoch_train_loss = []
    epoch_train_acc = []
    epoch_val_loss = []
    epoch_val_acc = []

    best_val_loss = 1000
    early_stop_count = 0
    patience = 10

    for epoch in range(num_epochs):
        model.train()
        batch_train_loss = []
        batch_train_acc = []
        count = 0

        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1)
            optimizer.step()

            batch_train_loss.append(loss.item())
            batch_train_acc.append(torch.sum(torch.argmax(outputs, dim=1) == labels).item())
            count += len(labels)

        train_loss = sum(batch_train_loss) / len(batch_train_loss)
        train_acc = sum(batch_train_acc) / count
        val_loss, val_acc, val_f1, val_prec, val_rec = evaluate(model, val_loader, criterion, device)
        epoch_train_loss.append(train_loss)
        epoch_train_acc.append(train_acc)
        epoch_val_loss.append(val_loss)
        epoch_val_acc.append(val_acc)
        print(f'Epoch {epoch+1}/{num_epochs}:')
        print(f'  Train Loss: {train_loss:.4f} | Val Loss: {val_loss:.4f}')
        print(f'  Train Acc:  {train_acc:.4f} | Val Acc:  {val_acc:.4f}')
        print(f'  Val F1:     {val_f1:.4f}   | Val P:    {val_prec:.4f} | Val R: {val_rec:.4f}')
        print("-" * 50)

        if val_loss < best_val_loss:
            best_val_loss = val_loss
            early_stop_count = 0
            torch.save(model.state_dict(), os.path.join(save_path, 'best_model.pt'))
        else:
            early_stop_count += 1

        if early_stop_count == patience:
            break

    model.load_state_dict(torch.load(os.path.join(save_path, 'best_model.pt'), map_location=device))

    return epoch_train_loss, epoch_train_acc, epoch_val_loss, epoch_val_acc
```

The code snippet above implements the two core components of the model development process: performance evaluation (`evaluate`) and model training (`fit`).

8.1.1 Evaluation Function

The `evaluate` function is responsible for assessing the model's performance on the validation set without altering its parameters.

- **Inference Mode:** The command `model.eval()` is invoked initially to switch the model to inference mode. This step is critical for disabling specific layers that behave differently during training, such as Dropout and Batch Normalization.
- **Resource Optimization:** The `with torch.no_grad():` context manager ensures that PyTorch disables gradient calculation and tracking. This significantly reduces memory consumption and accelerates computation during the validation phase.
- **Performance Metrics:** The function computes the average loss and calculates key classification metrics—including Accuracy, F1-Score, Precision, and Recall—providing a comprehensive overview of the model's generalization capability on unseen data.

8.1.2 Training Function (Fit)

The `fit` function manages the entire training lifecycle across multiple epochs.

- **Backpropagation and Optimization:** In each iteration (batch), the model performs the forward pass to compute the loss, executes `loss.backward()` to calculate gradients, and employs `optimizer.step()` to update the weights.
- **Gradient Clipping:** Notably, the command `torch.nn.utils.clip_grad_norm_(model.parameters(), 1)` is utilized to cap the gradient norm at 1.0. This technique is particularly vital when training NLP models (such as RNNs or Transformers) to prevent the "exploding gradient" phenomenon, ensuring more stable convergence.
- **Early Stopping Mechanism:** To mitigate overfitting, the function incorporates an Early Stopping strategy with a `patience` parameter set to 10. If the validation loss does not improve for 10 consecutive epochs, the training process is automatically terminated.
- **Model Checkpointing:** During training, the model state yielding the lowest validation loss is saved as `best_model.pt`. At the conclusion of the function, these optimal weights are reloaded into the model to ensure the returned instance represents the best-performing version, rather than the state at the final epoch.

8.2 Recurrent Neural Network

8.3 Recurrent Neural Network

```python
device = 'cuda' if torch.cuda.is_available() else 'cpu'
batch_size = 64
embed_dim = 128
hidden_dim = 128
num_layers = 1
n_classes = 2
dropout = 0.5
learning_rate = 0.001
num_epochs = 20

X_train, X_temp, y_train, y_temp = train_test_split(X_vectorized, y_vectorized,
    test_size=0.2)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5)
```

