21CSE356T

Natural Language Processing

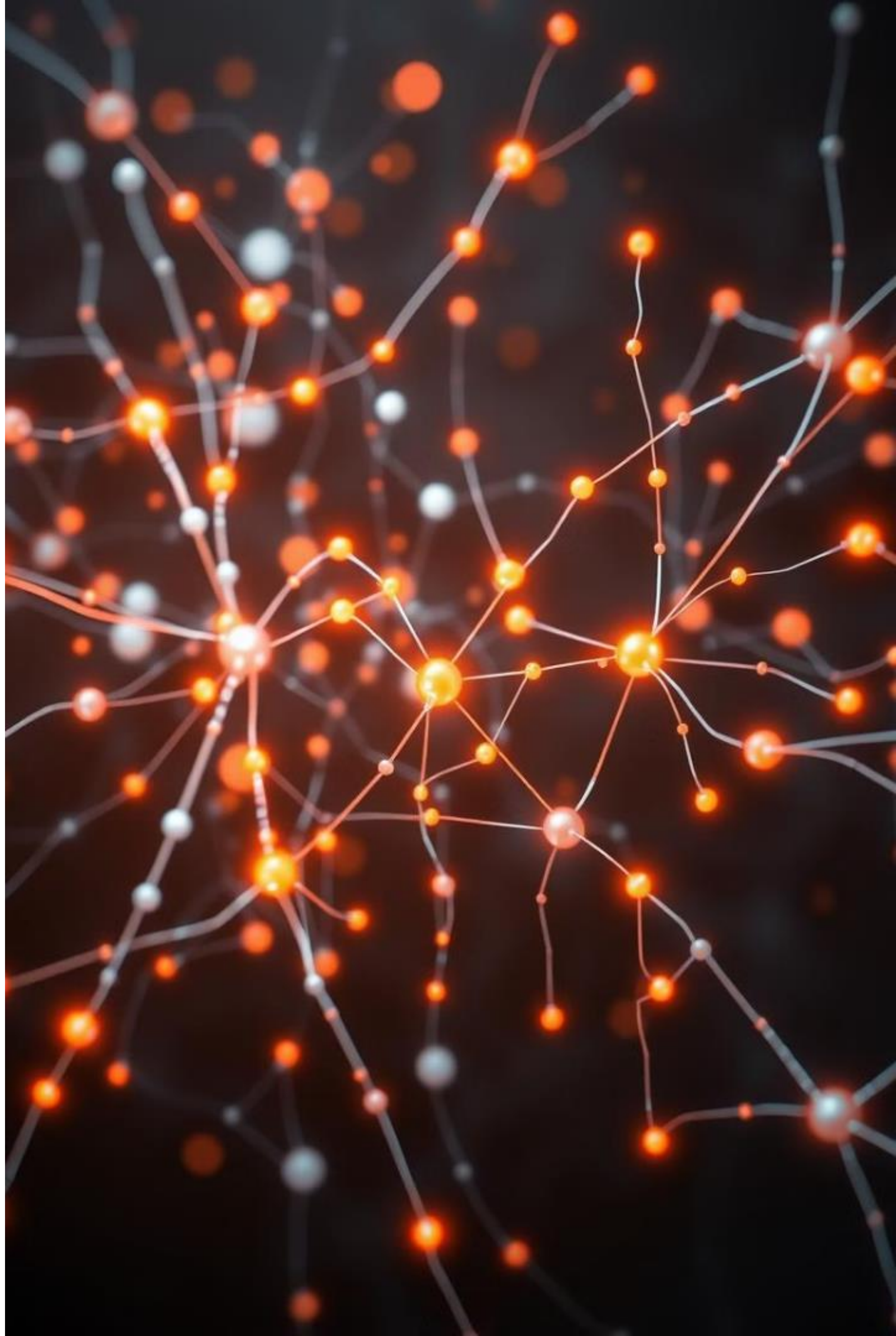
Dr. R. USHARANI
Assistant Professor
Dept. of Computational Intelligence
SRM Institute of Science & Technology
Kattankulathur 603 203.
Chennai.

Recurrent Neural Networks (RNNs)

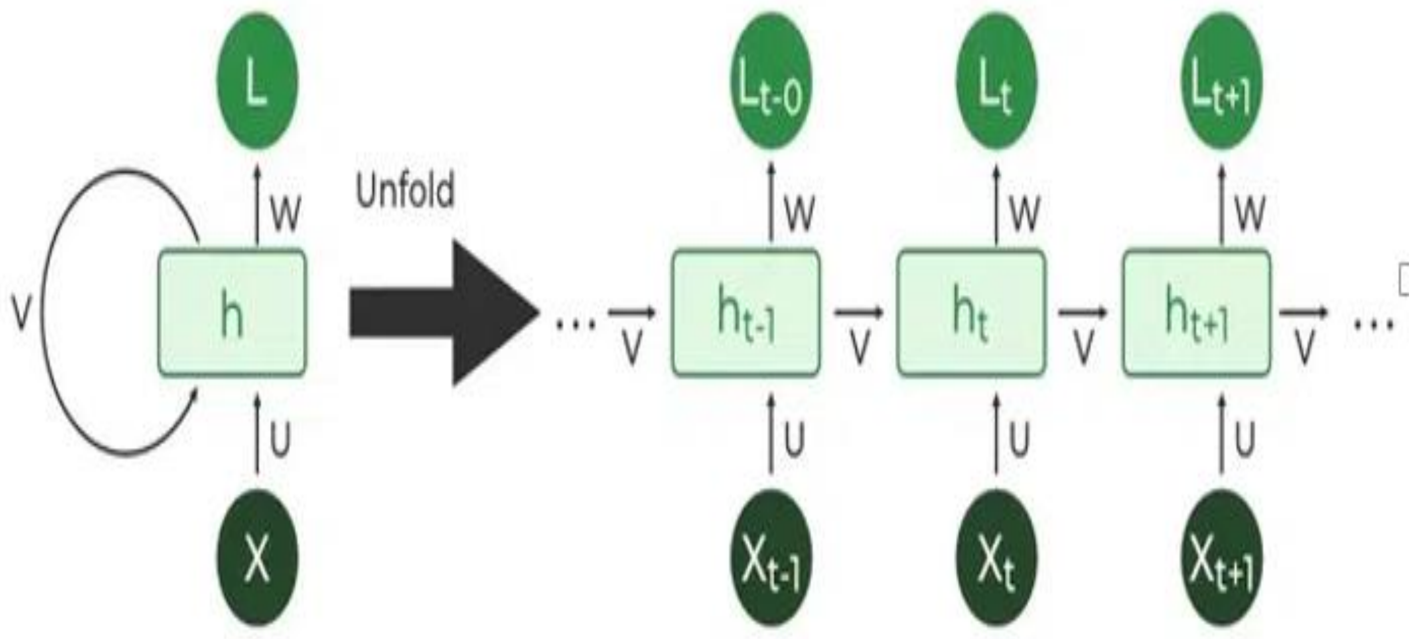
Recurrent Neural Networks (RNNs) are deep learning models designed to process sequential data by maintaining an internal memory state. Unlike traditional neural networks, RNNs capture dependencies across time steps, making them useful for time-series data, speech recognition, and natural language processing (NLP).

RNNs model sequential dependencies, where past information influences future predictions. They are widely applied in language modeling, translation, and handwriting recognition.

