Natural Language Processing
Course code: CSE3015

Module 4
NLP Using Deep Learning

Prepared by

Dr. Venkata Rami Reddy Ch

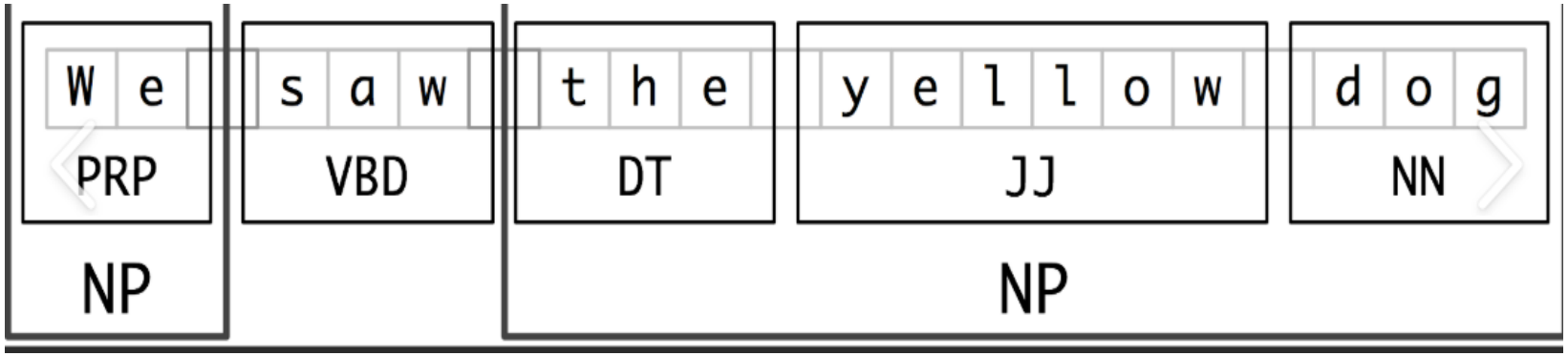
SCOPE

Syllabus

- Types of learning techniques,
- Chunking,
- Information extraction & Relation Extraction,
- Recurrent neural networks,
- LSTMs/GRUs,
- Transformers,
- Self-attention Mechanism
- Sub-word tokenization
- Positional encoding

Chunking in NLP

- Chunking in NLP refers to the process of breaking down a text into meaningful phrases or segments called "chunks.“
- Chunking, also known as **shallow parsing**, is a technique in **NLP** used to extract meaningful phrases (chunks) from a sentence.
- It groups words into **phrases(chunks)** based on their **Part-of-Speech (POS) tags**.
- These chunks are usually bigger than individual words but smaller than full sentences.
- Chunking helps in better understanding the structure and meaning of a sentence.
- chunking focuses on **smaller, useful phrases**, such as **noun phrases (NPs)** and **verb phrases (VPs)**.



Chunking in NLP

How Chunking Works

1. **POS Tagging**: The sentence is first tokenized and assigned POS tags.
2. **Pattern Rules**: Regular expressions or machine learning models are used to define patterns for grouping words.
3. **Chunking Process**: Using these patterns, groups of words are extracted.

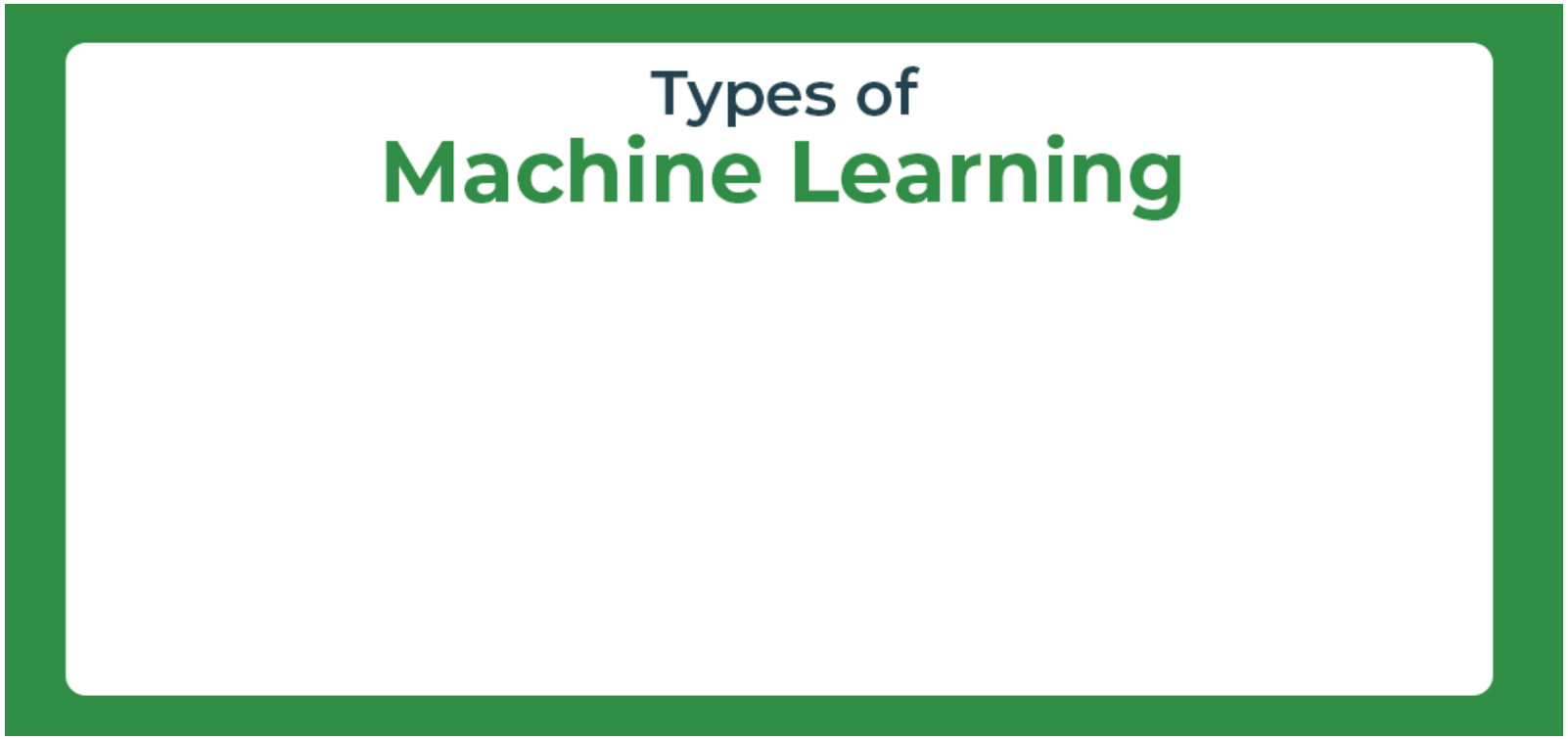
Chunking in NLP

```
import nltk
sentence = "The quick brown fox jumps over the lazy dog"
tokens = nltk.word_tokenize(sentence)
pos_tags = nltk.pos_tag(tokens)

# Define the chunk grammar for NP (Noun Phrase), VP (Verb Phrase), and PP (Prepositional Phrase)
chunk_grammar = r"""
    NP: {<DT>?<JJ>*<NN.*>}      # Determiner (optional) + Adjective (0 or more) + Noun
    VP: {<VB.*><NP|PP>*}        # Verb + (Optional NP or PP)
    PP: {<IN><NP>}              # Preposition + NP
"""
# Create chunk parser
chunk_parser = nltk.RegexpParser(chunk_grammar)
chunks = chunk_parser.parse(pos_tags)
print(chunks)
```

```
(S
  (NP The/DT quick/JJ brown/NN)
  (NP fox/NN)
  (VP jumps/VBZ)
  (PP over/IN (NP the/DT lazy/JJ dog/NN)))
```

Types of learning techniques

- Machine learning (ML) is a subset of artificial intelligence (AI) that enables computers to learn patterns from data and make decisions or predictions based on learned patterns.



Types of learning techniques

supervised Learning

- In this approach, the model is trained on labeled data.
- Supervised learning is a type of machine learning where a model learns from labeled data (i.e., input-output pairs).
- The goal is to find a mapping function(weights) from inputs to outputs so that the model can make accurate predictions on unseen data.
- labeled data refers to a dataset that includes input data paired with the correct output, or labels.
- Example models include linear regression, logistic regression, support vector machines, and neural networks

Examples of Labeled Data:

1. **Text Classification:**
   - Input: "This movie was amazing!"
   - Label: **Positive sentiment**
2. **Image Recognition:**
   - Input: A picture of a dog
   - Label: **"Dog"**
3. **Spam Detection:**
   - Input: Email content
   - Label: **"Spam" or "Not Spam"**
4. **Medical Diagnosis:**
   - Input: X-ray image
   - Label: **"Pneumonia" or "No Pneumonia"**

Types of learning techniques

Unsupervised Learning

- Here, the model is trained on unlabeled data and find patterns and relationships within the data.

•The model learns patterns from unlabelled data without explicit outputs.

Common techniques **:**

- **Clustering** (e.g., K-Means, DBSCAN, Hierarchical Clustering)
- **Dimensionality Reduction** (e.g., PCA, t-SNE, Autoencoders)

Semi-Supervised Learning

- This is a mix of supervised and unsupervised learning.
- The model is trained on a small amount of labeled data and a large amount of unlabeled data.
- This is useful when labeling data is expensive or time-consuming

Examples:

- Medical diagnosis with a few labeled cases and many unlabeled images

Types of learning techniques

Reinforcement Learning:

- In this type, an agent learns by interacting with its environment and receiving feedback in the form of rewards or penalties.
- It's commonly used in robotics, gaming, and autonomous systems.

Examples:

- **Self-driving** cars optimizing driving strategies

Self-Supervised Learning:

- A subset of supervised learning where the model generates its own labels.
- For example, predicting the next word in a sentence.

Information extraction

- **Information Extraction (IE) is** task of finding structured information from unstructured or semi structured text.
- The input to IE system is a collection of documents (email, web pages, news groups, news articles, business reports, research papers, blogs, resumes, proposals, and soon) and output is a representation of the relevant information.
- This process typically involves identifying and extracting specific types of information such as

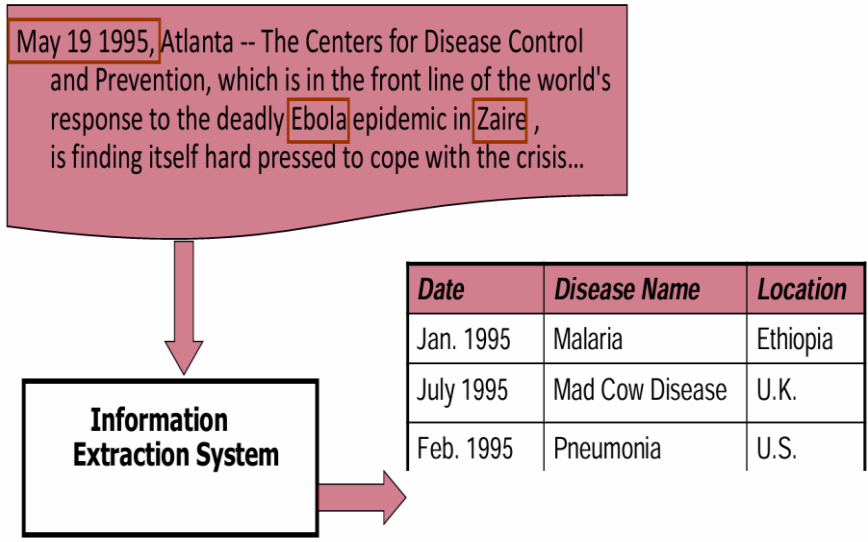
Entities(NER)

- People, organizations, locations, times, dates, prices, ...
- Or sometimes: genes, proteins, diseases, medicines, ...

Relations between entities(Relation extraction)

- Located in, employed by, part of, married to, ...

larger events that are taking place(Event extraction)



Sub tasks in IE

1. Named Entity Recognition
2. Relation extraction
3. Event extraction
4. Coreference Resolution

Named Entity Recognition (NER)

- The first step in information extraction is to detect the entities in the text.
- A named entity is, anything that can be referred to with a proper name: a person, a location, an organization.
- The term is commonly extended to include things that aren't entities, including dates, times, and prices.
- Named Entity Recognition (NER) is used to identify and classify named entities in text into predefined categories such as Person, Organization, Location, Date, Time, Money, Percentages, Products, etc.

Citing high fuel prices, [ORG **United Airlines**] said [TIME **Friday**] it has increased fares by [MONEY **$6**] per round trip on flights to some cities also served by lower-cost carriers. [ORG **American Airlines**], a unit of [ORG **AMR Corp.**], immediately matched the move, spokesman [PER **Tim Wagner**] said. [ORG **United**], a unit of [ORG **UAL Corp.**], said the increase took effect [TIME **Thursday**] and applies to most routes where it competes against discount carriers, such as [LOC **Chicago**] to [LOC **Dallas**] and [LOC **Denver**] to [LOC **San Francisco**].

- The text contains 13 mentions of named entities including 5 organizations, 4 locations, 2 times, 1 person, and 1 mention of money.

Named Entity Recognition (NER)

- A list of generic named entity types with the kinds of entities they refer to.

Type	Tag	Sample Categories	Example sentences
People	PER	people, characters	**Turing** is a giant of computer science.
Organization	ORG	companies, sports teams	The **IPCC** warned about the cyclone.
Location	LOC	regions, mountains, seas	The **Mt. Sanitas** loop is in **Sunshine Canyon**.
Geo-Political Entity	GPE	countries, states, provinces	**Palo Alto** is raising the fees for parking.
Facility	FAC	bridges, buildings, airports	Consider the **Golden Gate Bridge**.
Vehicles	VEH	planes, trains, automobiles	It was a classic **Ford Falcon**.

Type ambiguity in NER

- Type ambiguity in NER occurs when a word belongs to multiple named entity types depending on the context.
- This can lead to incorrect classification by NER models.

type ambiguities in the use of the name Washington

- [PER Washington] was born into slavery on the farm of James Burroughs.
- [ORG Washington] went up 2 games to 1 in the four-game series.
- Blair arrived in [LOC Washington] for what may well be his last state visit.