```

train_dataset = ImdbDataset(X_train, y_train)
val_dataset = ImdbDataset(X_val, y_val)
test_dataset = ImdbDataset(X_test, y_test)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)
test_loader = DataLoader(test_dataset, batch_size=batch_size)

model = RNNClassifier(
    vocab_size=vocab_size,
    embed_dim=embed_dim,
    hidden_dim=hidden_dim,
    num_layers=num_layers,
    n_classes=n_classes,
    dropout=dropout
).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

model, t_loss, t_acc, v_loss, v_acc = fit(
    model, criterion, optimizer, train_loader, val_loader,
    num_epochs, device, save_path=save_path
)
```

This section outlines the training configuration specific to the Recurrent Neural Network model:

- **Device Configuration:** The model is trained on GPU if available (`'cuda'`), otherwise on CPU, ensuring efficient computation for large datasets.
- **Hyperparameters:** We utilize an embedding size and hidden dimension of 128. Critically, we employ a single RNN layer (`num_layers=1`). As discussed in the theoretical section, simple RNNs struggle with the vanishing gradient problem; stacking multiple layers often exacerbates this issue without improving performance on this dataset. A dropout rate of 0.5 is applied to the final hidden state to enhance generalization.
- **Optimization:** The model is optimized using the **Adam** optimizer with a learning rate of 0.001. Adam is chosen for its adaptive learning rate capabilities, which typically yield faster convergence compared to standard SGD. The **Cross-Entropy Loss** function is used to measure the discrepancy between the predicted logits and the true sentiment labels.
- **Training Procedure:** The training is executed for 20 epochs using the shared `fit` function. This function manages the epoch loops, backpropagation, and gradient clipping (set to 2.0 to prevent exploding gradients), while also evaluating the model on the validation set after each epoch.

8.4 Long Short Term Memory

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
batch_size = 32
embed_dim = 128
hidden_dim = 128
num_layers = 2
dropout = 0.2
num_epochs = 8
X_train, X_test, y_train, y_test = train_test_split(vectorize_corpus, y,
    test_size=0.1)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size
    =0.1)
train_dataset = ImdbDataset(X_train,y_train)
val_dataset = ImdbDataset(X_val, y_val)
```

```
test_dataset = ImdbDataset(X_test, y_test)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64)
test_loader = DataLoader(test_dataset, batch_size=64)

lstm_model = LSTMClassifier(
    vocab_size=vocab_size,
    embed_dim=embed_dim,
    hidden_dim=hidden_dim,
    num_layers=num_layers,
    dropout=dropout,
).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(lstm_model.parameters(), lr=0.001)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.9)

epoch_train_loss, epoch_train_acc, epoch_val_loss, epoch_val_acc = fit(
    lstm_model, criterion, optimizer, scheduler, train_loader, val_loader,
    num_epochs, device)
```

This section defines hyperparameters and prepares the dataset for model training:

- **Device Configuration:** The model is trained on GPU if available ('cuda'), otherwise on CPU, ensuring efficient computation for large datasets.
- **Hyperparameters:** Defines embedding dimension (128), hidden dimension (128), number of layers (2), dropout rate (0.2), and epochs (8). These values balance model complexity with training speed for sentiment tasks.
- **Dataset Splitting:** The preprocessed corpus (vectorize_corpus) and labels (y) are split into training (80%), validation (10%), and testing (10%) sets using train_test_split for robust evaluation.
- **DataLoader:** Converts dataset objects into iterable batches: shuffled training batches of size 32 for stochastic updates, and larger validation/testing batches of size 64 for efficient inference.
- **Model and Optimization:** The model is initialized using the LSTMClassifier architecture and trained with Cross-Entropy loss (nn.CrossEntropyLoss), suitable for binary classification. The Adam optimizer is employed with a learning rate of 0.001 to adaptively adjust parameters. A StepLR scheduler is defined to decay the learning rate by a factor of 0.9 every 3 epochs, helping to fine-tune the model's parameters and potentially achieve better convergence.