U **by Usharani R**

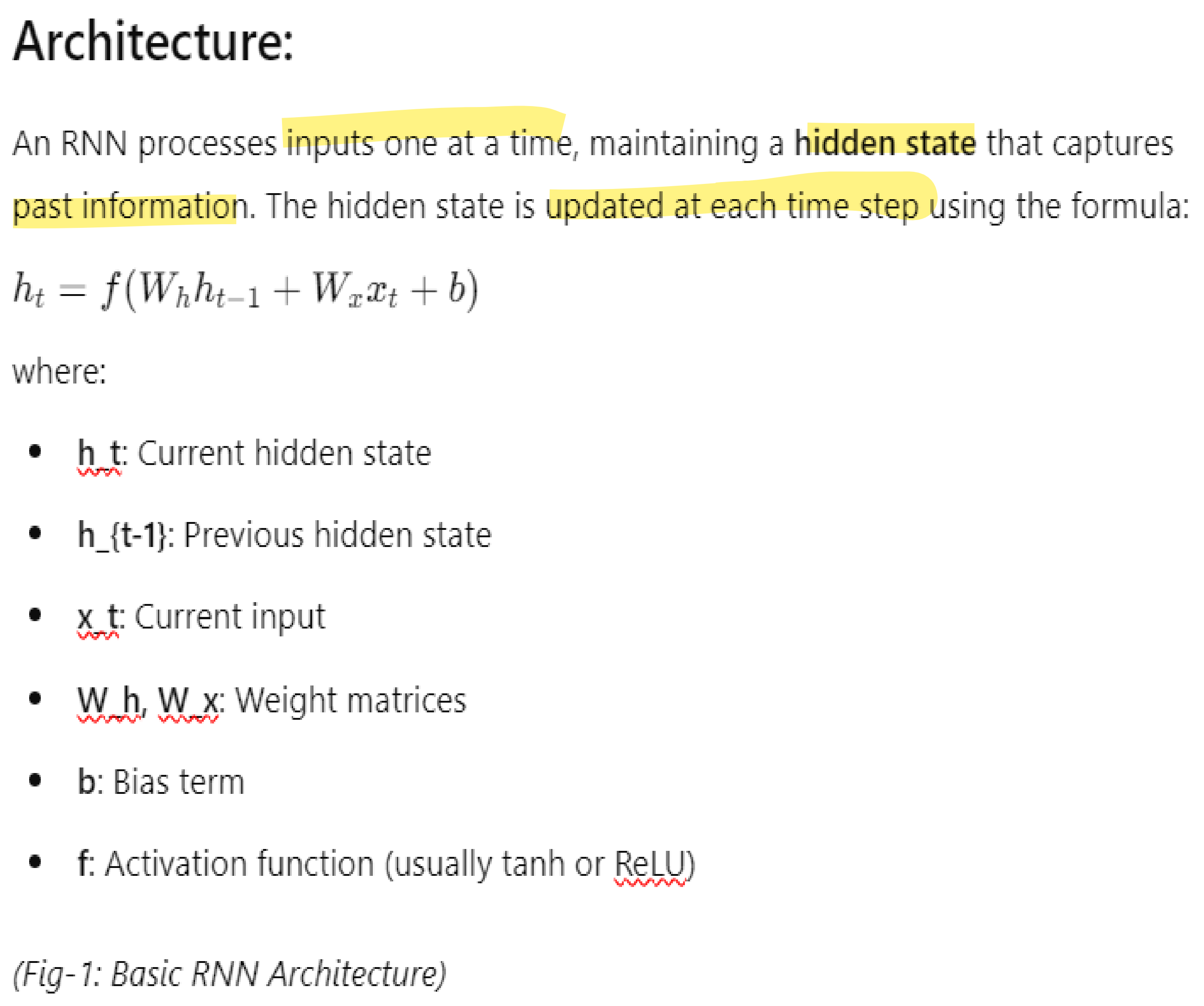


RNN Architecture



process sequential data
internal memory state.
captures dependencies using time steps
past information influences future predictions
egs-

input one at a time
maintain hidden state
captures past info
hidden state - updated at each time step
formula



Architecture:

An RNN processes inputs one at a time, maintaining a **hidden state** that captures past information. The hidden state is updated at each time step using the formula:

$$h_t = f(W_h h_{t-1} + W_x x_t + b)$$

where:

- h_t: Current hidden state
- h_{t-1}: Previous hidden state
- x_t: Current input
- W_h, W_x: Weight matrices
- b: Bias term
- f: Activation function (usually tanh or ReLU)

(Fig-1: Basic RNN Architecture)

RNN Architecture

How does RNN work?

At each time step RNNs process units with a fixed activation function. These units have an internal hidden state that acts as memory that retains information from previous time steps. This memory allows the network to store past knowledge and adapt based on new inputs.

Hidden State

At each time step, RNNs process units with a fixed activation function. These units have an internal hidden state that acts as memory, retaining information from previous time steps. This memory allows the network to store past knowledge and adapt based on new inputs.

State Update

The current hidden state depends on the previous state and the current input. The state is updated using weight matrices for both recurrent and input neurons, combined with an activation function to introduce non-linearity.

Updating the Hidden State in RNNs

The current hidden state ht depends on the previous state ht−1 and the current input xxt, and is calculated using the following relations:

Updating the Hidden State in RNNs

The current hidden state ht depends on the previous state ht−1 and the current input xxt, and is calculated using the following relations:

1. **State Update:**

$$h_t = f(h_{t-1}, x_t)$$

where:

- h_t is the current state
- h_{t-1} is the previous state
- x_t is the input at the current time step

2. **Activation Function Application:**

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

Here, W_{hh} is the weight matrix for the recurrent neuron, and W_{xh} is the weight matrix for the input neuron.

3. **Output Calculation:**

$$y_t = W_{hy} \cdot h_t$$

where y_t is the output and W_{hy} is the weight at the output layer.

Backpropagation Through Time (BPTT)

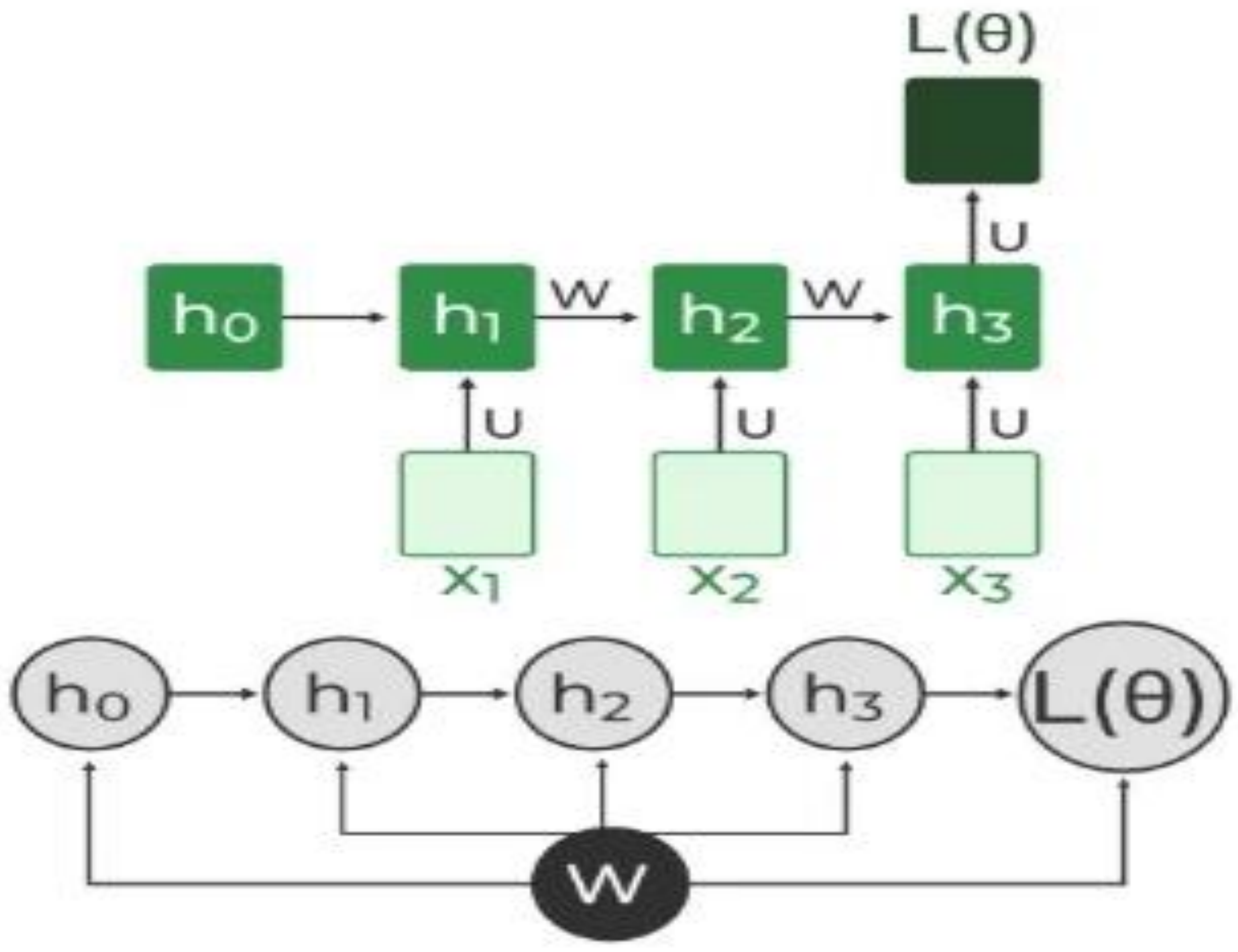
Sequential Dependency

In RNNs, the loss function depends on the final hidden state, and each hidden state relies on preceding ones, forming a sequential dependency chain.

Gradient Backpropagation

Backpropagation Through Time (BPTT) is used to update the network's parameters. Gradients are backpropagated through each time step, essential for updating network parameters based on temporal dependencies.

Backpropagation Through Time (BPTT)



Backpropagation Through Time (BPTT) In RNN

In BPTT, gradients are backpropagated through each time step. This is essential for updating network parameters based on temporal dependencies.

1. **Simplified Gradient Calculation:**

   $$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \cdot \frac{\partial h_3}{\partial W}$$

2. **Handling Dependencies in Layers:**

   Each hidden state is updated based on its dependencies:

   $$h_3 = \sigma(W \cdot h_2 + b)$$

   The gradient is then calculated for each state, considering dependencies from previous hidden states.