Approaches to NER

Rule-Based Approaches:

- These rely on predefined sets of rules and patterns to identify named entities.
- They are simple to implement but can be limited in their ability to generalize to new data.

Dictionary-Based Approaches:

- These use dictionaries or gazetteers of known named entities to match and identify entities in text.
- They are effective for well-defined domains but may struggle with new entities.

Machine Learning Approaches:

- These involve training models on labelled datasets to learn patterns that distinguish named entities.
- Common algorithms include Conditional Random Fields (CRFs) and Hidden Markov Models (HMMs).

Deep Learning Approaches:

- These use neural networks, such as Recurrent Neural Networks (RNNs) and Transformers, to automatically learn features from raw text.
- They have shown state-of-the-art performance in NER tasks but require large amounts of labelled data and computational resources.

Using Rule & Dictionary-Based Approaches

```
import re
import nltk

text = "Elon Musk is the CEO of Tesla Inc. He was born on 1971-06-28 in South
Africa."

# Predefined lists (gazetteers/Dictionary)
companies = {"Tesla", "Google", "Microsoft", "Amazon"}
persons = {"Elon Musk", "Bill Gates", "Jeff Bezos"}
locations = {"South Africa", "United States", "India"}

# Rule-based entity extraction
entities = {}

# Recognizing dates (YYYY-MM-DD format)
date_pattern = r'\b\d{4}-\d{2}-\d{2}\b'
entities["DATE"] = re.findall(date_pattern, text)

entities["PERSON"] = [name for name in persons if name in text]

entities["ORG"] = [name for name in companies if name in text]

entities["GPE"] = [name for name in locations if name in text]

print(entities)
```

{'DATE': ['1971-06-28'], 'PERSON': ['Elon Musk'], 'ORG': ['Tesla'], 'GPE': ['South Africa']}

Machine Learning Approaches:

Steps in Supervised NER

1. **Data Collection & Annotation**
   - Prepare a dataset with text labeled with named entities (e.g., using BIO tagging: B-PER, I-PER, O).
   - Example:

```
css                                                    Copy    Edit

John   B-PER
lives  O
in     O
New    B-LOC
York   I-LOC
```

   - Annotation tools: spaCy Prodigy, BRAT, Label Studio.
2. **Feature Engineering**
   - Extract linguistic features like:
     - Word embeddings (Word2Vec, FastText, BERT)
     - Part-of-Speech (POS) tags
     - Contextual information (neighboring words)
     - Capitalization and word shape (e.g., [illegible] → Xxxx)

Machine Learning Approaches:

3. **Model Selection**
    - Common models for NER:
        - **Conditional Random Fields (CRF)** – Uses sequence modeling.
        - **Recurrent Neural Networks (RNNs)** – LSTMs and Bi-LSTMs with CRF layers.
        - **Transformers (BERT, RoBERTa, etc.)** – State-of-the-art performance.
4. **Training the Model**
    - Split dataset into **training, validation, and test sets**.
    - Train using supervised learning techniques (e.g., **Cross-Entropy Loss** for classification).
    - Fine-tune pre-trained transformer models like **BERT-NER**.
5. **Evaluation**
    - Metrics: **Precision, Recall, F1-score**
    - Tools: **seqeval**, **scikit-learn**

Event extraction

- Event extraction is the process of identifying and classifying events mentioned in text.
- It involves extracting structured information about events, such as the participants, locations, dates, and other relevant details.

Key Components of Event Extraction

1.Event Trigger – The main verb or noun indicating an event (e.g., *"launched," "announced," "elected"*).
2.Event Type – The category of the event (e.g., *"Business," "Disaster," "Political," "Sports"*).
3.Event Arguments – The entities participating in the event (e.g., *Person, Organization, Location, Date, Time*).

Event extraction

Applications:

✓ **Financial News Monitoring** – Detecting stock-related events (e.g., mergers, acquisitions).

✓ **Disaster Alert Systems** – Extracting crisis events from social media (e.g., earthquakes, floods).

✓ **Legal Document Analysis** – Extracting case details (e.g., court cases, judgments).

Example:

text = "Elon Musk announced the launch of the new Tesla model at the event in California on March 3, 2025, at 10:00 AM."

Output:

```
Event Details:
- Event Triggers (Actions): ['announce']
- Participants: ['Elon Musk']
- Organizations: ['Tesla']
- Location: ['California']
- Date: ['March 3, 2025']
- Time: ['10:00 AM']
```

```python
import spacy

# Load spaCy model
nlp = spacy.load("en_core_web_sm")

text = "Elon Musk announced the launch of the new Tesla model at the event in California on March 3, 2025, at 10:00 AM."

# Process the text
doc = nlp(text)

# Extract named entities (including TIME)
entities = {"PERSON": [], "ORG": [], "GPE": [], "DATE": [], "TIME": []}
for ent in doc.ents:
    if ent.label_ in entities:
        entities[ent.label_].append(ent.text)

# Extract event triggers (filtering meaningful verbs)
event_triggers = [token.lemma_ for token in doc if token.pos_ == "VERB"]


# Print extracted details
print("Event Details:")
print(f"- Event Triggers (Actions): {event_triggers}")
print(f"- Participants: {entities['PERSON']}")
print(f"- Organizations: {entities['ORG']}")
print(f"- Location: {entities['GPE']}")
print(f"- Date: {entities['DATE']}")
print(f"- Time: {entities['TIME']}")
```

```
Event Details:
- Event Triggers (Actions): ['announce']
- Participants: ['Elon Musk']
- Organizations: ['Tesla']
- Location: ['California']
- Date: ['March 3, 2025']
- Time: ['10:00 AM']
```

Coreference Resolution in NLP

- Coreference resolution (CR) is the task of finding all words (called mentions) in a given text that refer to the same entity.
- After finding and grouping these words we can resolve them by replacing, pronouns with noun phrases.
- It helps in understanding context, improving text coherence, and enabling more accurate **question answering, summarization, and dialogue systems**.

Input:
"Elon Musk founded SpaceX in 2002. He is also the CEO of Tesla."

Output (Resolved Coreference):
*"Elon Musk founded SpaceX in 2002. **Elon Musk** is also the CEO of Tesla."*

Coreference Resolution in NLP

1 Pronominal Coreference

- Resolving **pronouns** (he, she, it, they, etc.) to actual entities.
- **Example:**
  - *"Obama was the U.S. President. He served for two terms."*
  - **"He" → "Obama“**

2 Named Entity Coreference

- Resolving different mentions of the **same entity** (e.g., “Tesla” and “the company”).
- **Example:**
  - *"Apple launched a new iPhone. The company expects high sales."*
  - **"The company" → "Apple"**

Relation extraction

- Relation extraction is the process of finding and extracting semantic relations among the text entities from text.
- Once entities are recognized, identify specific relations between entities
- These are often binary relations like child-of, employment, part-whole, and geospatial relations.

Relations		Examples	Types
Affiliations			
	Personal	*married to, mother of*	PER → PER
	Organizational	*spokesman for, president of*	PER → ORG
	Artifactual	*owns, invented, produces*	(PER | ORG) → ART
Geospatial			
	Proximity	*near, on outskirts*	LOC → LOC
	Directional	*southeast of*	LOC → LOC
Part-Of			
	Organizational	*a unit of, parent of*	ORG → ORG
	Political	*annexed, acquired*	GPE → GPE

Example:

– Michael Dell is the CEO of Dell Computer Corporation and lives in Austin Texas.

Rule-based Relation Extraction

- In Rule-based relation extraction, predefined linguistic rules or patterns are used to identify and classify relationships between entities in text.
- Hearst (1992a, 1998) proposed five patterns for identifying **is-a** relationships.
- These patterns help extract structured knowledge from unstructured text by recognizing the **"is-a"** relationship.

Pattern	Example
"X such as Y"	"Vehicles such as cars and bikes are common."
"X including Y"	"Programming languages including Python and Java are popular."
"X is a type of Y"	"A tulip is a type of flower."
"X, especially Y"	"Fruits, especially apples and oranges, are healthy."
"Y and other X"	"Eagles and other birds can fly.

Rule-based Relation Extraction

- Many instances of relations can be identified through hand-crafted patterns, looking for triples (X, α, Y) where X & Y are entities and α are words in between.
- For the “Paris is in France” example, α = ”is in”. This could be extracted with a regular expression.

PERSON

Founder?

Investor?

Member?

Employee?

President?

ORGANIZATION

Drug

Cure?

Prevent?

Cause?

Disease

Rule-based Relation Extraction

```
import re
def extract_custom_relations(text):

    pattern = r"(\w+) (was born in|is located in|works at|is a type of|and other) (\w+)"
    matches = re.findall(pattern, text)

    return [(match[0], match[1], match[2]) for match in matches]

text = "Musk was born in South Africa. Microsoft is located in the USA.\
         Sundar Pichai works at Google.Tulip is a type of flower.Eagles and other birds
can fly"

# Extract relations
print(extract_custom_relations(text))
```

```
[('Musk', 'was born in', 'South'), ('Microsoft', 'is located in', 'the'), ('Pichai', 'works at', 'Goo
gle'), ('Tulip', 'is a type of', 'flower'), ('Eagles', 'and other', 'birds')]
```

Extracting Richer Relations Using Rules and Named Entities

Pattern-1: PER, POSITION of ORG:
George Marshall, Secretary of State of the United States

([A-Z][a-z]+(?:\s[A-Z][a-z]+)*)\s*,\s*([A-Z][a-z]+(?:\s[A-Z][a-z]+)*)\s+of\s+([A-Z][a-z]+(?:\s[A-Z][a-z]+)*)

- ([A-Z][a-z]+(?:\s[A-Z][a-z]+)*) → Captures the **Person (PER)**
- ,\s* → Matches the comma and optional spaces
- ([A-Z][a-z]+(?:\s[A-Z][a-z]+)*) → Captures the **Position (POSITION)**
- \s+of\s+ → Matches the phrase " of "
- ([A-Z][a-z]+(?:\s[A-Z][a-z]+)*) → Captures the **Organization (ORG)**

```
import re
pattern = r"(\b[A-Z][a-z]+(?:\s[A-Z][a-z]+)*)\s*,\s*([A-Z][a-z]+(?:\s[A-Z][a-z]+)*)\s+of\s+([A-
Z][a-z]+(?:\s[A-Z][a-z]+)*)"
def extract_person_position_org(text):
    match = re.search(pattern, text)
    if match:
        return match.groups()  # Returns (Person, Position, Organization)
    return None
sentences = [
    "George Marshall, Secretary of State of the United States.",
    "John Doe, Chief Executive Officer of TechCorp.",
    "Alice Brown, Vice President of Marketing of Global Enterprises."
]

for sentence in sentences:
    result = extract_person_position_org(sentence)
    if result:
        print(f"\nSentence: {sentence}")
        print(f"Extracted: Person = {result[0]}, Relation = {result[1]}, Organization =
{result[2]}")
```

```
Sentence: George Marshall, Secretary of State of the United States.
Extracted: Person = George Marshall, Relation = Secretary, Organization = State

Sentence: John Doe, Chief Executive Officer of TechCorp.
Extracted: Person = John Doe, Relation = Chief Executive Officer, Organization = Tech

Sentence: Alice Brown, Vice President of Marketing of Global Enterprises.
Extracted: Person = Alice Brown, Relation = Vice President, Organization = Marketing
```

Relation Extraction via Supervised Learning

- **Supervised Learning**: A method where a model is trained on a dataset containing labeled entity pairs and their relationships.
- A fixed set of relations and entities is chosen.
- a training corpus is hand-annotated with the relations and entities, and the annotated text.

Data Collection & Annotation

•Gather a dataset containing text with entity pairs.

•Manually annotate the relationships between entities with predefined relation labels (or use existing labeled datasets like SemEval, TAC KBP).

```
data = [
    ("John works at Google.", "John", "Google", "works_at"),
    ("Elon founded Tesla.", "Elon", "Tesla", "founded"),
    ("Apple acquired Beats.", "Apple", "Beats", "acquired"),
]
```

Relation Extraction via Supervised Learning

Preprocessing

- **Tokenization**: Split text into words or subwords.
- **Named Entity Recognition (NER)**: Identify and classify named entities (e.g., persons, organizations, locations).
- **Part-of-Speech (POS) Tagging**: Label words with their grammatical roles.
- **Dependency Parsing**: Extract syntactic relations between words.

Feature Extraction

Some common feature types for relation extraction include:

- **Lexical Features**: Words between and around the entity pair.
- **Syntactic Features**: POS tags, dependency relations.
- **Entity-Based Features**: Named entity types, entity distance.
- **Positional Features**: Distance of words from entities.

Relation Extraction via Supervised Learning

Model Training

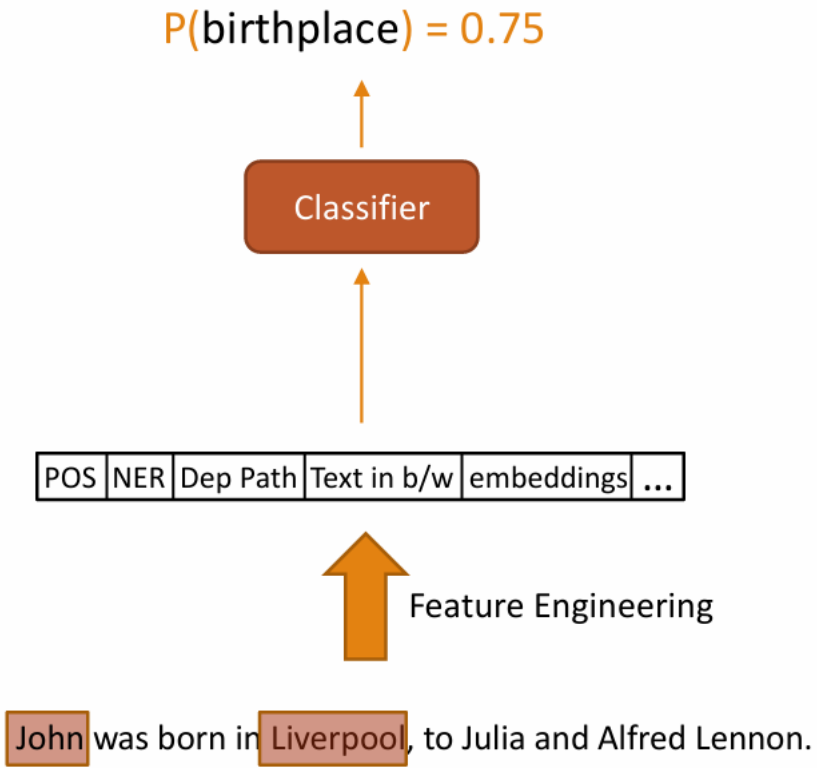
- Choose a supervised learning model:
  - **Traditional ML models**: SVM, Decision Trees, Random Forest, Logistic Regression.
  - **Deep Learning models**: CNN, BiLSTM, Transformer-based models (BERT, RoBERTa).
- Train the model on labelled data to learn patterns.