8.5 Transformer

```
device = 'cuda' if torch.cuda.is_available else 'cpu'
batch_size = 64
embed_dim = 256
num_heads = 4
num_blocks = 2
dropout=0.1
num_epochs = 10
learning_rate = 5e-4
X_train,X_test,y_train,y_test = train_test_split(X_vec,y,test_size=0.2)
X_test,X_val,y_test,y_val = train_test_split(X_test,y_test,test_size=0.1)
train_dataset = ImdbDataset(X_train,y_train)
val_dataset = ImdbDataset(X_val,y_val)
test_dataset = ImdbDataset(X_test,y_test)
train_loader = DataLoader(train_dataset,batch_size,shuffle=True)
val_loader = DataLoader(val_dataset,batch_size)
test_loader = DataLoader(test_dataset,batch_size)
```

```python
model = TransformerEncoder(vocab_size,max_seq_len,embed_dim,num_heads,num_blocks,dropout,n_classes,pad_token_id).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),lr=learning_rate)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.9)
epoch_train_loss,epoch_train_acc,epoch_val_loss,epoch_val_acc = fit(model,criterion,optimizer,
                                                                    train_loader,val_loader,num_epochs,device)
```

This section defines hyperparameters and prepares the dataset for model training:

- **Device Configuration:** The model is configured to utilize the GPU if available ('cuda'); otherwise, it defaults to the CPU. This ensures efficient computation, particularly for the matrix operations required by the Transformer architecture.
- **Hyperparameters:** Key parameters are established to control model complexity: embedding dimension (256), number of attention heads (4), number of transformer blocks (2), and a dropout rate (0.1). The training is set to run for 10 epochs with a learning rate of 5×10^{-4}.
- **Dataset Splitting:** The vectorized data (X_vec) and labels (y) undergo a two-stage split using `train_test_split`. First, the data is divided into training (80%) and a temporary test set (20%). This temporary set is further split to create a final validation set and testing set, ensuring independent data for tuning and evaluation.
- **DataLoader:** The `ImdbDataset` objects are wrapped into `DataLoader` instances. A batch size of 64 is used for all sets. Notably, the training loader enables shuffling to ensure stochastic gradient updates, while validation and test loaders process data sequentially.
- **Model and Optimization:** The model is initialized using the `TransformerEncoder` architecture and trained with `CrossEntropyLoss`, which is standard for categorical classification. The `Adam` optimizer is employed to adaptively adjust weights. A `StepLR` scheduler is defined to decay the learning rate by a factor of 0.9 every 3 epochs, though it is not explicitly passed to the current `fit` function call.

IX Model Evaluation

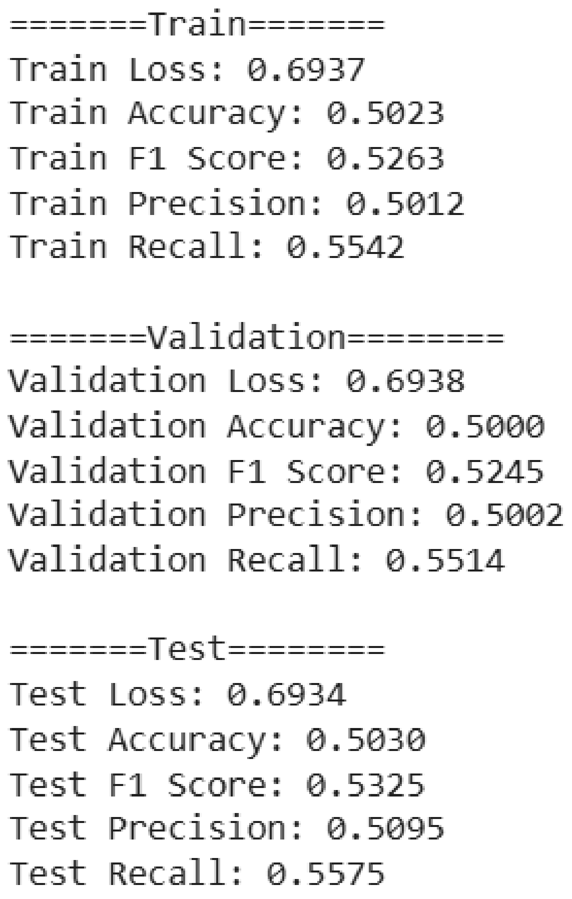
This code segment demonstrates the procedure for restoring the best-performing model state and conducting a comprehensive evaluation across all data splits (Training, Validation, and Testing).