3. **Gradient Calculation with Explicit and Implicit Parts:** The gradient is broken down into explicit and implicit parts summing up the indirect paths from each hidden state to the weights.

   $$\frac{\partial h_3}{\partial W} = \frac{\partial h_3^+}{\partial W} + \frac{\partial h_3}{\partial h_2} \cdot \frac{\partial h_2^+}{\partial W}$$

4. **Final Gradient Expression:**

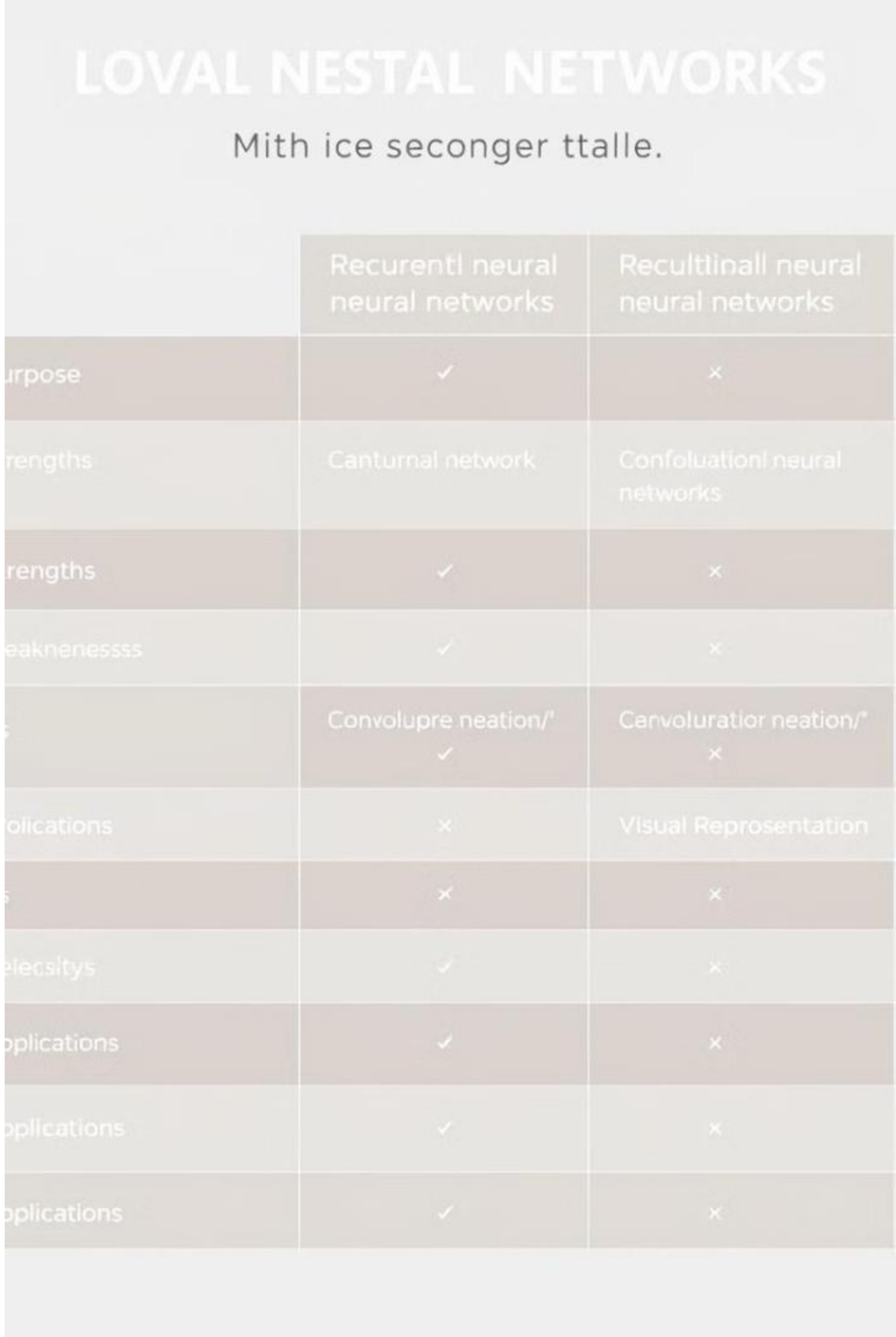
   The final derivative of the loss function with respect to the weight matrix W is computed:

   $$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \cdot \sum_{k=1}^{3} \frac{\partial h_3}{\partial h_k} \cdot \frac{\partial h_k}{\partial W}$$

This iterative process is the essence of backpropagation through time.

RNN vs. Feed Forward NN & CNN

Feature	RNN	CNN
Data Type	Sequential	Spatial
Memory Retention	Yes (short-term)	No
Suitable for	Time-series, NLP	Images, Grid-based data



LOVAL NESTAL NETWORKS

Mith ice seconger ttalle.

	Recurentl neural neural networks	Recultlinall neural neural networks
urpose	✓	×
rengths	Canturnal network	Confoluationl neural networks
rengths	✓	×
eaknenessss	✓	×
s	Convolupre neation/' ✓	Cervoluratior neation/' ×
olications	×	Visual Reprosentation
s	×	×
elecsitys	✓	×
pplications	✓	×
pplications	✓	×
pplications	✓	×

Advantages of RNNs

1 **Sequential Dependencies**

RNNs capture sequential dependencies, making them suitable for tasks where the order of data matters.

2 **Variable-Length Inputs**

RNNs can process variable-length inputs, allowing them to handle sequences of different lengths.

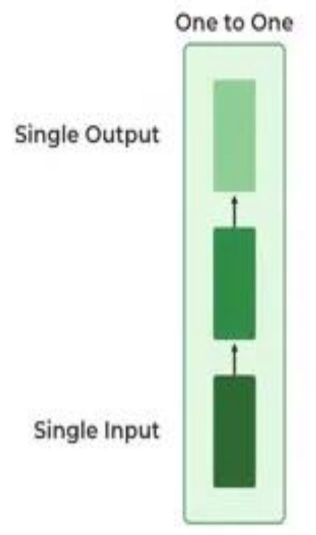
3 **Text and Speech**

RNNs are effective for text and speech applications, such as language modeling and speech recognition.

Types of RNNs: Input/Output

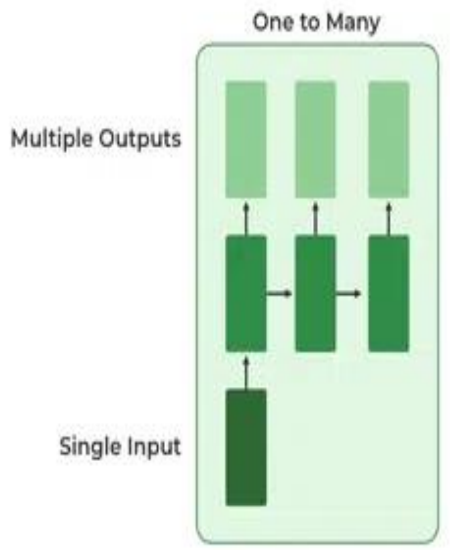
One-to-One

Single input, single output. Used for straightforward classification tasks where no sequential data is involved.



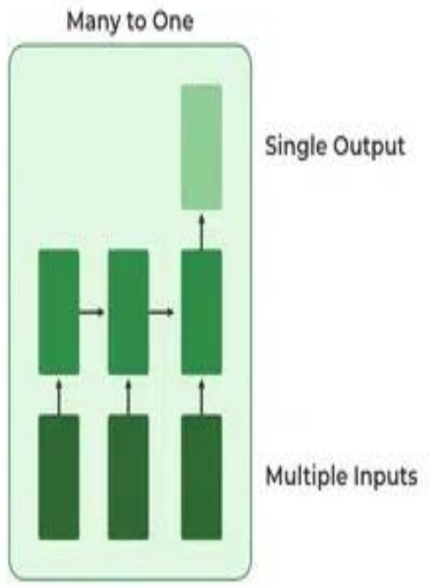
One-to-Many

Single input, multiple outputs over time. Useful in tasks where one input triggers a sequence of predictions.



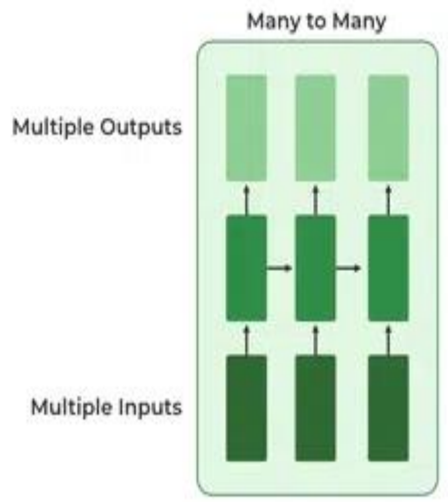
Many-to-One

Sequence of inputs, single output. Useful when the overall context of the input sequence is needed to make one prediction.