Prediction

- Apply the trained model to new text data.
- Extract relationships between entities.

Pros & Cons:

- Can get high accuracies if enough training data
- Labeling a large training set is expensive

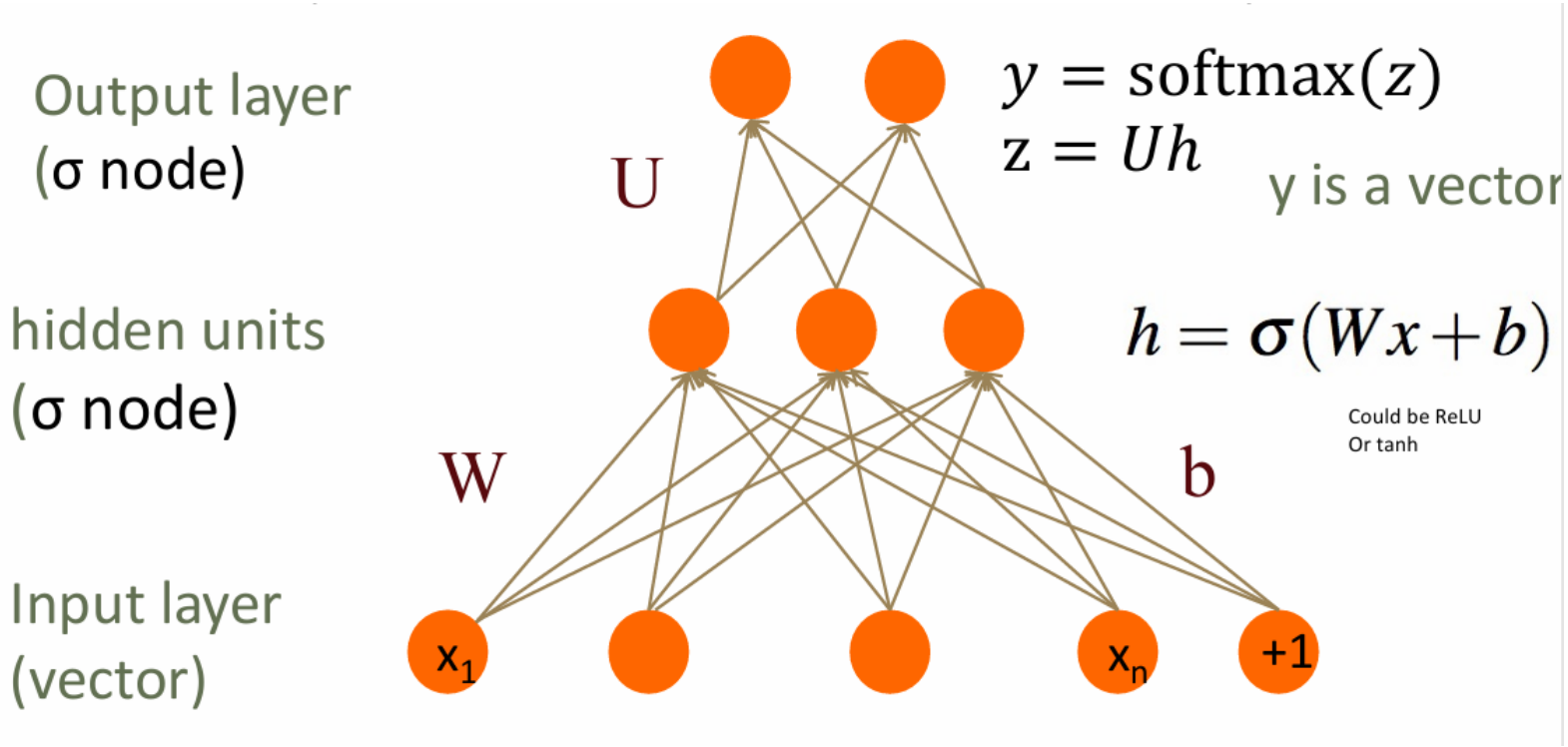


```
import nltk
import spacy
import numpy as np
from sklearn.svm import SVC
from sklearn.feature_extraction import DictVectorizer
from sklearn.pipeline import make_pipeline

# Load Spacy NLP model
nlp = spacy.load("en_core_web_sm")
# Sample training data: (sentence, entity1, entity2, relation)
train_data = [
    ("John works at Google.", "John", "Google", "works_at"),
    ("Elon founded Tesla.", "Elon", "Tesla", "founded"),
    ("Apple acquired Beats.", "Apple", "Beats", "acquired"),
    ]
# Extract features from sentences
def extract_features(sentence, entity1, entity2):
    doc = nlp(sentence)
    features = {}
    # POS tagging of words
    for token in doc:
        features[f"word_{token.text}"] = token.pos_
    # Distance between entities
    e1_idx = sentence.index(entity1)
    e2_idx = sentence.index(entity2)
    features["entity_distance"] = abs(e1_idx - e2_idx)
    # Dependency parsing
    for token in doc:
        features[f"dep_{token.text}"] = token.dep_
    return features

X_train = []
y_train = []

for sentence, e1, e2, relation in train_data:
    features = extract_features(sentence, e1, e2)
    X_train.append(features)
    y_train.append(relation)

# Convert features to numerical form
vectorizer = DictVectorizer(sparse=False)
X_train_vectorized = vectorizer.fit_transform(X_train)

# Train SVM model
svm_clf = SVC(kernel="linear", probability=True)
svm_clf.fit(X_train_vectorized, y_train)

# Function to predict relation in a new sentence
def predict_relation(sentence, entity1, entity2):
    features = extract_features(sentence, entity1, entity2)
    features_vectorized = vectorizer.transform([features])
    prediction = svm_clf.predict(features_vectorized)
    return prediction[0]

# Test the model with a new sentence
test_sentence = "Mark acquired Facebook."
print(predict_relation(test_sentence, "Mark", "Facebook"))
```

O/P: acquired

Feedforward Neural Network (FNN)

- A **Feedforward Neural Network (FNN)** is a type of artificial neural network where information moves in one direction—from the input nodes, through the hidden nodes and to the output nodes—without cycles or loops.
- It consists of an **input layer, hidden layers,** and an **output layer**.
- The network learns by adjusting weights through **backpropagation**.



How Feedforward Neural Networks Work

Feedforward Phase:

- In this phase, the input data is fed into the network, and it propagates forward through the network.
- At each hidden layer, the weighted sum of the inputs is calculated and passed through an activation function.
- This process continues until the output layer is reached, and a prediction is made.

Backpropagation Phase:

- Once a prediction is made, the error (difference between the predicted output and the actual output) is calculated.
- This error is then propagated back through the network, and the weights are adjusted to minimize this error.
- The process of adjusting weights is typically done using a gradient descent optimization algorithm.

Issues relating to sequential data

- FNNs process inputs independently without considering past information.
- FNNs do not have any memory of past inputs; each input is processed in isolation.
- In FNNs, each input has its own set of parameters, making learning inefficient for sequences.

Sequence Models

- Sequence Models are neural network architectures that process sequence data.
- Sequential data is data—such as words, sentences, or time-series data
- Applications of Sequence Models are in Speech Recognition, Machine Translation, Music Generation, Sentiment classification, Image captioning etc.

1. Recurrent Neural Networks(RNN),
2. Gated Recurrent Units(GRU),
3. Long-short-term Memory(LSTM),
4. Transformers

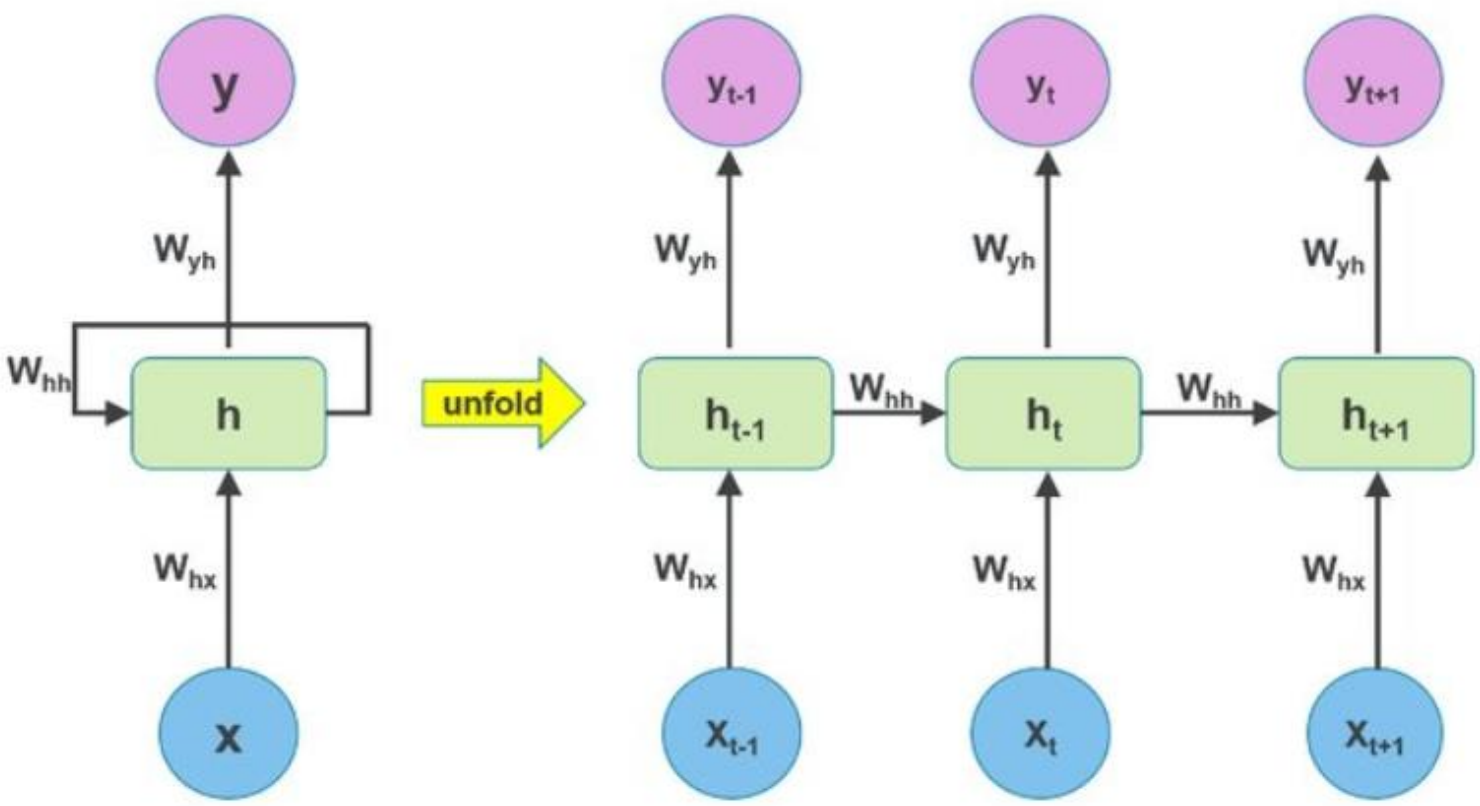
Recurrent neural networks

- Recurrent Neural Networks (RNNs) are a type of neural network architecture designed to handle sequential data, where the order of the data matters.
- RNNs are designed to process sequential data by maintaining a hidden state that captures information about previous inputs
- RNN has a concept of “memory” which remembers all information about what has been calculated till time step t.
- RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being depended on the previous computations.
- The main difference between RNN and conventional ANN is that RNN has a feedback loop that goes as input into the next time step.
- The output of hidden state at time step t-1 goes as input into the next time step t.
- This way, the model is able to learn the information from previous time steps.

The Structure of RNN

A basic RNN consists of three layers:

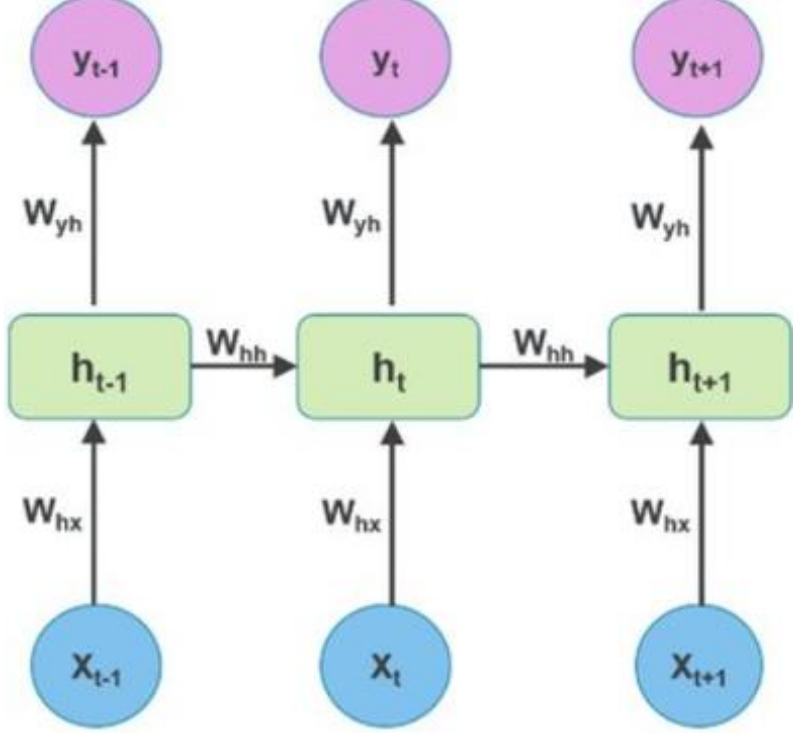
1. **Input Layer** – Takes in the current input (e.g., a word, a time-series value).
2. **Hidden Layer (Memory Unit)** – Stores past information and updates itself at each time step.
3. **Output Layer** – Produces the final prediction based on the hidden state.