- **Model Restoration:** A new instance of the model(RNN, LSTM, Transformer) is initialized with the original hyperparameters. The optimal weights, saved as 'best_model.pt' during the training phase, are loaded into this instance using `torch.load` with `weights_only=True` for security. This ensures that the evaluation is performed on the specific version of the model that minimized validation loss.
- **Holistic Assessment:** Unlike a simple test run, this procedure triggers the `evaluate` function on the `train_loader`, `val_loader`, and `test_loader` sequentially.
- **Generalization Analysis:** By computing and printing metrics (Loss, Accuracy, F1, Precision, Recall) for all three subsets simultaneously, we can perform a direct comparative analysis. This allows us to rigorously check for overfitting (e.g., if Training Accuracy significantly exceeds Validation Accuracy) and verify the model's stability before final deployment.

```python
model = TransformerEncoder(vocab_size,max_seq_len,embed_dim,num_heads,num_blocks,dropout,n_classes,pad_token_id).to(device)
model.load_state_dict(torch.load(os.path.join("/content/drive/MyDrive/Colab Notebooks/DATH-IMDB",'best_model.pt'),weights_only=True))
train_loss, train_acc, train_f1, train_precision, train_recall = evaluate(model,train_loader,criterion,device)
```

```python
print('=======Train======')
print(f'Train Loss: {train_loss}')
print(f'Train Accuracy: {train_acc}')
print(f'Train F1 Score: {train_f1}')
print(f'Train Precision: {train_precision}')
print(f'Train Recall: {train_recall}\n')
print('=======Validation=======')
val_loss, val_acc, val_f1, val_precision, val_recall = evaluate(model,val_loader
    ,criterion,device)
print(f'Validation Loss: {val_loss}')
print(f'Validation Accuracy: {val_acc}')
print(f'Validation Accuracy: {val_acc}')
print(f'Validation F1 Score: {val_f1}')
print(f'Validation Precision: {val_precision}')
print(f'Validation Recall: {val_recall}\n')
print('=======Test=======')
test_loss, test_acc, test_f1, test_precision, test_recall = evaluate(model,
    test_loader,criterion,device)
print(f'Test Loss: {test_loss}')
print(f'Test Accuracy: {test_acc}')
print(f'Test F1 Score: {test_f1}')
print(f'Test Precision: {test_precision}')
print(f'Test Recall: {test_recall}')
```

9.1 Recurrent Neural Network



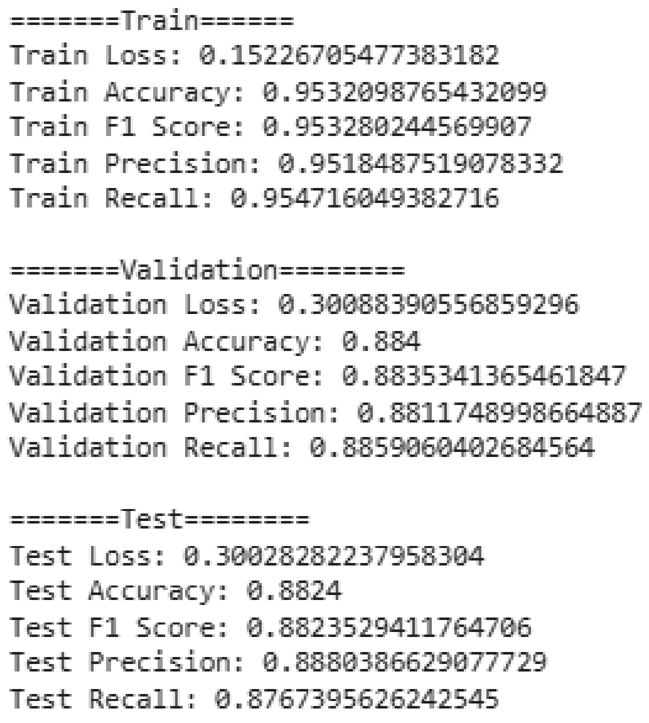
```
=======Train=======
Train Loss: 0.6937
Train Accuracy: 0.5023
Train F1 Score: 0.5263
Train Precision: 0.5012
Train Recall: 0.5542

=======Validation========
Validation Loss: 0.6938
Validation Accuracy: 0.5000
Validation F1 Score: 0.5245
Validation Precision: 0.5002
Validation Recall: 0.5514

=======Test========
Test Loss: 0.6934
Test Accuracy: 0.5030
Test F1 Score: 0.5325
Test Precision: 0.5095
Test Recall: 0.5575
```