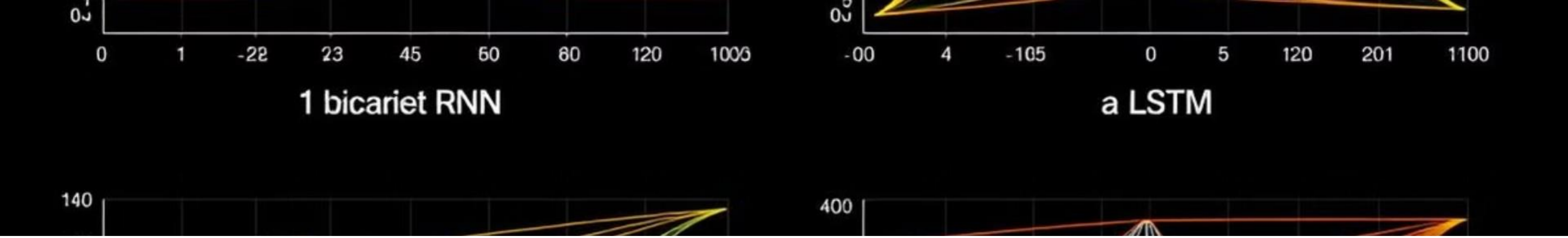


Many-to-Many

Sequence of inputs, sequence of outputs. Used in tasks like language translation, where a sequence in one language is translated to another.



Made with Gamma



Variants of RNNs

1. **Vanilla RNN**

   Simplest form, suitable for short-term dependencies but limited by the vanishing gradient problem.

2. **Bidirectional RNNs**

   Process inputs in both directions, capturing both past and future context for each time step.

3. **LSTMs**

   Introduce a memory mechanism to overcome the vanishing gradient problem, handling long-term dependencies.

4. **GRUs**

   Simplify LSTMs by combining gates, computationally efficient and useful in tasks where faster training is beneficial.

Key Takeaways

Sequential Data

RNNs are designed to process sequential data by maintaining an internal memory state.

Temporal Dependencies

Backpropagation Through Time (BPTT) is used to update network parameters based on temporal dependencies.

Variants

Variants like LSTMs and GRUs address the vanishing gradient problem and improve long-term dependency handling.

Thank You

21CSE356T

Natural Language Processing

Dr. R. USHARANI
Assistant Professor
Dept. of Computational Intelligence
SRM Institute of Science & Technology
Kattankulathur 603 203.
Chennai.

LSTM and GRU: Mastering Sequential Data

Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks are advanced types of recurrent neural networks (RNNs) designed to handle complex sequential data. These architectures address the vanishing gradient problem, enabling them to learn long-term dependencies more effectively than traditional RNNs.

This presentation will explore the components, equations, key differences, and implementation tips for both LSTM and GRU networks, providing a comprehensive understanding of these powerful tools in deep learning.

Dr R Usharani
Assistant Professor
Dept. of CINTEL
SRMIST-KTR

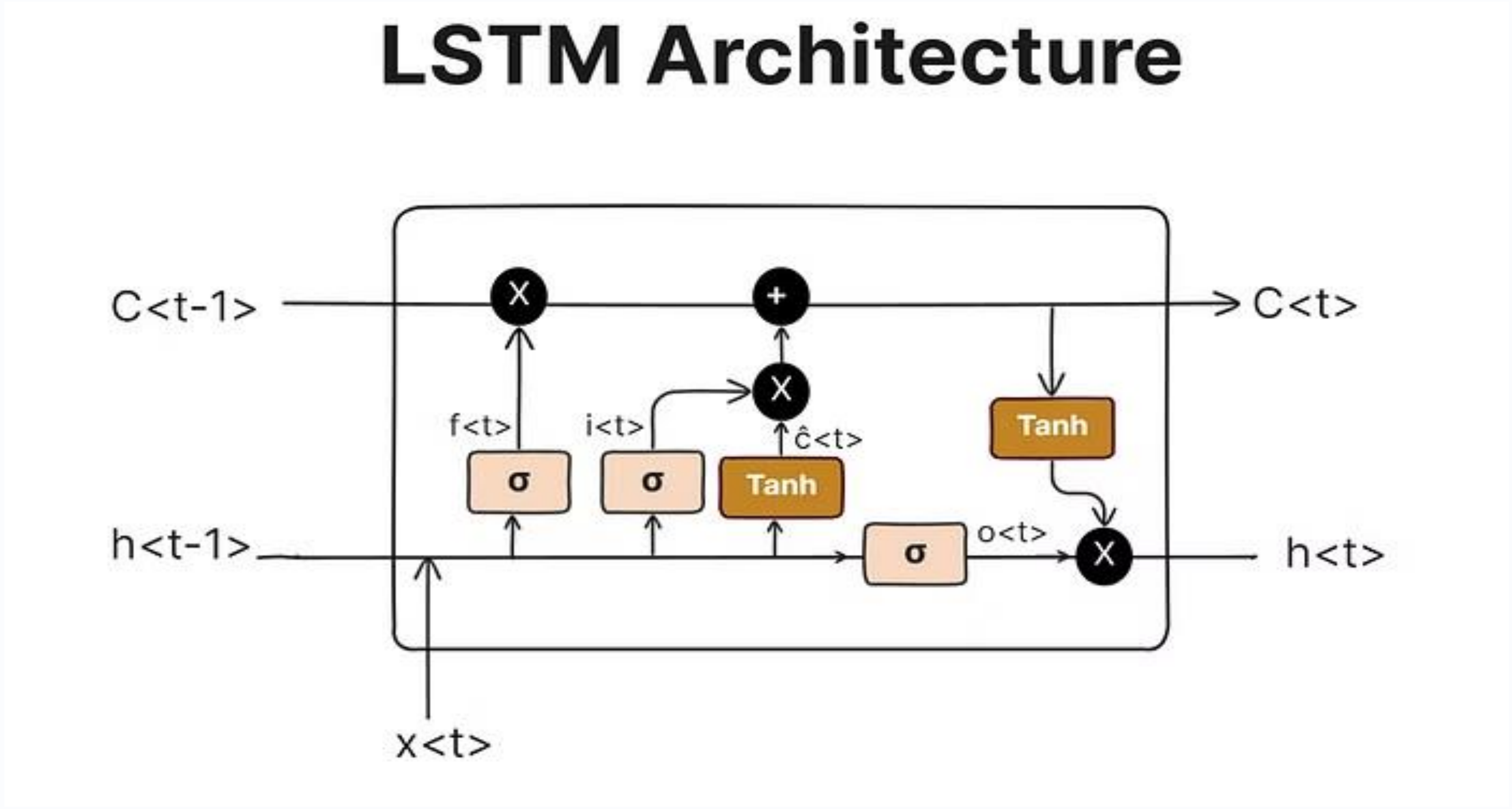
Long Short-Term Memory (LSTM)

Addressing the Vanishing Gradient Problem

LSTM is a special kind of RNN capable of learning long-term dependencies. The key to LSTMs is the cell state, which can carry information across many time steps, and three types of gates that regulate the flow of information into and out of the cell.

Components of LSTM

- Cell State (C): The memory of the network.
- Forget Gate (f): Decides what information to discard.
- Input Gate (i): Decides what new information to add.
- Output Gate (o): Decides what information to output.
- Hidden State (h): The output of the LSTM unit.



σ: denotes the sigmoid function
∗: denotes element-wise multiplication
b: bias terms
W: weight matrices
X<t>: is the input at time *t*

Made with Gamma

$$\text{Candidate Cell State: } \hat{C}^{\langle t \rangle} = tanh(W_c[h^{\langle t-1 \rangle}, X^{\langle t \rangle}] + b_c)$$

$$\text{Input Gate: } i^{\langle t \rangle} = \sigma(W_i[h^{\langle t-1 \rangle}, X^{\langle t \rangle}] + b_i)$$

$$\text{Forget Gate: } f^{\langle t \rangle} = \sigma(W_f[h^{\langle t-1 \rangle}, X^{\langle t \rangle}] + b_f)$$

$$\text{Output Gate: } o^{\langle t \rangle} = \sigma(W_o[h^{\langle t-1 \rangle}, X^{\langle t \rangle}] + b_o)$$

$$\text{Cell State: } C^{\langle t \rangle} = i * \hat{C}^{\langle t \rangle} + f * C^{\langle t-1 \rangle}$$

$$\text{Hidden State: } h^{\langle t \rangle} = o * tanh(C^{\langle t \rangle})$$

LSTM Equations

LSTM networks use a set of equations to manage the flow of information through the cell. These equations define how the forget gate, input gate, output gate, and cell state are updated at each time step.

The sigmoid function (σ) is used to produce values between 0 and 1, which determine the extent to which information is passed through the gates. Weight matrices (W) and bias terms (b) are learned during training to optimize the network's performance. The input at time t is denoted as X.