Working of RNN

- RNNs work the same way as conventional ANNs.
- It has weights, bias, activation, nodes, and layers.
- We train the RNN model with multiple sequences of data, and each sequence has time steps.
- A RNN computes hidden state at each time step based on the current input and the previous hidden state.
- The output from the hidden state goes to the output layer and to the next hidden state.
- While working with sequential data, the output at any time step(t) should depend on the input at that time step as well as previous time steps.

Working of RNN



computation at the hidden state :

The hidden state h_t is computed as:

$$h_t = \sigma_h(W_{hx}x_t + W_{hh}h_{t-1} + b_h)$$

where:

- W_{hx} is the input-to-hidden weight matrix.
- W_{hh} is the recurrent weight matrix (hidden-to-hidden).
- b_h is the bias term.
- σ_h is the **activation function** (commonly **tanh** or **ReLU**).

computation at the output :

The output at time t is computed as:

$$y_t = \sigma_o(W_{yh}h_t + b_y)$$

where:

- W_{yh} is the hidden-to-output weight matrix.
- b_y is the bias term.
- σ_o is the **activation function** (depends on the task).

Working of RNN

Compute Loss

If we use **Mean Squared Error (MSE)**, the loss at each time step is:

$$\mathcal{L}_t = \frac{1}{2}(y_t - \hat{y}_t)^2$$

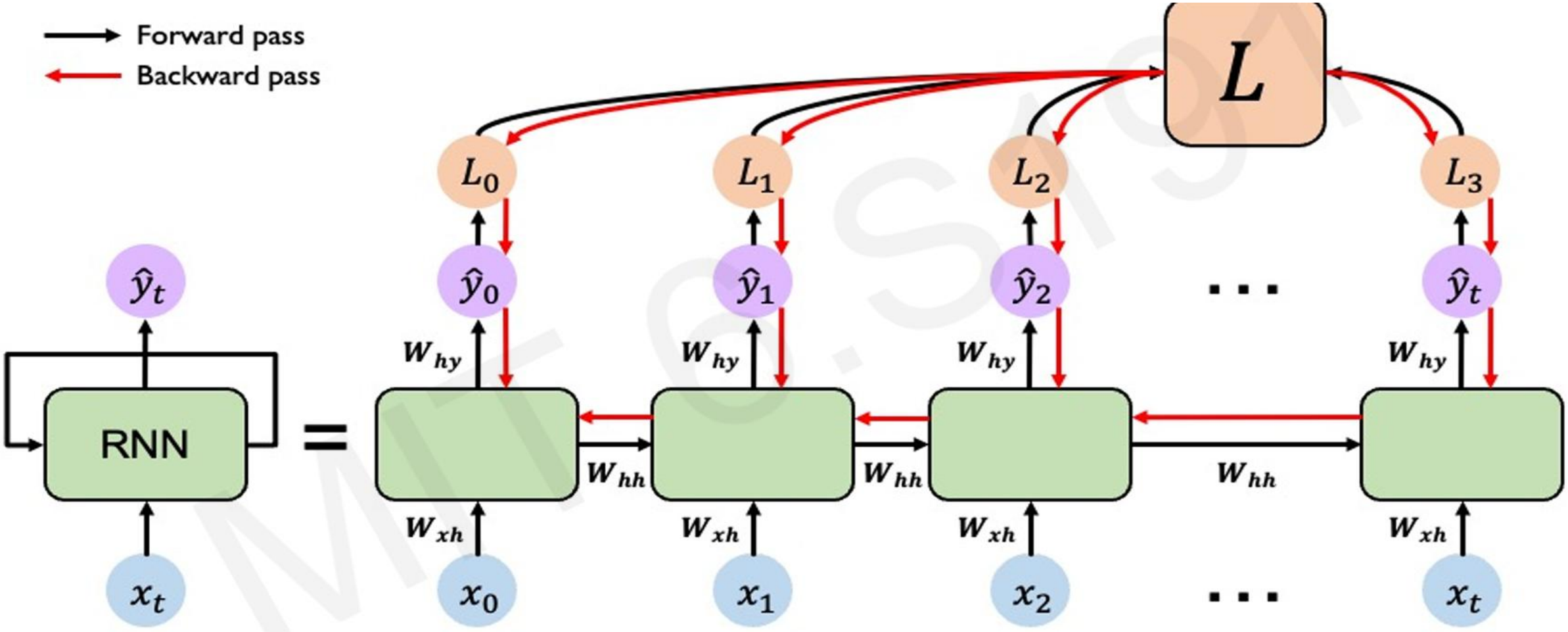
Thus, the total loss for the sequence is:

$$\mathcal{L} = \sum_{t=1}^{T} \frac{1}{2}(y_t - \hat{y}_t)^2$$

Working of RNN

Backward Propagation Through Time (BPTT):

- Backward Propagation Through Time (BPTT) is an extension of standard backpropagation used to train **Recurrent Neural Networks (RNNs)**.



Working of RNN

Backward Propagation Through Time (BPTT):

1. **Forward Pass**: Compute the hidden states and predictions for each time step.
2. **Loss Calculation**: Compute L by summing the losses at each time step.
3. **Backward Pass (Gradient Computation)**:
    1. Compute gradients of the loss w.r.t. the outputs.
    2. Backpropagate through time to update weights using the chain rule.
4. **Weight Update**: Update parameters using gradient descent (or another optimizer).

Working of RNN

Weight Updates Using Gradient Descent

Using the computed gradients, update all weights with **Stochastic Gradient Descent (SGD)**

$$W \leftarrow W - \eta \frac{\partial \mathcal{L}}{\partial W}$$

where:

- W represents all weight matrices W_{hx}, W_{hh}, W_{yh}.
- η is the learning rate.

Repeat for Multiple Epochs:

- Forward and backward propagation are repeated for multiple epochs.
- The model learns patterns and improves accuracy over time.

Types of RNN

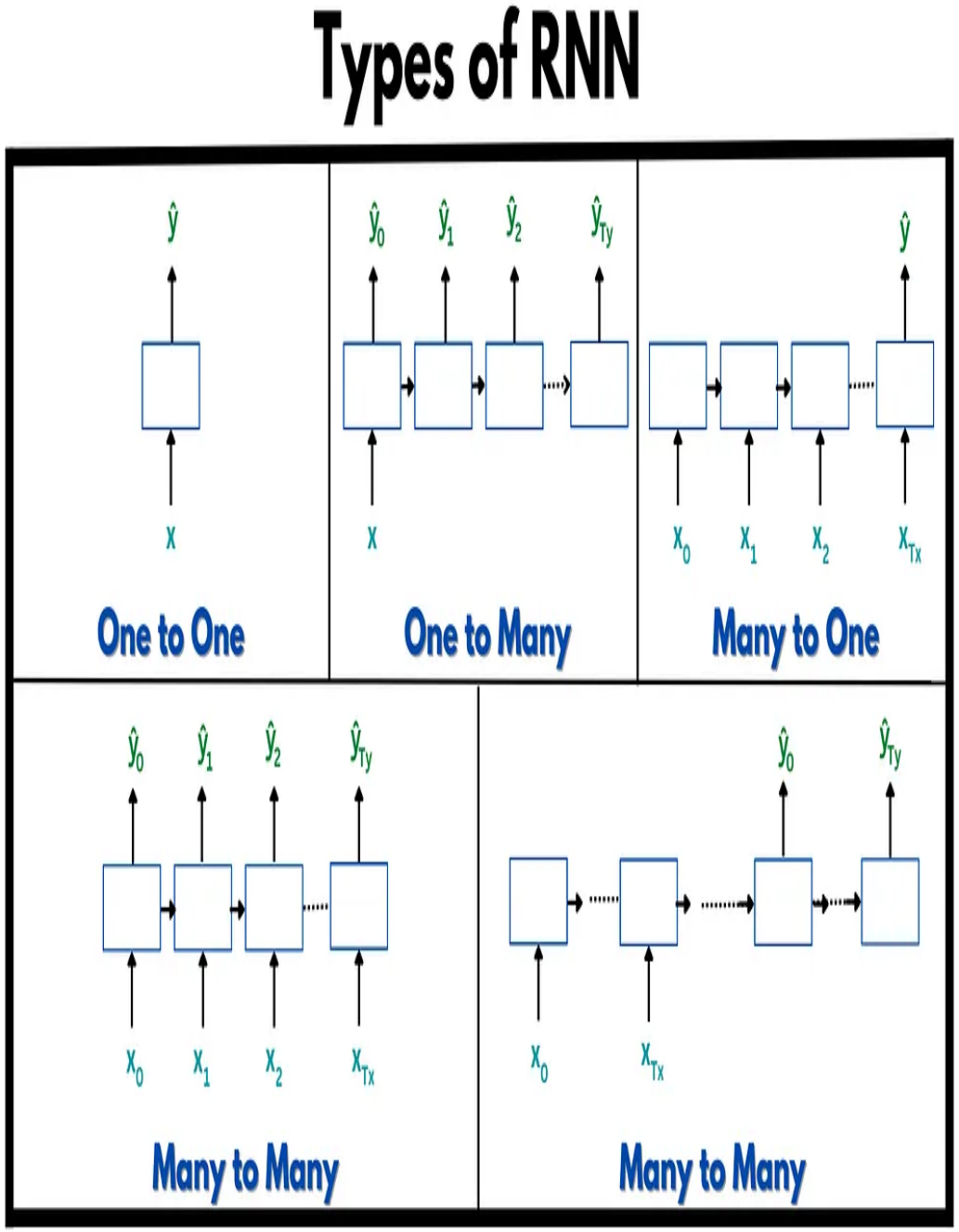
Recurrent Neural Networks (RNNs) have different architectures based on how inputs and outputs are structured.

One-to-One (Vanilla Neural Network)

1. **Input:** Single input
2. **Output:** Single output
3. **Example:** Image classification (e.g., CNNs on MNIST)

One-to-Many

1. **Input:** Single input
2. **Output:** Sequence of outputs
3. **Use Case:** Sequence generation
   Image captioning



Types of RNN

Many-to-One

1. **Input:** Sequence of inputs
2. **Output:** Single output
3. **Use Case:** Sentiment analysis, text classification

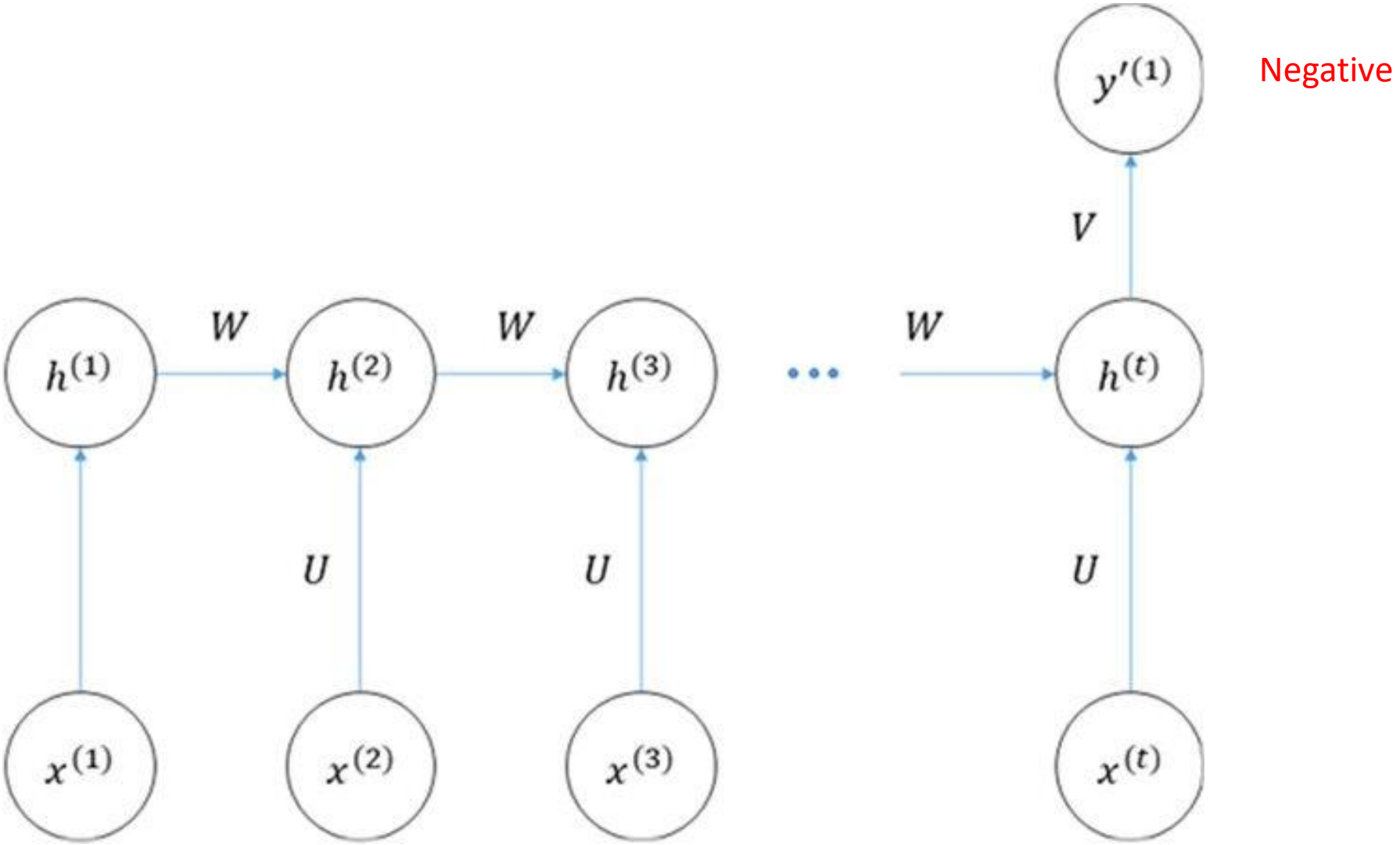
Many-to-Many (Same Length)

1. **Input:** Sequence of inputs
2. **Output:** Sequence of outputs (same length)
3. **Use Case:** POS tagging, Named Entity Recognition (NER)

Many-to-Many (Different Lengths, Encoder-Decoder)

1. **Input:** Sequence of inputs
2. **Output:** Sequence of outputs (different length)
3. **Use Case:** Machine Translation, Speech-to-Text

RNN for Text classification tasks



Issues in RNNs

- Vanishing Gradient Problem
- Exploding Gradient Problem
- Long-Term Dependencies Issue

Vanishing and Exploding Gradients problem in RNNs

Vanishing Gradient Problem

- In RNNs, gradients are propagated backward through many time steps.
- If the weight matrices have small values, repeated multiplication causes the gradients to shrink exponentially.
- Eventually, gradients become so small (close to zero) that earlier layers in the network stop updating weights.