Figure 11: *Evaluation metrics on the IMDB test set (RNN)*

The quantitative performance of the proposed Recurrent Neural Network (RNN) model was recorded across Training, Validation, and Testing phases. The detailed metrics are summarized as follows:

- **Generalization Capability:** A comparison between the Training and Validation sets reveals that the model failed to capture the underlying patterns in the data. The Validation Accuracy (50.00%) and Training Accuracy (50.23%) are nearly identical and equivalent to random guessing. Furthermore, the Loss values for both sets remained stagnant around 0.69 (approximately ln(2)), indicating that the model weights did not converge significantly from their initial random states. This behavior is a characteristic symptom of the *Vanishing Gradient* problem inherent in Vanilla RNNs, where the model struggles to learn dependencies in long sequences such as IMDB reviews.
- **Test Set Performance:** On the independent Test set, the model achieved an Accuracy of 50.30% and a Loss of 0.6934. These results mirror the training phase, confirming that the simple RNN architecture lacks the capacity to generalize or make meaningful predictions on unseen real-world data for this specific task.
- **Precision-Recall Trade-off:** The evaluation metrics show a balanced but ineffective performance:
  - **Recall (55.75%):** The model identifies slightly more than half of the positive samples, but this is likely due to a tendency to predict one class more frequently rather than true discriminative power.
  - **Precision (50.95%):** The precision is close to 50%, implying that when the model predicts a review as positive, it is correct only about half the time.
- **Overall Effectiveness:** With an F1-Score of 0.5325, the model demonstrates poor reliability. The inability of the F1-Score to surpass the baseline significantly suggests that the Vanilla RNN architecture, without gating mechanisms (like LSTM or GRU), is unsuitable for processing the long-term dependencies present in the IMDB sentiment analysis dataset.

9.2 Long Short Term Memory



```
=======Train======
Train Loss: 0.15226705477383182
Train Accuracy: 0.9532098765432099
Train F1 Score: 0.953280244569907
Train Precision: 0.9518487519078332
Train Recall: 0.954716049382716

=======Validation========
Validation Loss: 0.30088390556859296
Validation Accuracy: 0.884
Validation F1 Score: 0.8835341365461847
Validation Precision: 0.8811748998664887
Validation Recall: 0.8859060402684564

=======Test========
Test Loss: 0.30028282237958304
Test Accuracy: 0.8824
Test F1 Score: 0.8823529411764706
Test Precision: 0.8880386629077729
Test Recall: 0.8767395626242545
```

Figure 12: *Evaluation metrics on the IMDB test set (LSTM)*

The quantitative performance of the proposed LSTM-based model was recorded across Training, Validation, and Testing phases. The detailed metrics are summarized as follows:

- **Generalization Behavior:** The model achieves strong in-sample performance with a Training Accuracy of 95.32% and a Training Loss of 0.152. However, the Validation Accuracy drops to 88.40% with a higher Loss of 0.301. This gap suggests mild-to-moderate overfitting, which is not uncommon for recurrent models on long text sequences due to their high capacity and sequential processing constraints. Nevertheless, the close alignment between Validation and Test results indicates that generalization on unseen data remains stable.
- **Test Set Robustness:** On the independent Test set, the model yields an **Accuracy of** 88.24% and a **Loss of** 0.300, closely matching the Validation performance. This consistency confirms that the LSTM retains reliable predictive power beyond the training distribution and does not exhibit severe instability across evaluation splits.
- **Precision–Recall Characteristics:** The evaluation metrics reveal a relatively balanced decision boundary:
  - **Validation:** Precision = 88.12%, Recall = 88.59%, indicating near-symmetric error behavior.
  - **Test:** Precision = 88.80% exceeds Recall = 87.67%, suggesting the classifier is slightly more conservative in predicting positive sentiment, producing fewer False Positives at the cost of a small increase in False Negatives.
- **Overall Effectiveness:** The model obtains an **F1-Score** of 0.9533 on Training, 0.8835 on Validation, and **0.8824 on Testing**. The latter two values reflect a stable and reasonably balanced performance profile suitable for sequence-based sentiment classification. While the observed train–validation gap implies room for improvement via stronger regularization (e.g., higher dropout, early stopping, or reduced hidden size), the LSTM remains a competitive baseline that effectively captures sequential sentiment cues such as negations and contrastive discourse.