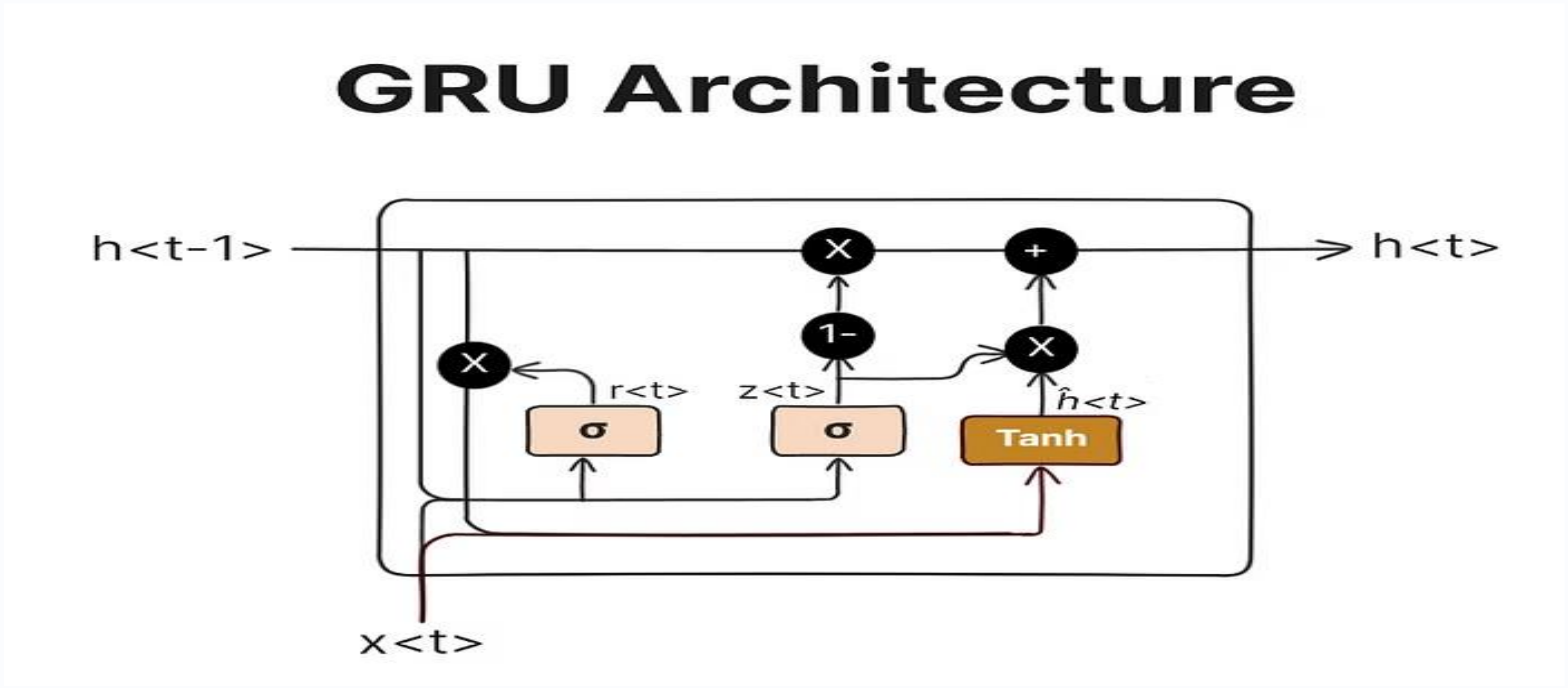
Gated Recurrent Unit (GRU)

Simpler Structure, Similar Performance

GRU is a variant of LSTM that aims to achieve similar performance with a simpler structure. GRUs combine the forget and input gates into a single update gate and merge the cell state and hidden state.

Components of GRU

- Reset Gate (r): Determines how much past information to forget.
- Update Gate (z): Controls the balance between the previous hidden state and the new candidate's hidden state.
- Hidden State (h): The output of the GRU unit.



GRU Equations

Similar to LSTMs, GRUs use a set of equations to update the hidden state and manage the flow of information. The reset gate determines how much of the past hidden state to forget, while the update gate controls how much of the new candidate hidden state to incorporate.

The sigmoid function (σ) is used to produce values between 0 and 1 for the gates. Weight matrices (W) and bias terms (b) are learned during training. The input at time t is denoted as X.

$$\text{Reset Gate: } r^{<t>} = \sigma(W_r X^{<t>} + W_r h^{<t-1>} + b_r)$$
$$\text{Update Gate: } z^{<t>} = \sigma(W_z X^{<t>} + W_z h^{<t-1>} + b_z)$$
$$\text{Condidate Hidden State: } \hat{h}^{<t>} = tanh(W_h X^{<t>} + W_h(r^{<t>} * h^{<t-1>}) + b_h)$$
$$\text{Hidden State: } h^{<t>} = \hat{h}^{<t>} * z^{<t>} + (1 - z^{<t>}) * h^{<t-1>}$$

σ: denotes the sigmoid function
∗: denotes element-wise multiplication
b: bias terms
W: weight matrices
X<t>: is the input at time *t*

Key Differences Between LSTM and GRU

Gates

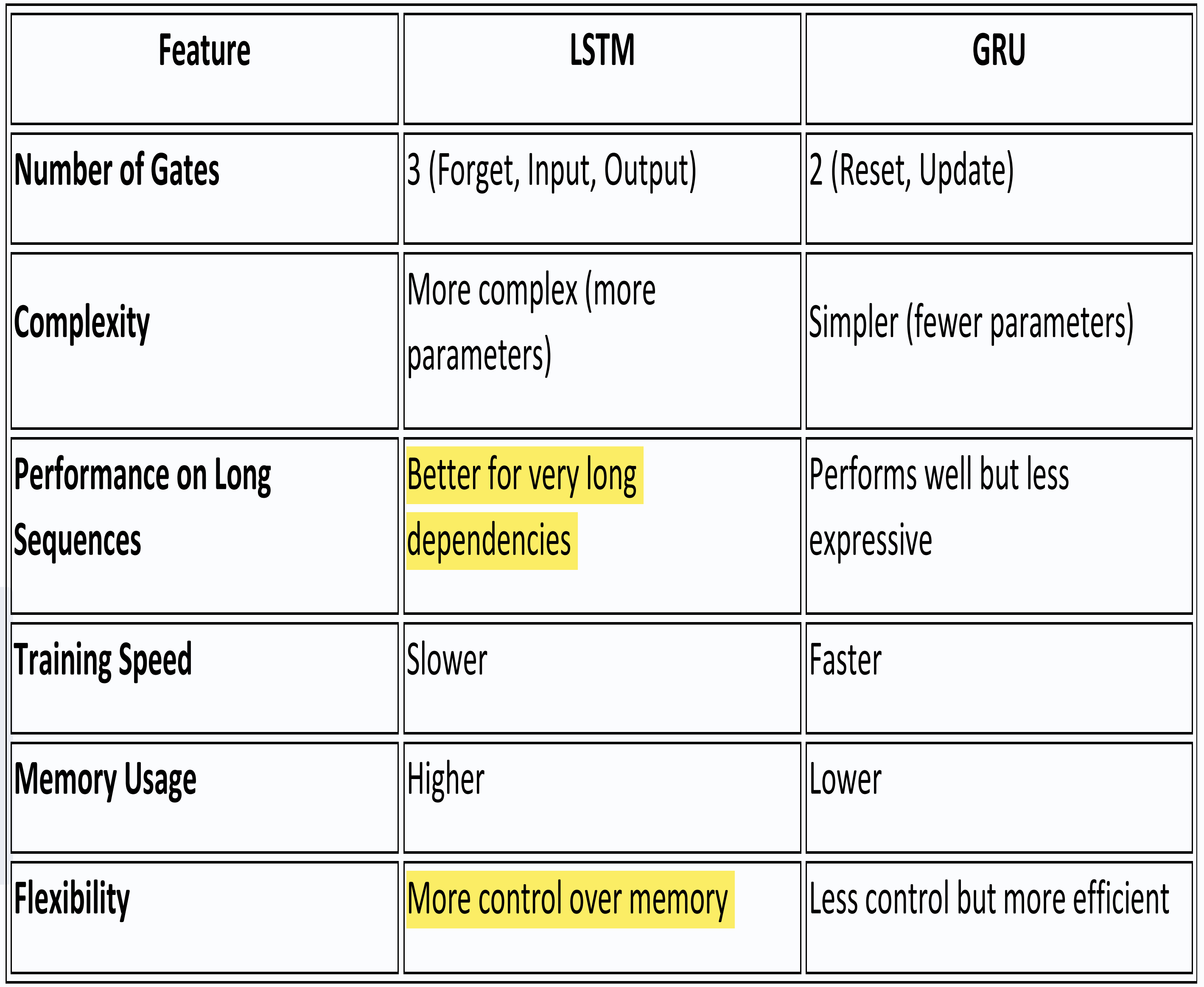
LSTM has three gates (forget, input, output), while GRU has two gates (update and reset).

Complexity

LSTM is more complex due to having more gates and separate cell states. GRU is simpler and thus faster to train and easier to implement.

Performance

Both can perform similarly on various tasks, but specific tasks might favor one architecture over the other.



Feature	LSTM	GRU
Number of Gates	3 (Forget, Input, Output)	2 (Reset, Update)
Complexity	More complex (more parameters)	Simpler (fewer parameters)
Performance on Long Sequences	Better for very long dependencies	Performs well but less expressive
Training Speed	Slower	Faster
Memory Usage	Higher	Lower
Flexibility	More control over memory	Less control but more efficient

Implementation Tips

1

Choosing Between LSTM and GRU

If you need a simpler and faster model, start with GRU. For tasks requiring longer-term memory and more control over memory, use LSTM.