Exploding Gradient Problem

- If the weight matrices have large values, repeated multiplication causes gradients to grow exponentially.
- The model weights may become NaN during training.

Long-Term Dependencies Issue

- The **long-term dependencies issue** in Recurrent Neural Networks (RNNs) arises from their difficulty in learning and retaining information over long sequences.
- Over long sequences, the gradients (error signals) can shrink exponentially, leading to minimal updates for earlier layers.
- This makes it hard for the network to learn long-range dependencies

Example:

Let's take this sentence.

The Sun rises in the _____.

- An RNN could easily return the correct output that the sun rises in the East as all the necessary information is nearby.

Let's take another example.

I was born in Japan, …… and I speak fluent _____.

- In this sentence, the RNN would be unable to return the correct output as it requires remembering the word Japan for a long duration.
- Since RNN only has a "Short-term" memory, it doesn't work well..

How RNN Issues Are Addressed:

1. Advanced RNN Architectures:
    1.LSTM (Long Short-Term Memory):
    2.GRU (Gated Recurrent Units):
2.Attention Mechanisms:
3.Transformers:

LSTM

- LSTM (Long Short-Term Memory) is a type of **Recurrent Neural Network (RNN)** designed to handle **sequential data** and capture **long-term dependencies**.

Why LSTMs Are Effective

- Traditional RNNs struggle with learning **long-term dependencies** due to issues like the **vanishing gradient problem**.
- LSTMs solve this by using **gates** to **control the flow of information**, ensuring that **relevant data** is **retained** or **discarded** as needed.

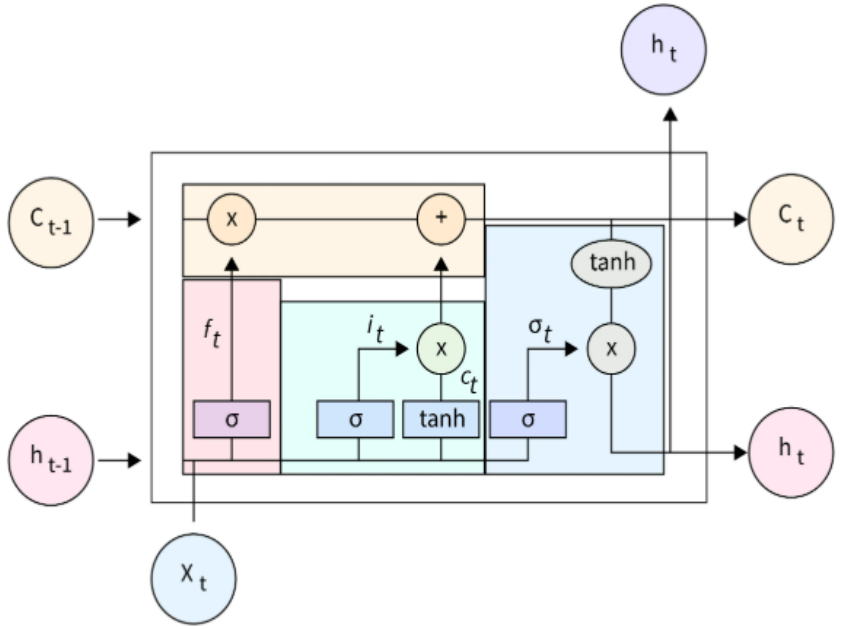
Key components of LSTM

- **Memory Cell/cell state** – Stores relevant information over long sequences.
- **Forget Gate** – Decides which past information to discard.
- **Input Gate** – Determines what new information to store.
- **Output Gate** – Controls what is passed to the next step.

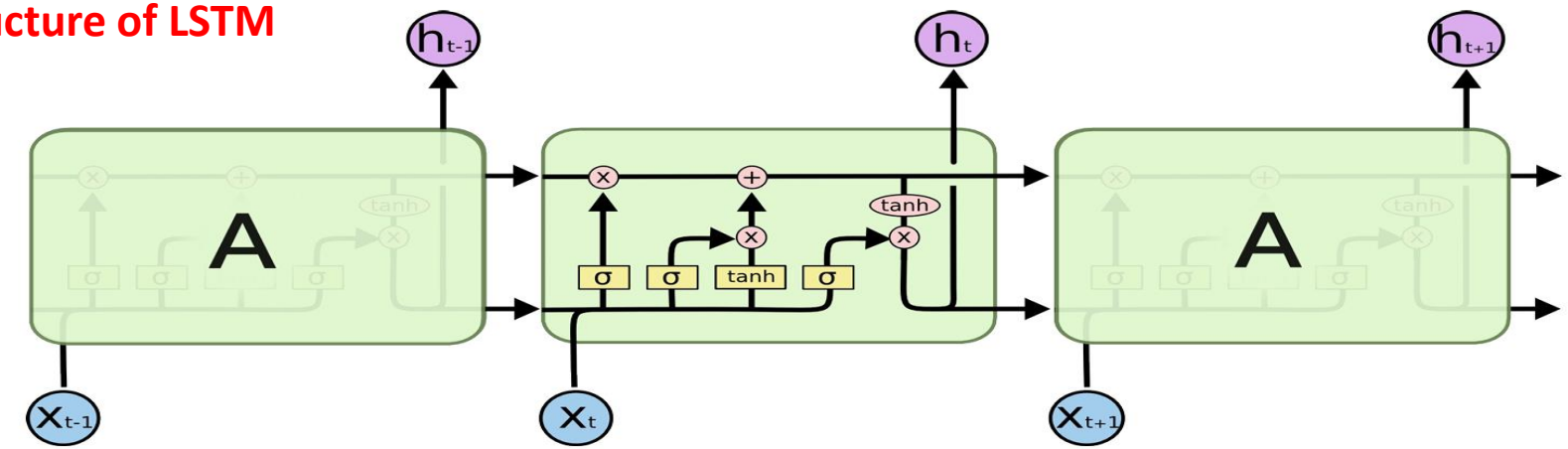
Structure of LSTM

- Each LSTM cell consists of 3 gates A **forget gate**, **input gate**, and **output gate**.
- The cell has two states **Cell State** and **Hidden State**.
- They are continuously updated and carry the information from the previous to the current time steps.
- The cell state is the “long-term” memory, while the hidden state is the “short-term” memory.

Structure of each LSTM cell



Structure of LSTM



Structure of LSTM

Forget Gate:

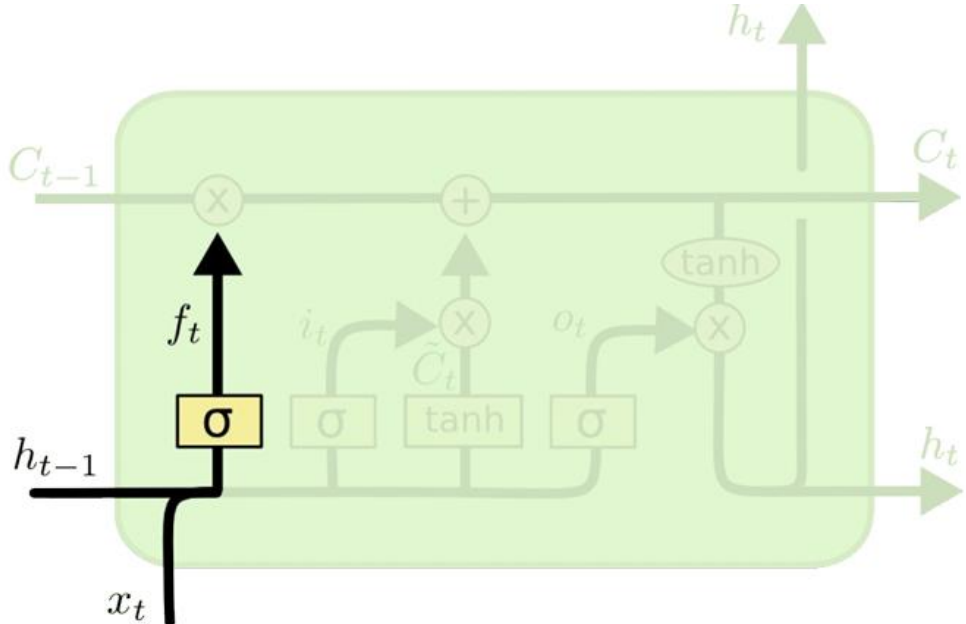
- Forget gate is responsible for deciding what past information should be **removed from the cell state**.

Mathematical Representation

The forget gate takes the **previous hidden state** (h_{t-1}) and the **current input** (x_t), applies a **sigmoid activation function** (σ), and outputs a value between **0 and 1**:

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

- W_f → Weight matrix for the forget gate
- b_f → Bias term
- σ → Sigmoid activation (outputs values between 0 and 1)
- f_t → Forget gate output (indicates how much memory to keep)



How It Works

- If f_t is **close to 0**, the LSTM **forgets** the information.
- If f_t is **close to 1**, the LSTM **retains** the informat

Structure of LSTM

Input Gate in LSTM

- The **Input Gate** in an LSTM controls **what new information should be added** to the cell state.
- It helps the model **update its memory** by selecting relevant information from the current input.

1. **Deciding what to update:**

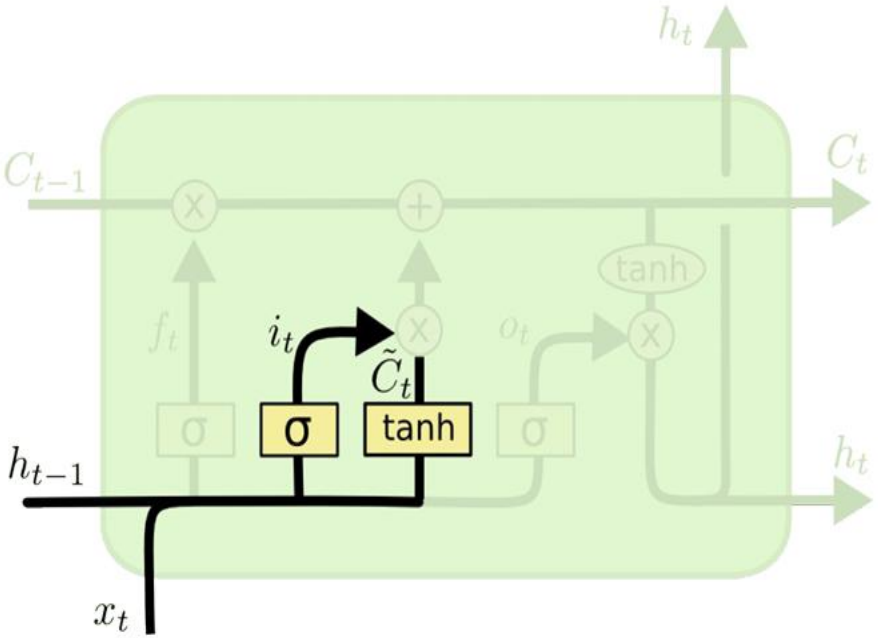
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

- i_t → Input gate activation (decides how much information to add)
- W_i, b_i → Learnable weight matrix and bias
- h_{t-1} → Previous hidden state
- x_t → Current input
- σ → Sigmoid activation function (values between 0 and 1)

2. **Creating candidate values for memory update:**

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

- $\tilde{C}_t$ → Candidate memory content (new information to be added)
- W_C, b_C → Weight matrix and bias



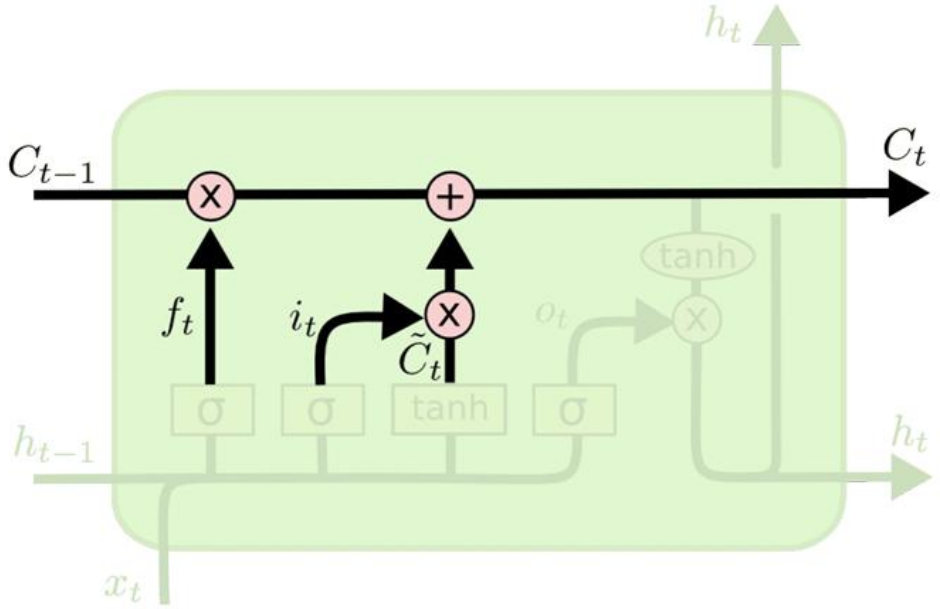
Structure of LSTM

Cell State:

3. **Updating the cell state:**

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

- C_t → Updated cell state
- f_t → Forget gate output
- i_t → Input gate output (controls new info addition)
- $\tilde{C}_t$ → Candidate values for update
- $\odot$ → Element-wise multiplication



How It Works

- If i_t is **close to 1**, the LSTM **adds new information** to the memory.
- If i_t is **close to 0**, it **ignores new information** and relies on past memory.