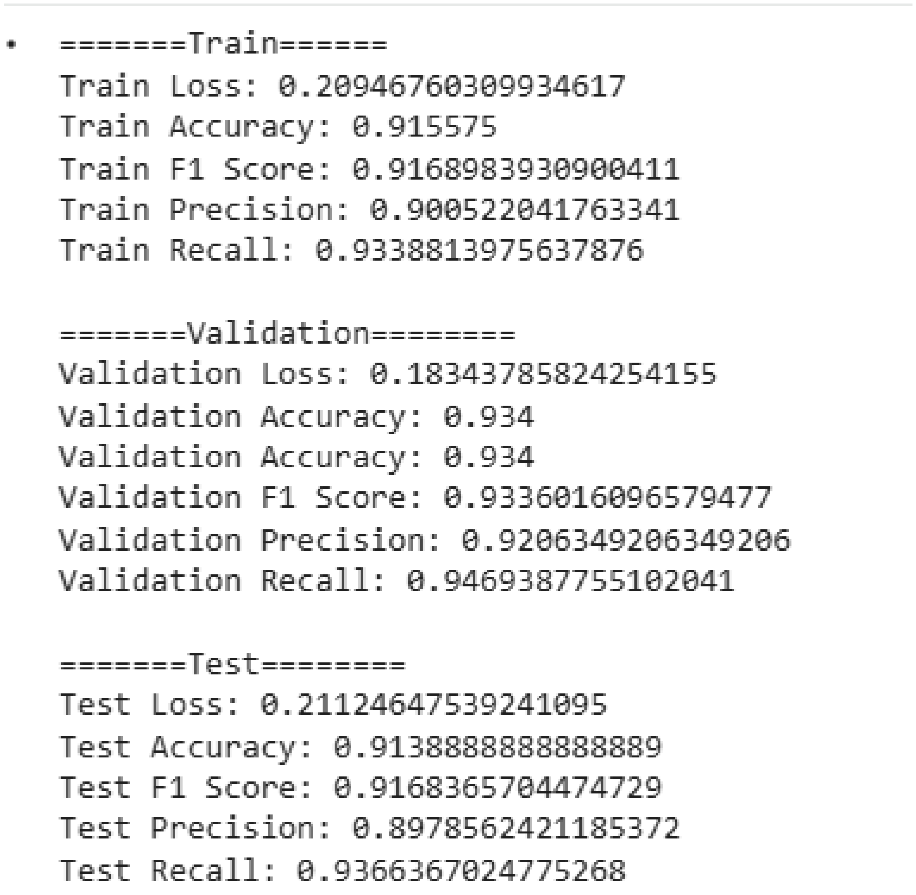
9.3 Transformer

The quantitative performance of the proposed Transformer-based model was recorded across Training, Validation, and Testing phases. The detailed metrics are summarized as follows:

- **Generalization Capability:** A comparison between the Training and Validation sets reveals highly favorable behavior. The Validation Accuracy (93.4%) and Loss (0.183) are slightly better than the Training Accuracy (91.6%) and Loss (0.209). This pattern—often attributed to the use of Dropout regularization which is active during training but disabled during inference—strongly indicates that the model has not overfitted to the training data and possesses excellent generalization capabilities.
- **Test Set Performance:** On the independent Test set, the model achieved a robust **Accuracy of 91.39%** and a **Loss of 0.211**. This confirms that the model's performance remains consistent when applied to completely unseen real-world data.
- **Precision-Recall Trade-off:** The evaluation metrics highlight a specific characteristic of the model's decision boundary:
  - **Recall** (93.66%)**:** The model exhibits high sensitivity, meaning it is very effective at detecting positive reviews and rarely misses them (low False Negative rate).
  - **Precision** (89.79%)**:** While still high, the precision is slightly lower than recall, suggesting that the model is occasionally "optimistic," classifying some neutral or ambiguous reviews as positive.
- **Overall Effectiveness:** With an **F1-Score of 0.9168**, which represents the harmonic mean of Precision and Recall, the model demonstrates a balanced and reliable performance suitable for automated sentiment classification tasks.