2

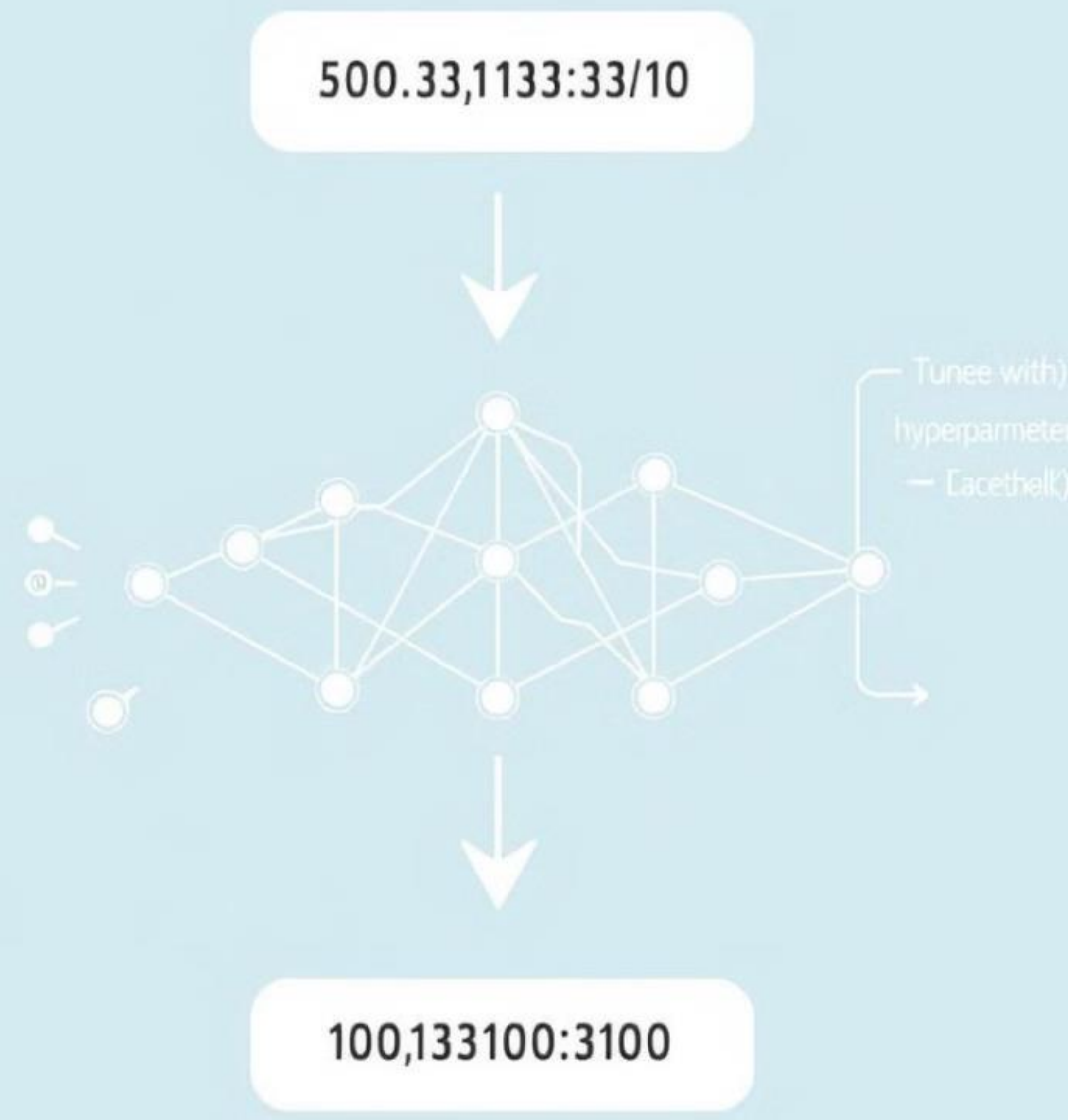
Hyperparameters

Tune the number of layers and units per layer, learning rate, batch size, and dropout rates.

3

Regularization

Use techniques like dropout to prevent overfitting.



Application Areas of LSTM & GRU in NLP

Both LSTM and GRU are widely used in NLP tasks, such as:

- **Speech Recognition** – Google Assistant, Alexa, Siri
- **Chatbots & Conversational AI** – OpenAI's ChatGPT, Facebook's BlenderBot
- **Machine Translation** – Google Translate
- **Text Summarization** – AI-generated news summaries
- **Autocorrect & Predictive Text** – Smartphone keyboards
- **Question Answering Systems** – AI-based search engines
- **Emotion & Sentiment Analysis** – Social media sentiment tracking

Conclusion

- **RNNs** are powerful but struggle with long-term dependencies.
- **LSTM** introduces gates to improve memory and solve the vanishing gradient problem.
- **GRU** simplifies LSTM while maintaining performance.
- Both **LSTM and GRU are widely used in NLP** applications, with GRU being more efficient and LSTM being more expressive for complex tasks.

Conclusion

- **RNNs** are powerful but struggle with long-term dependencies.
- **LSTM** introduces gates to improve memory and solve the vanishing gradient problem.
- **GRU** simplifies LSTM while maintaining performance.
- Both **LSTM and GRU are widely used in NLP** applications, with GRU being more efficient and LSTM being more expressive for complex tasks.

LSTM and GRU networks are powerful tools for handling sequential data, each with its own strengths and trade-offs. LSTMs offer more control over memory and are suitable for tasks requiring long-term dependencies, while GRUs provide a simpler and faster alternative.

By understanding the components, equations, key differences, and implementation tips for both architectures, you can effectively leverage these networks to solve a wide range of sequence modeling problems. Frameworks like TensorFlow and PyTorch provide built-in support for LSTM and GRU layers, making it easier to implement and experiment with these models.

Transformer Attention Mechanism in NLP

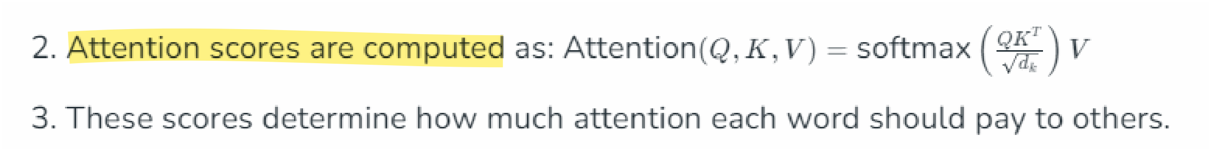
Transformer model is a type of neural network architecture designed to handle sequential data primarily for tasks such as language translation, text generation and many more. Unlike traditional recurrent neural networks (RNNs) or convolutional neural networks (CNNs), **Transformers uses attention mechanism to capture relationships between all words in a sentence regardless of their distance from each other.**

The attention mechanism is a technique that **allows models to focus on specific parts of the input sequence when producing each element of the output sequence**. It assigns different weights to different input elements enabling the model to prioritize certain information over others. This is particularly useful in tasks like language translation where the meaning of a word often depends on its context. In this article we will learn about different types of Transformer's attention mechanism.

The **attention mechanism** allows transformers to determine **which words in a sentence are most relevant** to each other. This is done using a **scaled dot-product attention** approach:

1. Each word in a sequence is **mapped to three vectors**:

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



2. Attention scores are computed as: $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$

3. These scores determine how much attention each word should pay to others.

Multi-Head Attention

Instead of using a single attention mechanism transformers apply **multi-head attention** where multiple attention layers run in parallel. This enables the model to **capture different types of relationships within the input**.

4. Encoder-Decoder Architecture

The **encoder-decoder** structure is key to transformer models. **The encoder processes the input sequence into a vector**, while the **decoder converts this vector back into a sequence**. Each encoder and decoder layer includes **self-attention** and **feed-forward layers**. In the decoder, an encoder-decoder attention layer is added to focus on relevant parts of the input.

For example, a French sentence ***"Je suis étudiant"*** *is translated into* ***"I am a student"*** *in English.*

The **encoder** consists of multiple layers (typically **6 layers**). Each layer has **two main components**:

- **Self-Attention Mechanism** – Helps the model understand word relationships.
- **Feed-Forward Neural Network** – Further transforms the representation.

The **decoder** also consists of **6 layers**, but with an additional **encoder-decoder attention** mechanism. This allows the decoder to focus on **relevant parts of the input sentence** while generating output.