Structure of LSTM

Output Gate in LSTM

- The **output gate** in an LSTM determines what part of the information in the memory cell should be passed on to the next hidden state and used as the output for the current step.

The output gate works in two steps:

1. **Deciding what to output:**

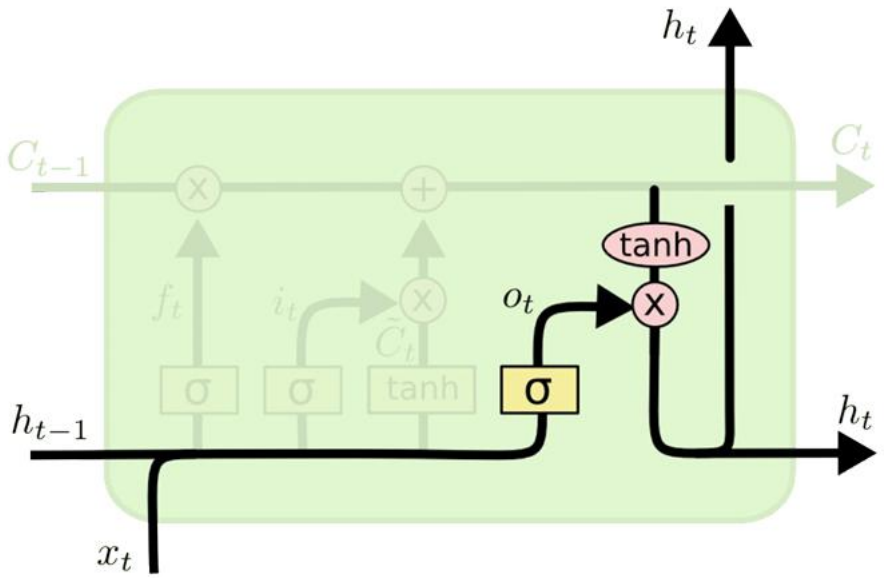
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

- o_t → Output gate activation (controls how much information from C_t is used)
- W_o, b_o → Weight matrix and bias for the output gate
- h_{t-1} → Previous hidden state
- x_t → Current input
- σ → Sigmoid activation function (values between 0 and 1)

2. **Computing the new hidden state:**

$$h_t = o_t \odot \tanh(C_t)$$

- h_t → New hidden state (used as output and passed to the next time step)
- o_t → Output gate activation (determines how much of C_t is exposed)
- C_t → Current cell state (long-term memory)



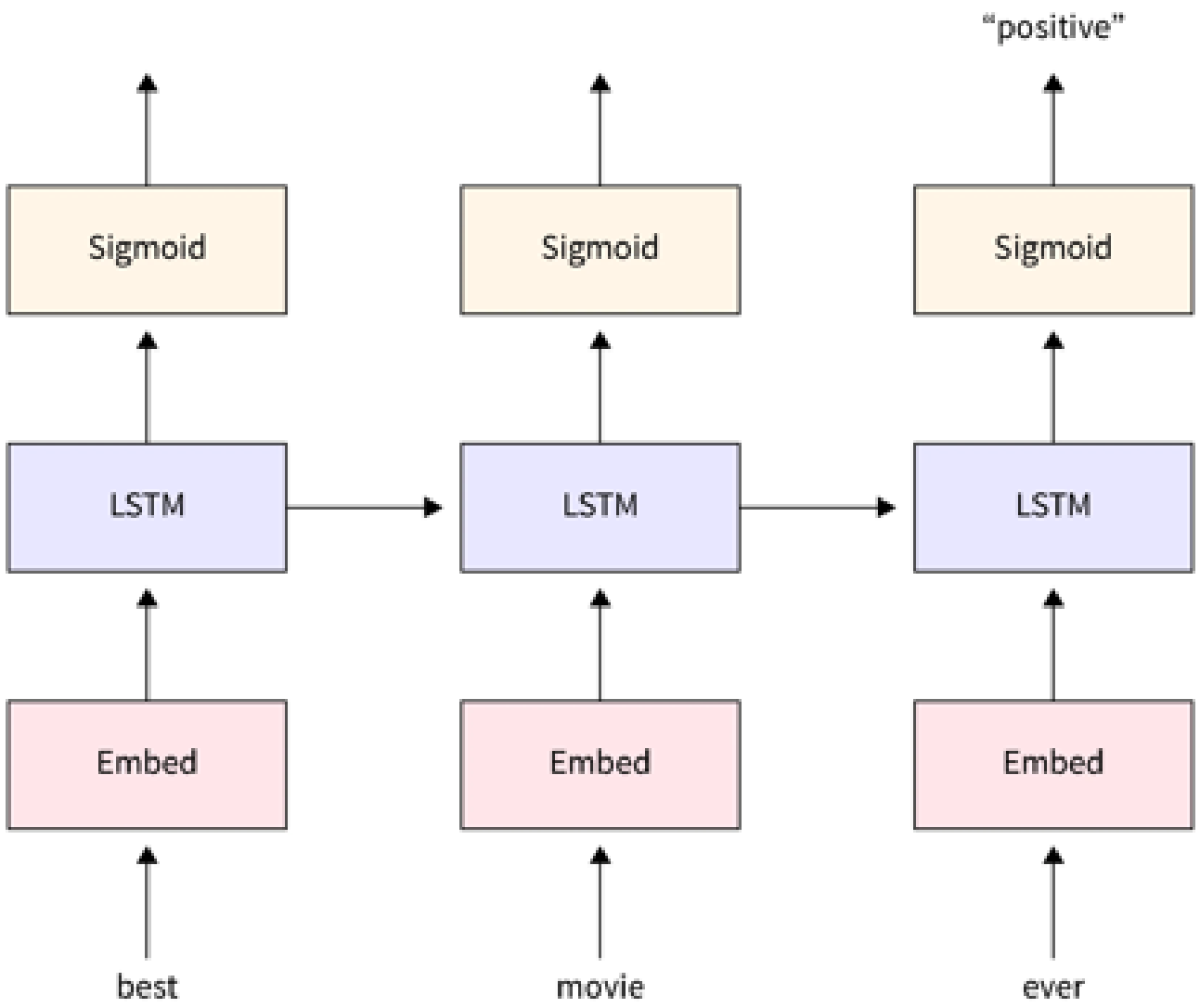
How do LSTMs Work?

- The LSTM architecture is similar to RNN, but instead of the feedback loop has an LSTM cell.
- The sequence of LSTM cells in each layer is fed with the **output of the last cell**.
- This enables the cell to get the previous inputs and sequence information.
- A cyclic set of steps happens in each LSTM cell
  - The Forget gate is computed.
  - The Input gate value is computed.
  - The Cell state is updated using the above two outputs.
  - The output(hidden state) is computed using the output gate.
- The intuition behind LSTM is that the Cell and Hidden states carry the previous information and pass it on to future time steps.
- The **Cell state** is **aggregated** with all the past data information and is the **long-term** information retainer.
- The Hidden state carries the output of the last cell, i.e. **short-term memory**.
- This combination of Long term and short-term memory techniques enables LSTM's to perform well In time series and sequence data.

LSTM Applications in NLP

- **Text Classification** (e.g., spam detection, sentiment analysis)
- **Named Entity Recognition (NER)**
- **Part-of-Speech (POS) Tagging**
- **Machine Translation**
- **Speech Recognition**
- **Chatbots & Conversational AI**

LSTM for text classification



GRU (Gated Recurrent Unit)

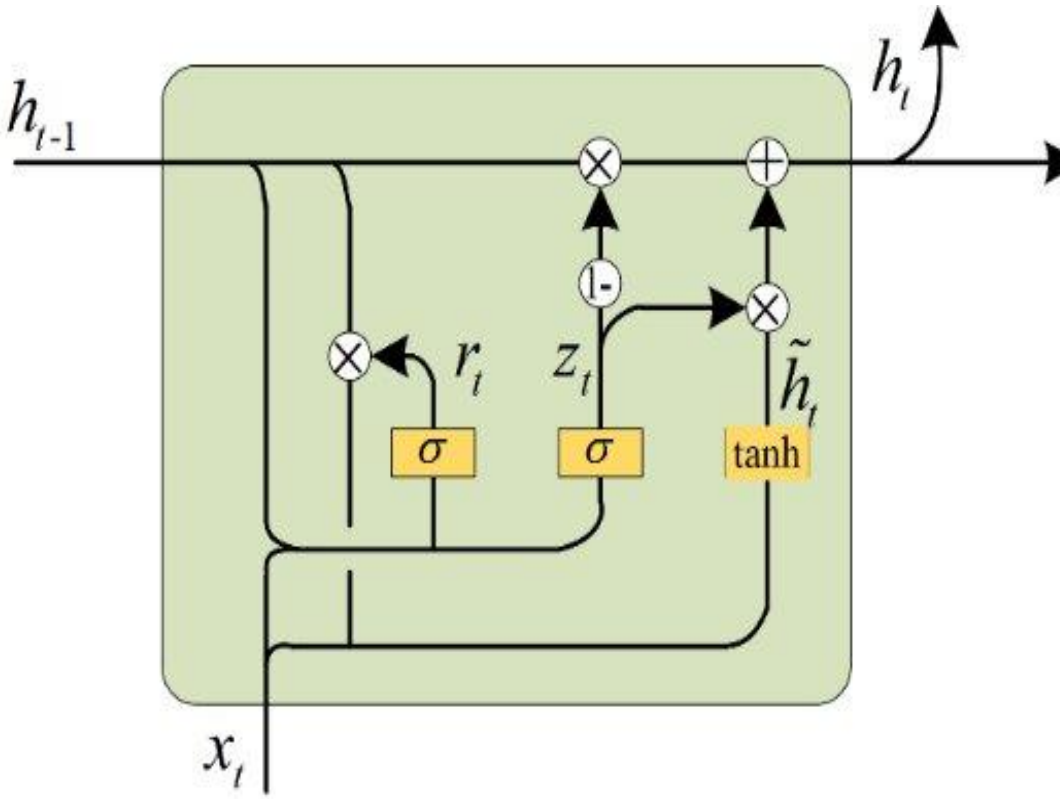
- GRU is a type of recurrent neural network (RNN) architecture designed to handle sequential data while addressing the vanishing gradient problem.
- A **Gated Recurrent Unit (GRU)** solves the **vanishing gradient problem** by using **gates** to control the flow of information.
- It is similar to LSTMs (Long Short-Term Memory networks) but has a simpler structure, making it computationally more efficient.
- Unlike LSTM, GRU has only one State i.e Hidden State and only two Gates i.e Update Gate and Reset Gate

How GRU Works?

- Like other recurrent neural network architectures, GRU processes sequential data one element at a time, updating its hidden state based on the current input and the previous hidden state.

- At each time step, GRU computes
  - reset gate **(r_t)**, which determines how much of the previous hidden state to forget,
  - update gate **(z_t)**, which determines how much of the past information and how much of the candidate activation vector to incorporate into the new hidden state.
  - candidate activation vector **($\tilde{h}_t$)** that combines information from the input and the previous hidden state.
  - Final hidden state **(h_t)** based on candidate activation vector ,update gate and passed to next time step

Structure of each GRU cell



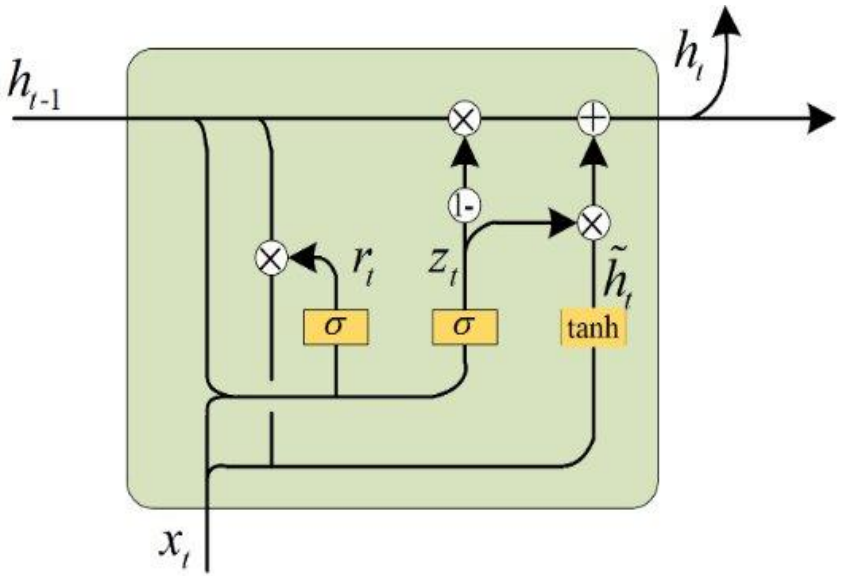
Reset gate (r_t) in GRU

- The **reset gate** in a **Gated Recurrent Unit (GRU)** is responsible for controlling How much of the past information to be forgotten before computing the new candidate activation

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

Where:

- W_r = Weight matrix for the reset gate
- h_{t-1} = Previous hidden state
- x_t = Current input
- b_r = Bias for the reset gate
- σ = **Sigmoid activation function** (output between 0 and 1)



How the Reset Gate Works

- If r_t **is close to 0**, the model **ignores most of the past hidden state** (h_{t-1}).
- If r_t **is close to 1**, the model **retains the past hidden state** for further computation.

The reset gate helps the GRU decide whether **past information is still relevant** for predicting the next state.