X Future Work

Based on the findings and limitations of this study, several avenues for future research are proposed:



```
=======Train======
Train Loss: 0.20946760309934617
Train Accuracy: 0.915575
Train F1 Score: 0.9168983930900411
Train Precision: 0.900522041763341
Train Recall: 0.9338813975637876

=======Validation========
Validation Loss: 0.18343785824254155
Validation Accuracy: 0.934
Validation Accuracy: 0.934
Validation F1 Score: 0.9336016096579477
Validation Precision: 0.9206349206349206
Validation Recall: 0.9469387755102041

=======Test========
Test Loss: 0.21124647539241095
Test Accuracy: 0.9138888888888889
Test F1 Score: 0.9168365704474729
Test Precision: 0.8978562421185372
Test Recall: 0.9366367024775268
```

Figure 13: *Evaluation metrics on the IMDB test set (Transformer)*

1. **Fine-grained Sentiment Analysis:** Currently, the model performs binary classification (Positive vs. Negative). Future iterations will extend the architecture to handle multi-class classification (e.g., predicting 1-to-5 star ratings) to capture more granular sentiment nuances.

2. **Integration of Pre-trained Models:** While the custom Transformer encoder demonstrated strong performance, integrating large-scale pre-trained language models (PLMs) such as BERT or RoBERTa could further enhance accuracy via transfer learning, particularly for complex linguistic structures.

3. **Cross-Domain Evaluation:** To assess the model's generalization capabilities, future work will evaluate its performance on diverse datasets from different domains, such as e-commerce product reviews (Amazon) or informal social media text (Twitter/X).

XI Conclusion

In this study, we successfully developed and compared three distinct deep learning architectures—Recurrent Neural Networks (RNN), Long Short-Term Memory (LSTM), and the Transformer—for the task of Sentiment Analysis on the IMDB movie review dataset. The primary objective was to evaluate how these models handle the complexities of processing large-scale, unstructured textual data.
Our experimental results and theoretical analysis highlight the following key findings:

1. **Limitations of Recurrent Models:** While the RNN and LSTM models are capable of processing sequential data, they face inherent challenges with long sequences found in movie reviews. The Vanilla RNN is particularly susceptible to the vanishing gradient problem, limiting its ability to learn dependencies in long texts. Although the LSTM mitigates this through its gating mechanisms (forget, input, and output gates), it still suffers from sequential processing constraints that hinder parallelization and training speed.

2. **Superiority of the Transformer:** The Transformer architecture proved to be the most robust approach. By leveraging the **Multi-Head Attention** mechanism and **Positional Encoding**, the model successfully captured global dependencies and semantic nuances without the need for recurrence.

3. **Effectiveness of Subword Tokenization:** The implementation of **Byte Pair Encoding (BPE)** for the Transformer significantly improved the handling of Out-Of-Vocabulary (OOV) terms. This allowed the model to recognize morphological variations (e.g., *"unimpressed"* as *"un"* + *"impressed"*), which is critical for detecting sentiment polarity in complex negations.

4. **Quantitative Success:** The Transformer model achieved a testing accuracy of **91.39%** and an F1-Score of **0.9168**. The closeness of the validation accuracy (93.4%) and training accuracy (91.6%) demonstrates that the model, aided by Dropout regularization, generalized well to unseen data without significant overfitting.

In conclusion, while LSTM remains a viable option for sequence modeling, the Transformer architecture offers a superior trade-off between computational efficiency and semantic understanding for Sentiment Analysis. Future work may involve extending this system to multi-class classification (e.g., rating scales) or integrating pre-trained language models like BERT to further enhance performance.

Reference

1. U. Gupta, A. Chatterjee, R. Srikanth, and P. Agrawal, “A Sentiment-and-Semantics-Based Approach for Emotion Detection in Textual Conversations,”, 2017.

2. A. Vaswani et al., "Attention is all you need," in Advances in Neural Information Processing Systems, vol. 30, 2017

3. H. Deng and A. Alkhayyat, “Sentiment analysis using long short term memory and amended dwarf mongoose optimization algorithm,” *Journal of King Saud University - Computer and Information Sciences*, vol. 35, no. 8, p. 101664, 2023.