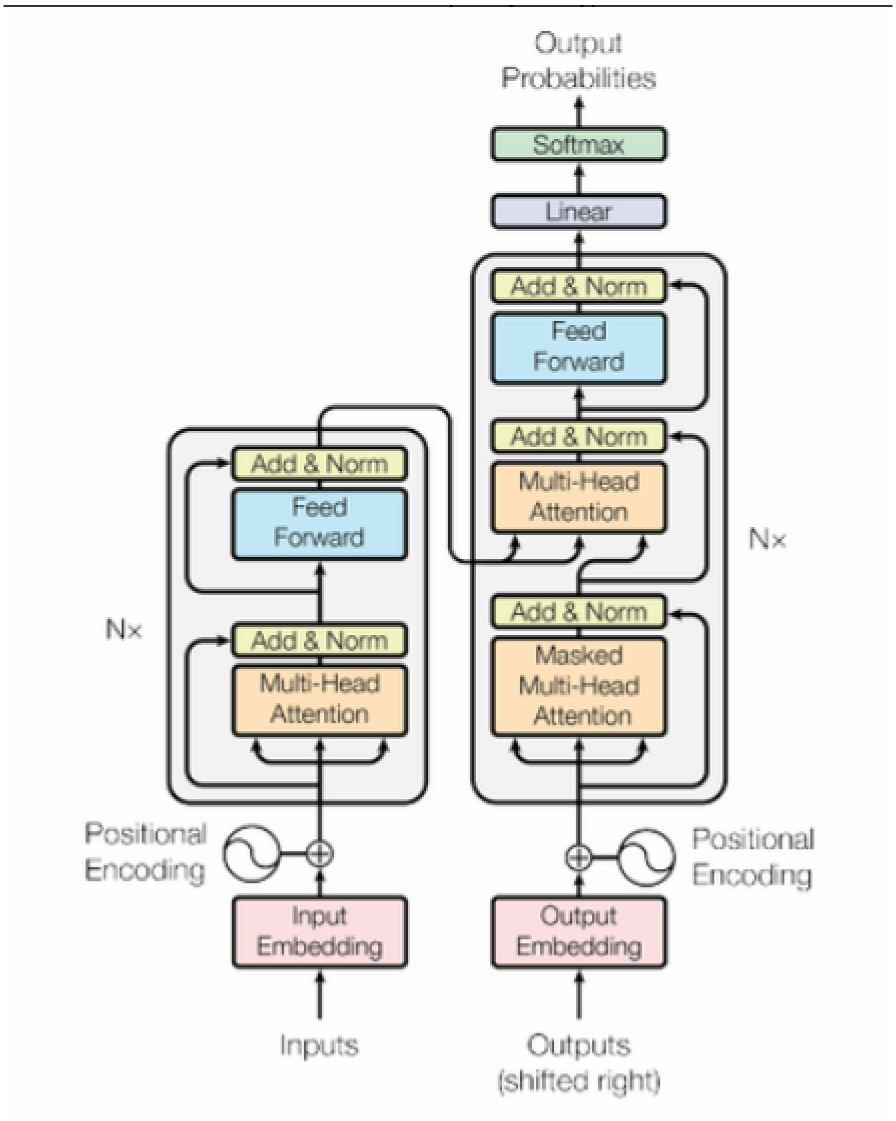


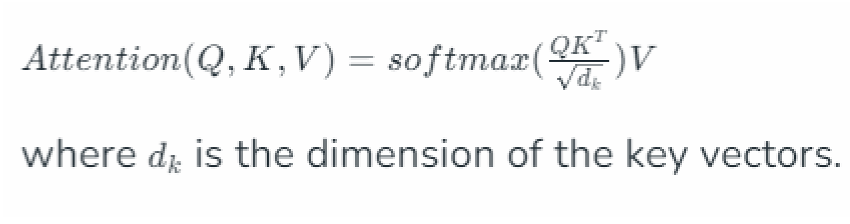
Figure 1: The Transformer - model architecture.

For instance in the sentence ***"The cat didn't chase the mouse, because it was not hungry",*** the word 'it' refers to 'cat'. The self-attention mechanism helps the model correctly associate 'it' with 'cat' ensuring an accurate understanding of sentence structure.

Types of Attention Mechanisms in Transformers

1. Scaled Dot-Product Attention

The Scaled Dot-Product Attention is the fundamental building block of the Transformer's attention mechanism. It involves three main components: queries (Q), keys (K), and values (V). The attention score is computed as the dot product of the query and key vectors, scaled by the square root of the dimension of the key vectors. This score is then passed through a softmax function to obtain the attention weights, which are used to compute a weighted sum of the value vectors.



$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

where d_k is the dimension of the key vectors.

Scaled Dot-Product Attention is key to the Transformer.

Uses three vectors: Query (Q), Key (K), Value V).

Compute attention score

Apply softmax to get attention weights.

Use weights to compute a weighted sum of values (V).

2. Multi-Head Attention

Multi-Head Attention enhances the model's ability to focus on different parts of the input sequence simultaneously. It involves multiple attention heads, each with its own set of query, key, and value matrices. The outputs of these heads are concatenated and linearly transformed to produce the final output. This allows the model to capture different features and dependencies in the input sequence.

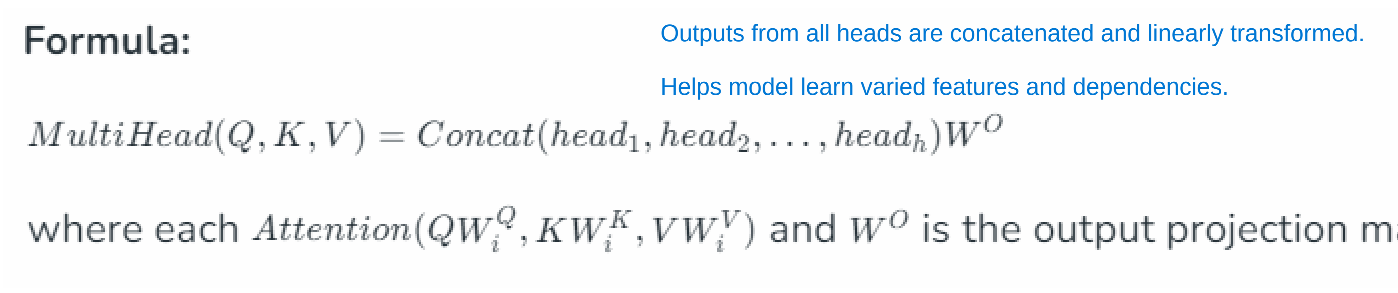
Multi-Head Attention = multiple attention layers (heads) in parallel. Each head has its own Q, K, V matrices.

Heads focus on different parts of the input.

Outputs from all heads are concatenated and linearly transformed.

Helps model learn varied features and dependencies.

Formula:



Formula:

$$MultiHead(Q, K, V) = Concat(head_1, head_2, \ldots, head_h)W^O$$

where each $Attention(QW_i^Q, KW_i^K, VW_i^V)$ and W^O is the output projection m

3. Self-Attention

Self-Attention is also known as intra-attention, allows the model to consider different positions of the same sequence when computing the representation of a word. In the context of the Transformer, self-attention is applied in both the encoder and decoder layers. It enables the model to capture long-range dependencies and relationships within the input sequence.

4. Encoder-Decoder Attention

Encoder-Decoder Attention also known as cross-attention, is used in the decoder layers of the Transformer. It allows the decoder to focus on relevant parts of the input sequence (encoded by the encoder) when generating each word of the output sequence. This type of attention ensures that the decoder has access to the entire input sequence, helping it produce more accurate and contextually appropriate translations.

5. Causal or Masked Self-Attention

Causal or Masked Self-Attention is used in the decoder to ensure that the prediction for a given position only depends on the known outputs at positions before it. This is crucial for tasks like language modeling where future tokens should not be visible during training. The attention scores for future tokens are masked out, ensuring that the model cannot look ahead.

Formula:

$$MaskedAttention(Q, K, V) = softmax(\frac{QK^T + M}{\sqrt{d_k}})V$$

where M is the mask matrix with $-\infty$ in positions that should be masked.

Significance of Transformer Attention Mechanism

The attention mechanism in Transformers offers several advantages:

1. **Parallel Processing:** Unlike RNNs, Transformers can process all words in a sequence simultaneously, significantly reducing training time.
2. **Long-Range Dependencies:** The attention mechanism can capture relationships between distant words, addressing the limitations of traditional models that struggle with long-range dependencies.
3. **Scalability:** Transformers can handle larger datasets and complex tasks due to their scalable architecture.

Applications of Transformers

Some of the applications of transformers are:

1. **NLP Tasks**: Transformers are used for machine translation, text summarization, named entity recognition and sentiment analysis.
2. **Speech Recognition**: They process audio signals to convert speech into transcribed text.
3. **Computer Vision**: Transformers are applied to image classification, object detection, and image generation.
4. **Recommendation Systems**: They provide personalized recommendations based on user preferences.
5. **Text and Music Generation**: Transformers are used for generating text (e.g., articles) and composing music.

Transformers have redefined deep learning across NLP, computer vision, and beyond. With advancements like BERT, GPT and Vision Transformers (ViTs) they continue to push the boundaries of AI and language understanding and multimodal learning.