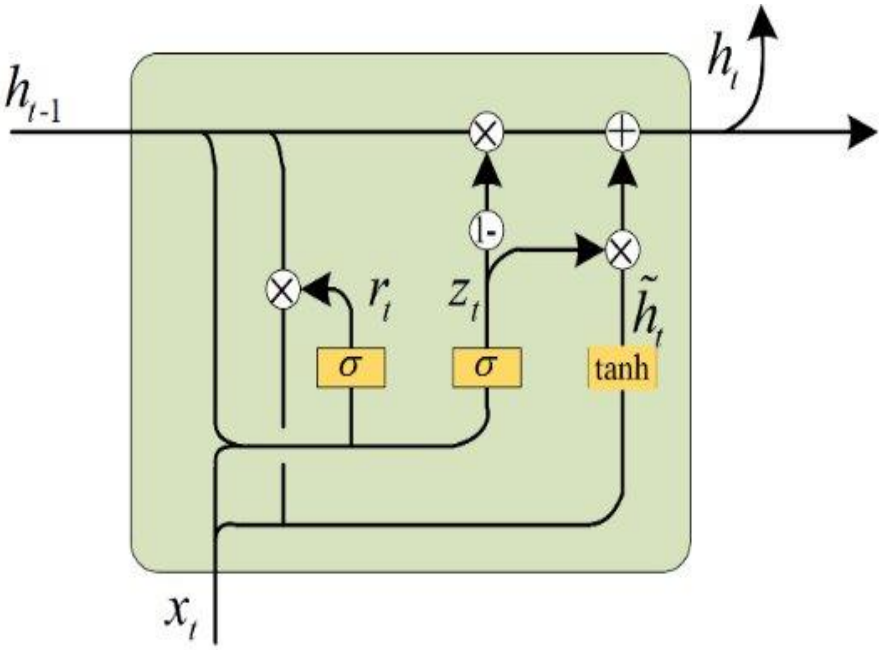
update gate (z_t) in GRU

- The **update gate** in a **Gated Recurrent Unit (GRU)** controls how much of the **previous information** should be **carried forward** to the next time step.

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$

Where:

- W_z = Weight matrix for the update gate
- h_{t-1} = Previous hidden state
- x_t = Current input
- b_z = Bias for the update gate
- σ = **Sigmoid activation function** (output between 0 and 1)



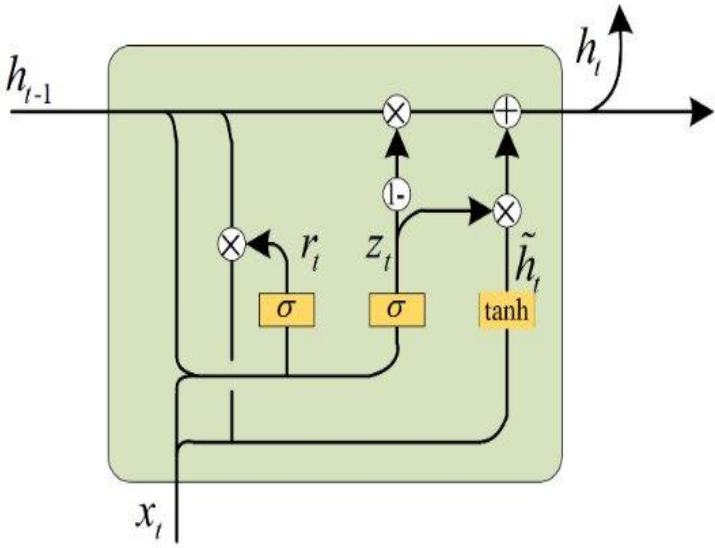
Candidate Activation Vector(Current Memory Content)

- Consider the current input and relevant information of the previous hidden state to produce new internal information.

$$\tilde{h}_t = \tanh(W_h \cdot [r_t * h_{t-1}, x_t] + b_h)$$

Where:

- W_h = Weight matrix for candidate activation
- h_{t-1} = Previous hidden state
- x_t = Current input
- r_t = **Reset gate**, which determines how much past memory to use
- b_h = Bias for candidate activation
- $\tanh$ = **Tanh activation function** (range: -1 to 1)



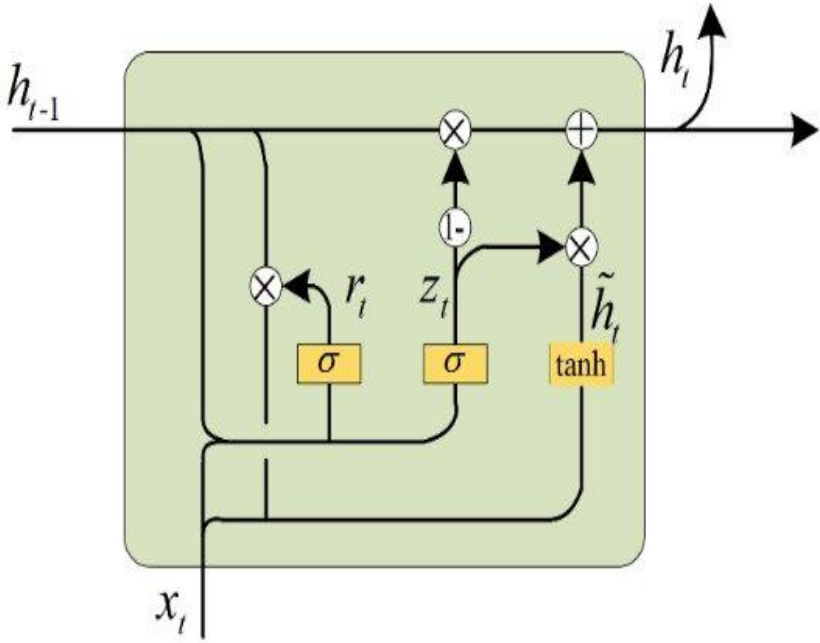
How It Works

1. The **reset gate** r_t decides how much past memory (h_{t-1}) should be **forgotten**.
2. The model **computes a new memory candidate** ($\tilde{h}_t$) using the modified past state and the current input x_t.
3. The **update gate** (z_t) later decides **how much of this new memory** should be used in the final hidden state.

Final hidden State (h_t) in GRU

- The final hidden state (h_t) is the actual output of the GRU cell at time step t_t.

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$



Where:

- h_t = Final hidden state (output of GRU at time t)
- h_{t-1} = Previous hidden state
- $\tilde{h}_t$ = Candidate activation (new memory content)
- z_t = Update gate (controls how much of the new memory is used)

How It Works

1. The **update gate** z_t determines how much of the **old state** (h_{t-1}) is kept and how much of the **new candidate state** ($\tilde{h}_t$) is incorporated.
2. If z_t **is close to 1**, more of the **new candidate activation** ($\tilde{h}_t$) is used.
3. If z_t **is close to 0**, more of the **previous hidden state** (h_{t-1}) is retained.
4. The final hidden state h_t is passed to the **next time step** or **used as the output**.

Why is the Final Hidden State Important?

- It **stores memory** of previous inputs.
- It **helps in sequence prediction** (e.g., text generation, speech processing).
- It **is passed to the next GRU unit** or **used in classification tasks.**

All GRU Equations

1. Reset Gate:

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

2. Update Gate:

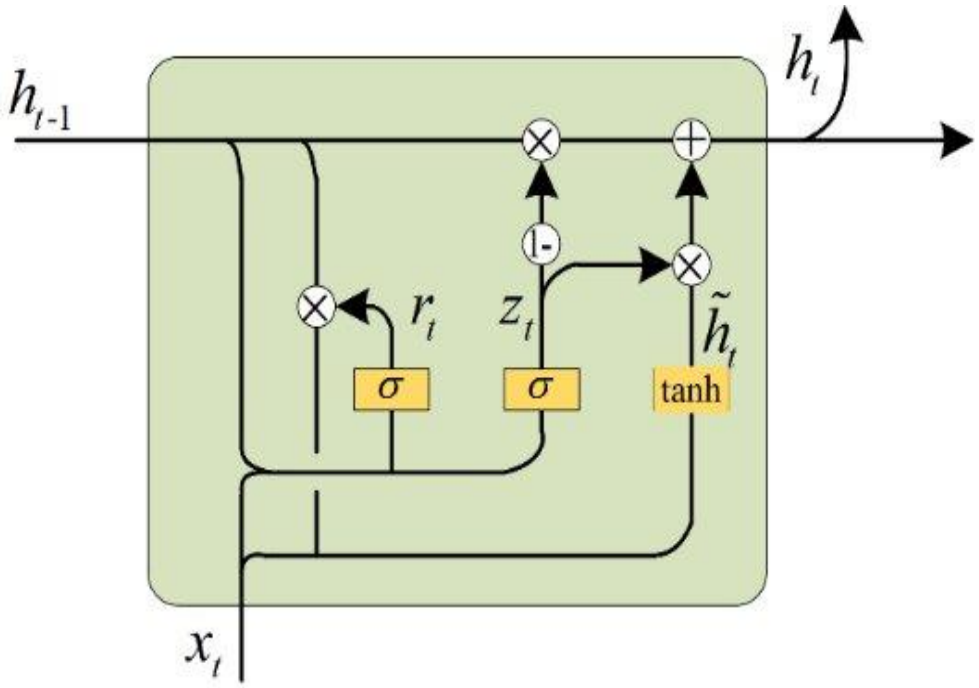
$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$

3. Candidate Activation (New Memory Content):

$$\tilde{h}_t = \tanh(W_h \cdot [r_t * h_{t-1}, x_t] + b_h)$$

4. Final Hidden State:

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$



GRU vs. LSTM: Key Differences

Feature	GRU (Gated Recurrent Unit)	LSTM (Long Short-Term Memory)
Number of Gates	2 (Reset Gate, Update Gate)	3 (Forget Gate, Input Gate, Output Gate)
Memory Cell	No separate memory cell (c_t)	Has a dedicated memory cell (c_t)
Computational Complexity	Lower (Fewer parameters)	Higher (More parameters)
Training Speed	Faster	Slower
Performance on Small Datasets	Works well	May need more data
Ability to Capture Long-Term Dependencies	Moderate	Stronger due to explicit memory cell
Vanishing Gradient Handling	Good	Better due to forget gate
Architecture Simplicity	Simpler	More complex

Sequence-to-Sequence (Seq2Seq) Models in NLP

- The **Seq2Seq (Sequence-to-Sequence)** model is a neural network architecture designed to handle sequential data, transforming one sequence into another.
- This model is particularly useful for tasks where the input and output sequences can vary in length, such as machine translation, text summarization, speech recognition and Q&A.

Applications

- **Machine Translation**: Translating text from one language to another.
- **Text Summarization**: Generating concise summaries of longer documents.
- **Speech Recognition**: Converting spoken language into written text.
- **Image Captioning**: Describing the content of an image in natural language.
- **Video Captioning**: Generating descriptive texts for video content

Sequence-to-Sequence (Seq2Seq) Models in NLP

Popular Sequence-to-Sequence (Seq2Seq) Models in NLP

- RNN-Based Seq2Seq Models (Traditional)
- Transformer-Based Seq2Seq Models (Modern)

RNN-Based Seq2Seq Models (Traditional)

- **RNNs:** Process input word by word in a sequential manner, making them slow and difficult to parallelize.
- **RNNs:** Struggle to retain information over long sequences due to **vanishing gradients**, making it hard to learn long-term dependencies.
- Even LSTMs and GRUs have difficulty capturing very long sequences.
- **RNNs:** Store hidden states for each step, leading to high memory usage for long sequences.
- **RNNs:** Require sequential computation, making them hard to scale on GPUs/TPUs.

Transformer-Based Seq2Seq Models

- Transformer-based Sequence-to-Sequence (Seq2Seq) models have revolutionized NLP with their efficient architecture and state-of-the-art performance.
- Unlike traditional Seq2Seq models reliant on recurrent networks (e.g., RNNs or LSTMs), transformer-based models leverage **self-attention** mechanisms, enabling them to process sequences in parallel and capture long-range dependencies more effectively.
- Transformers process all tokens in a sequence **simultaneously**, this allows for parallel computation, making transformers much faster.
- The self-attention mechanism enables transformers to capture **global dependencies** efficiently, regardless of sequence length. Each token attends to every other token, making them adept at understanding context over long distances
- Since transformers process tokens simultaneously, they use **positional encodings** to inject information about the order of tokens, ensuring the model understands sequence structure.

Transformer Architecture

- The **Transformer architecture**, introduced in the paper *"Attention Is All You Need"* by Vaswani et al.
- It consists of an **encoder-decoder structure**, which is the foundation of models like **BERT, GPT, and T5**.

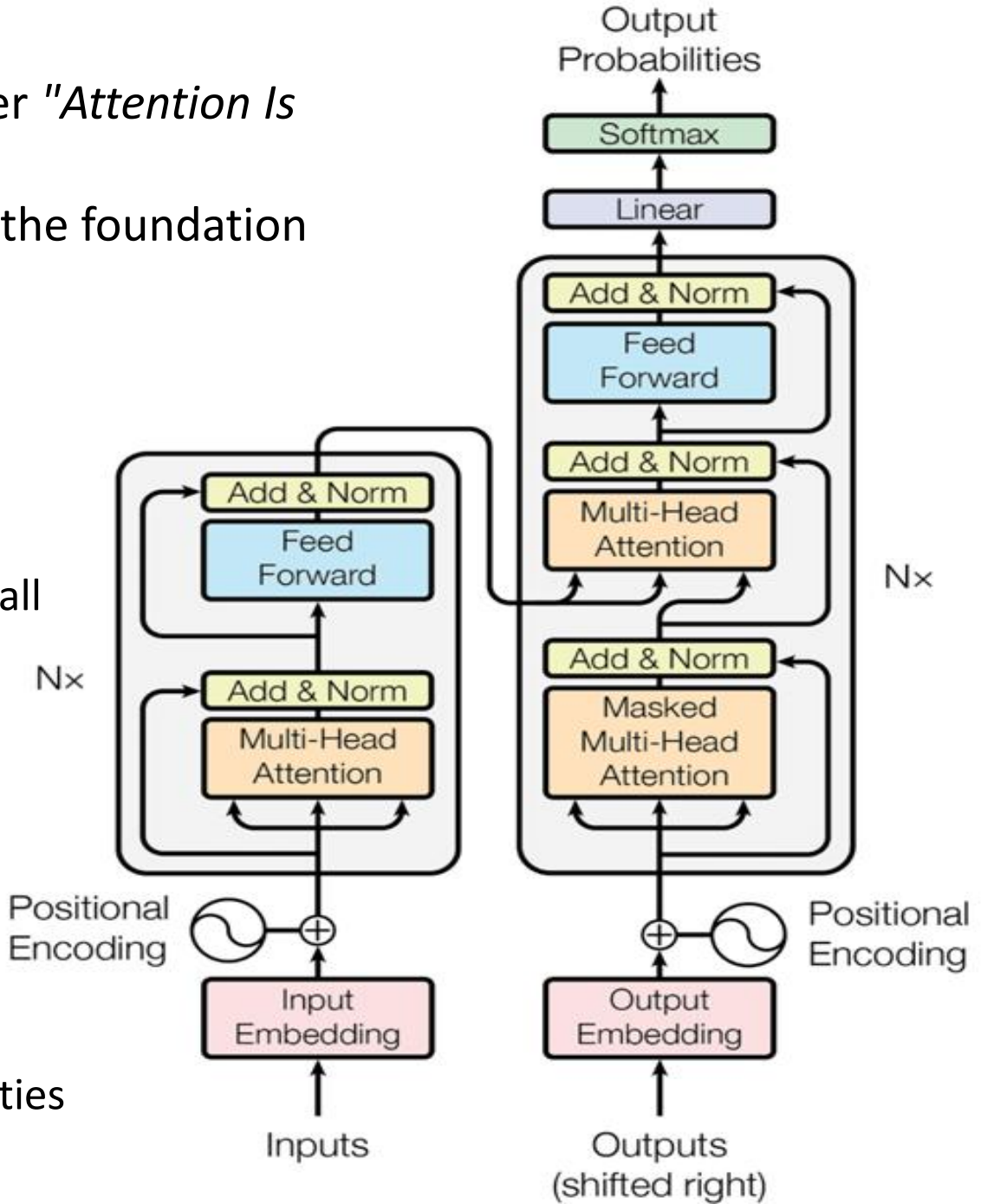
Key Components

Encoder (Left Side)

1. **Input Embedding:** Converts input tokens into dense vectors.
2. **Positional Encoding:** Adds sequence order information.
3. **Multi-Head Self-Attention:** Captures relationships between all words in the input.
4. **Feed Forward Layer:** Processes the representations.
5. **Nx Layers:** Stack of identical encoder blocks.