BERT

Introduction to BERT

- What is BERT?
- Developed by Google in 2018
- A breakthrough in NLP using Transformers
- Enables contextual understanding of words

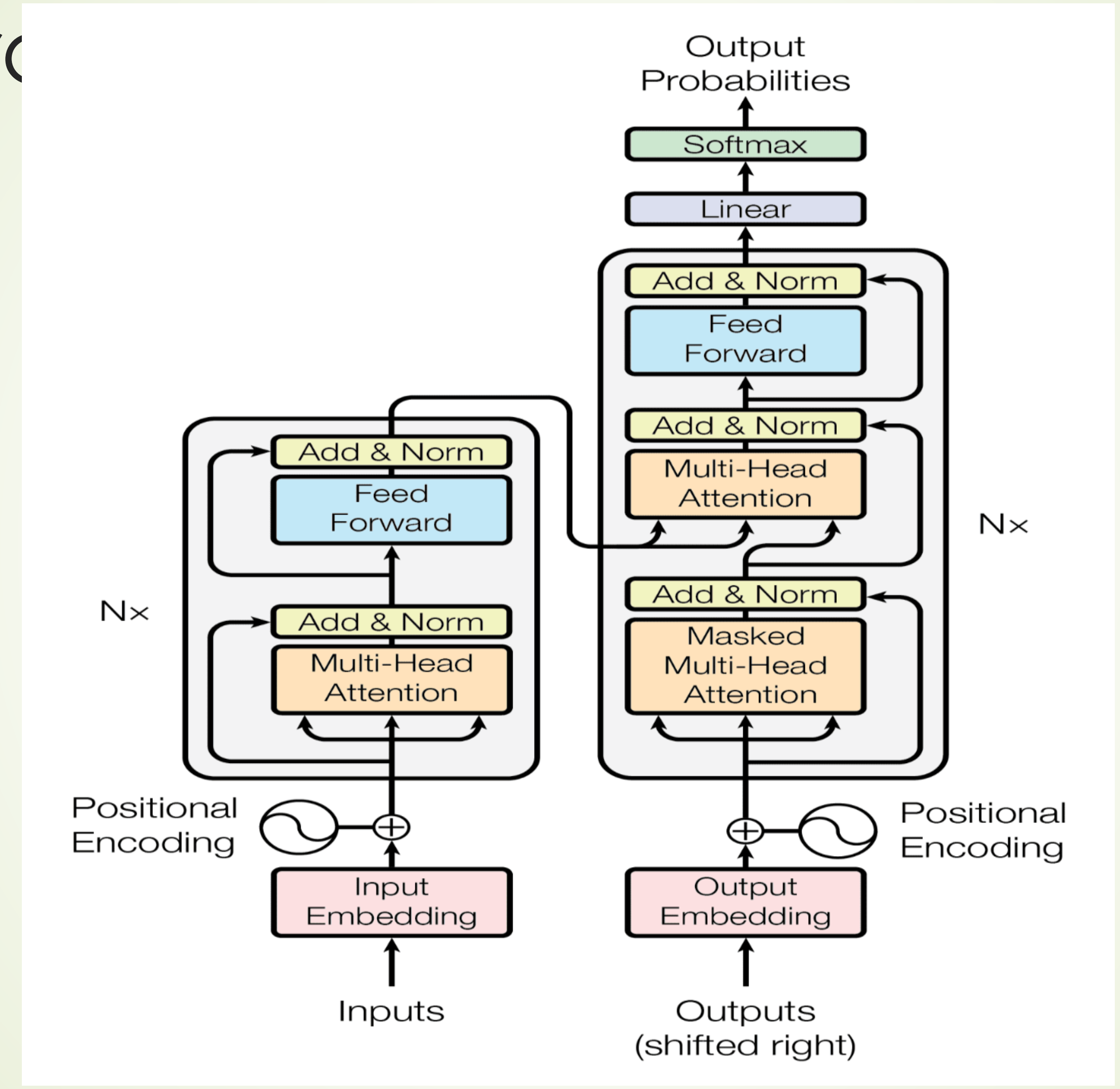
Key Features of BERT

- **Bidirectional Learning:** Context is captured from both left and right
- **Pre-trained on Large Datasets:** Wikipedia and BooksCorpus
- **Fine-Tunable:** Can be adapted for various NLP tasks
- **Uses Transformers:** Attention-based architecture

Architecture of BERT

- Based on Transformer Encoder
- Input Representation: Token embeddings, Segment embeddings, Position embeddings
- Multi-head Self-Attention Mechanism
- Output Representations for different tasks

Arc


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Training BERT

- **Two Training Objectives:**
  - Masked Language Modeling (MLM)
  - Next Sentence Prediction (NSP)
- Trained on massive text corpora
- Requires large computational power

Fine Tuning BERT for NLP Tasks

- Can be adapted for various applications
- Common tasks:
  - Text Classification
  - Named Entity Recognition (NER)
  - Question Answering (QA)
  - Sentiment Analysis
- Process: Add task-specific layers and train on labeled data

Applications of BERT

- Search Engines (Google Search)
- Chatbots and Virtual Assistants
- Machine Translation
- Medical and Legal Document Analysis
- Sentiment Analysis in Social Media

Limitations of BERT

- Computationally Expensive
- Requires Large Training Data
- Cannot Handle Very Long Sequences Well
- Not Always Interpretable

RoBERTa

Introduction

- Developed by Facebook AI in 2019
- Improves upon BERT with better training strategies
- More robust and optimized for NLP tasks

Key Differences Between BERT and RoBERTa

- Removes Next Sentence Prediction (NSP)
- Trains with More Data and Larger Batch Sizes
- Uses Dynamic Masking Instead of Static Masking
- Longer Training Time with More Computation
- Outperforms BERT in Several NLP Benchmarks

Architecture of RoBERTa

- Based on BERT’s Transformer Encoder
- Input Representations: Token, Position, and Segment Embeddings
- Uses Multi-head Self-Attention
- Outputs Contextualized Representations for NLP Tasks

- **No Next Sentence Prediction (NSP)**
- **Uses More Data:** BooksCorpus + OpenWebText + CC-News + Stories
- **Longer Training with Bigger Batches**
- **Dynamic Masking for Better Generalization**

- Adapted for various NLP applications
- Common tasks:
  - Sentiment Analysis
  - Text Classification
  - Named Entity Recognition (NER)
  - Question Answering (QA)
  - Machine Translation

Applications of RoBERTa

- Search Engines and Web Ranking
- Chatbots and Virtual Assistants
- Fake News and Spam Detection
- Financial Document Analysis
- Healthcare and Biomedical NLP

Advantages of RoBERTa

- Outperforms BERT on multiple NLP benchmarks
- More efficient use of training data
- Better generalization due to dynamic masking
- Eliminates unnecessary NSP task

Limitations of RoBERTa

- Computationally Expensive
- Requires Large Amounts of Data
- Difficult to Interpret Compared to Traditional Models
- Cannot Handle Very Long Sequences Efficiently

Text Classification and Text Generation

Text Classification

- **Definition:** Assigning predefined labels to text data
- **Examples:** Spam detection, sentiment analysis, topic categorization
- **Role in NLP:** Helps in organizing, filtering, and understanding text data

Approaches to Text Classification

- **Traditional Methods:**
  - Naïve Bayes
  - Support Vector Machines (SVM)
  - Decision Trees
- **Deep Learning-Based Methods:**
  - LSTMs & GRUs
  - CNNs for text classification
  - Transformer models (BERT, RoBERTa, etc.)

Steps in Text Classification

1. **Data Preprocessing:** Tokenization, Stopword Removal, Stemming, Lemmatization
2. **Feature Extraction:** TF-IDF, Word Embeddings (Word2Vec, GloVe, FastText)
3. **Model Selection:** Choose the appropriate model (ML, DL, Transformers)
4. **Training & Evaluation:** Train on labeled data and validate performance
5. **Deployment & Optimization:** Fine-tune the model for real-world applications

Application of Text classification

- Sentiment Analysis (e.g., Movie and Product Reviews)
- Fake News Detection
- Spam Email Filtering
- Customer Support Automation
- Document Categorization

Text Generation

- **Definition:** Automatically generating coherent text from input prompts
- **Examples:** Chatbots, Article Generation, Code Completion
- **Importance in NLP:** Enables human-like text creation

Approaches to Text Generation

- **Rule-Based Methods:** Predefined templates and grammatical rules
- **Statistical Methods:** N-grams, Hidden Markov Models (HMM)
- **Deep Learning-Based Methods:**
    - Recurrent Neural Networks (RNNs)
    - Long Short-Term Memory (LSTMs)
    - Transformers (GPT, T5, BART)

Steps in Text Generation

1. **Data Collection:** Large-scale text datasets
2. **Model Training:** Training on large text corpora
3. **Fine-Tuning:** Adapting for specific applications
4. **Inference:** Generating text based on input prompts
5. **Evaluation:** Assessing coherence, relevance, and fluency

Applications

- Automated Content Creation (Articles, Blogs, Reports)
- Chatbots and Conversational AI (Customer Support, Virtual Assistants)
- Text Summarization
- Code Generation (e.g., GitHub Copilot)
- Creative Writing Assistance

Drawbacks in Text classification and Text Generation

- Handling Bias in Training Data
- Need for Large Computational Resources
- Ensuring Coherence and Avoiding Hallucination in Generation
- Security Concerns (e.g., Fake News, Misinformation)