Decoder (Right Side)

1. **Masked Multi-Head Attention:** Prevents looking at future tokens (causal attention).
2. **Multi-Head Attention:** Attends to the encoder outputs.
3. **Feed Forward Layer:** Processes the representations.
4. **Final Linear Layer + Softmax:** Converts output into probabilities over the vocabulary.

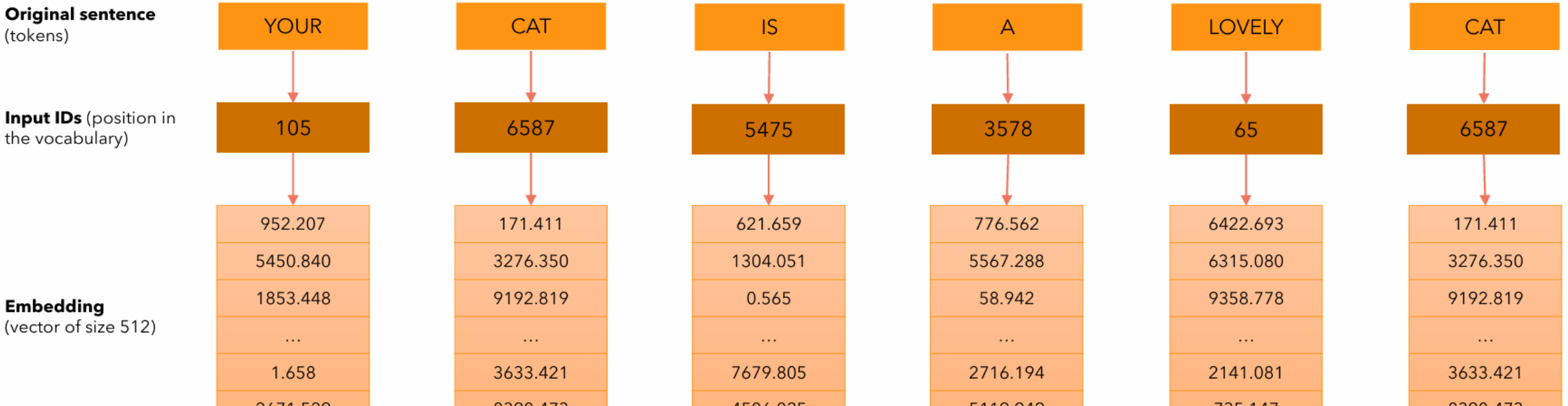


Encoder

- The encoder outputs a **context representation** of the input sentence that supposedly captures the meaning of the sentence.
- It processes the input sequence and generates contextualized representations, which are passed to the **Decoder** for further processing.

Input embedding

- An input embedding in a transformer model is a numerical representation of the input data that allows the model to process and understand it.



Positional encoding

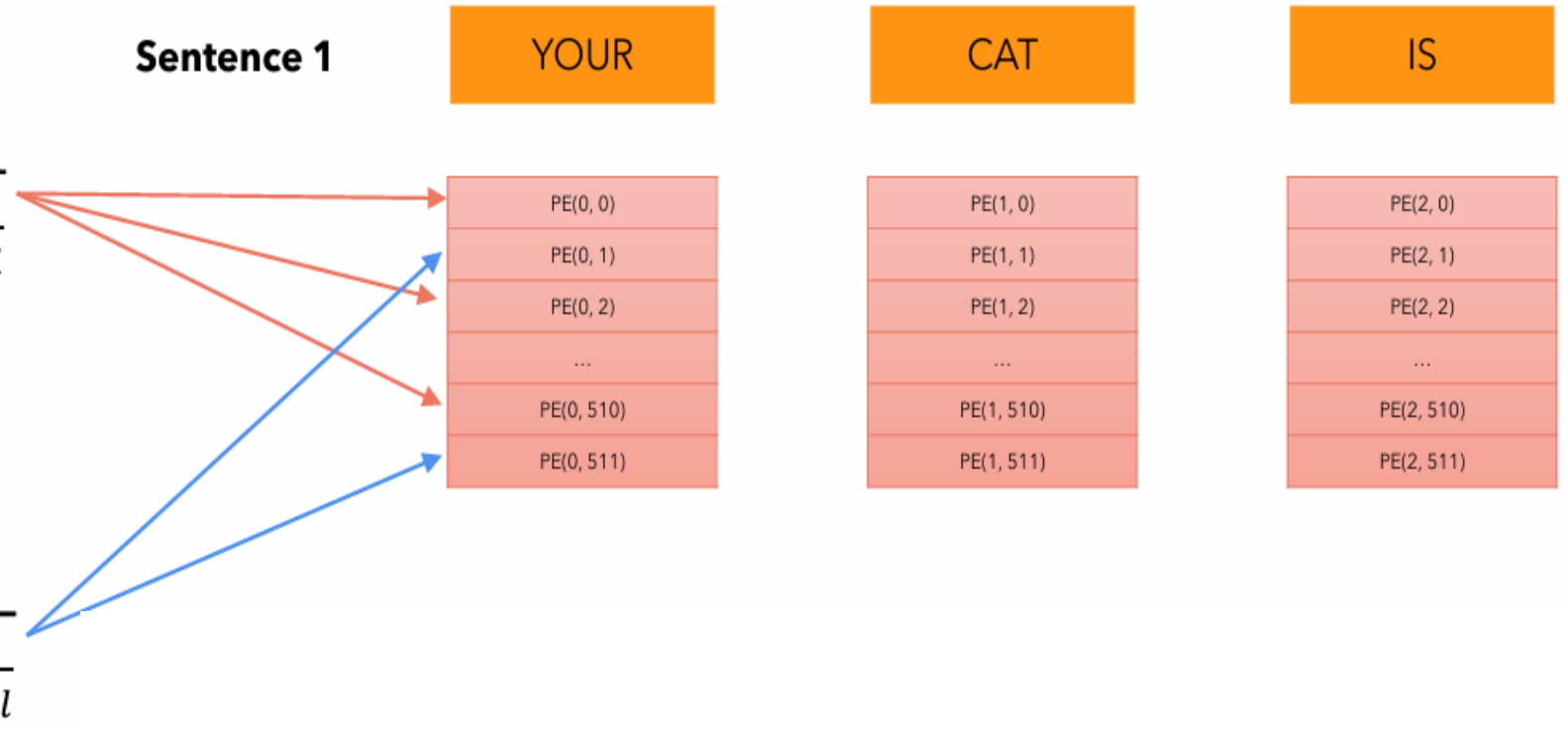
- **Transformers** process the entire sequence **in parallel**. This means they **don't have an inherent sense of word order**.
- **Positional Encoding** is used to inject **word order information** into the input embeddings so the model can distinguish between different positions in a sentence.
- We want each word to carry some information about its position in the sentence.

Positional encoding

- Transformers use a fixed **sine and cosine function** to create a **positional encoding vector** for each token.
- The **sin function** is used for even indices.
- The **cos function** is used for odd indices.

$$PE(pos, 2i) = \sin \frac{pos}{10000^{\frac{2i}{d_{model}}}}$$

$$PE(pos, 2i+1) = \cos \frac{pos}{10000^{\frac{2i}{d_{model}}}}$$



Self-Attention

- Self-attention, also known as scaled dot-product attention, is a mechanism that allows a Transformer to weigh the importance of different words in a sentence when processing a specific word.
- Given a word sequence, we recognize that some words within it are more closely related with one another than others.
- **self-attention** in which a given word "attends to" other words in the sequence. Essentially, attention is about representing context by giving weights to word relations.
- Self-Attention determines on how words in the same sentence relate to each other.

How Does Self-Attention Work?

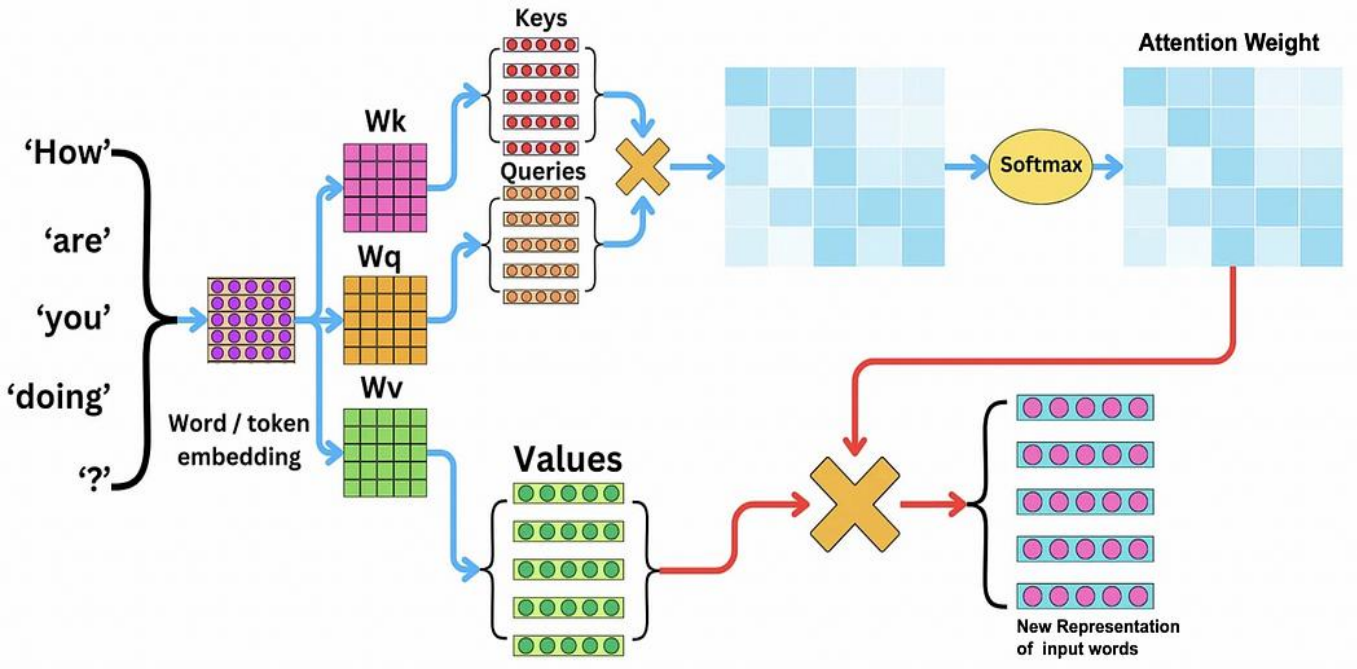
- Self-attention computes relationships between words using **three vectors per token**:

•**Query (Q)** – What this word is looking for.

•**Key (K)** – What this word has to offer.

•**Value (V)** – The actual content of the word.

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



How Does Self-Attention Work?

- We'll use weight matrices to project each vector $\mathbf{x}_i$ into a representation of its role as query, key, value:
- **Query weight matrix**: $\mathbf{W}^{\mathbf{Q}}$
- **Key weight matrix** : $\mathbf{W}^{\mathbf{K}}$
- **Value weight matrix** : $\mathbf{W}^{\mathbf{V}}$

Step-1: Create Query, Key, and Value Matrices

- Each word's embedding is transformed into **Q, K, and V** using learned weight matrices:

$$\mathbf{q}_i = \mathbf{x}_i\mathbf{W}^{\mathbf{Q}}; \quad \mathbf{k}_i = \mathbf{x}_i\mathbf{W}^{\mathbf{K}}; \quad \mathbf{v}_i = \mathbf{x}_i\mathbf{W}^{\mathbf{V}}$$

step-2: Compute Attention Scores between words

Given these 3 representation of $\mathbf{x}_i$

- To compute similarity of current element $\mathbf{x}_i$ with some prior element $\mathbf{x}_j$
- We'll use dot product between $\mathbf{q}_i$ and $\mathbf{k}_i$.

$$\mathbf{q}_i = \mathbf{x}_i\mathbf{W}^{\mathbf{Q}}; \quad \mathbf{k}_j = \mathbf{x}_j\mathbf{W}^{\mathbf{K}}; \quad \mathbf{v}_j = \mathbf{x}_j\mathbf{W}^{\mathbf{V}}$$

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}$$

Step 3: Apply Softmax

- Softmax is applied to convert the scores into probabilities:

Attention Weights:

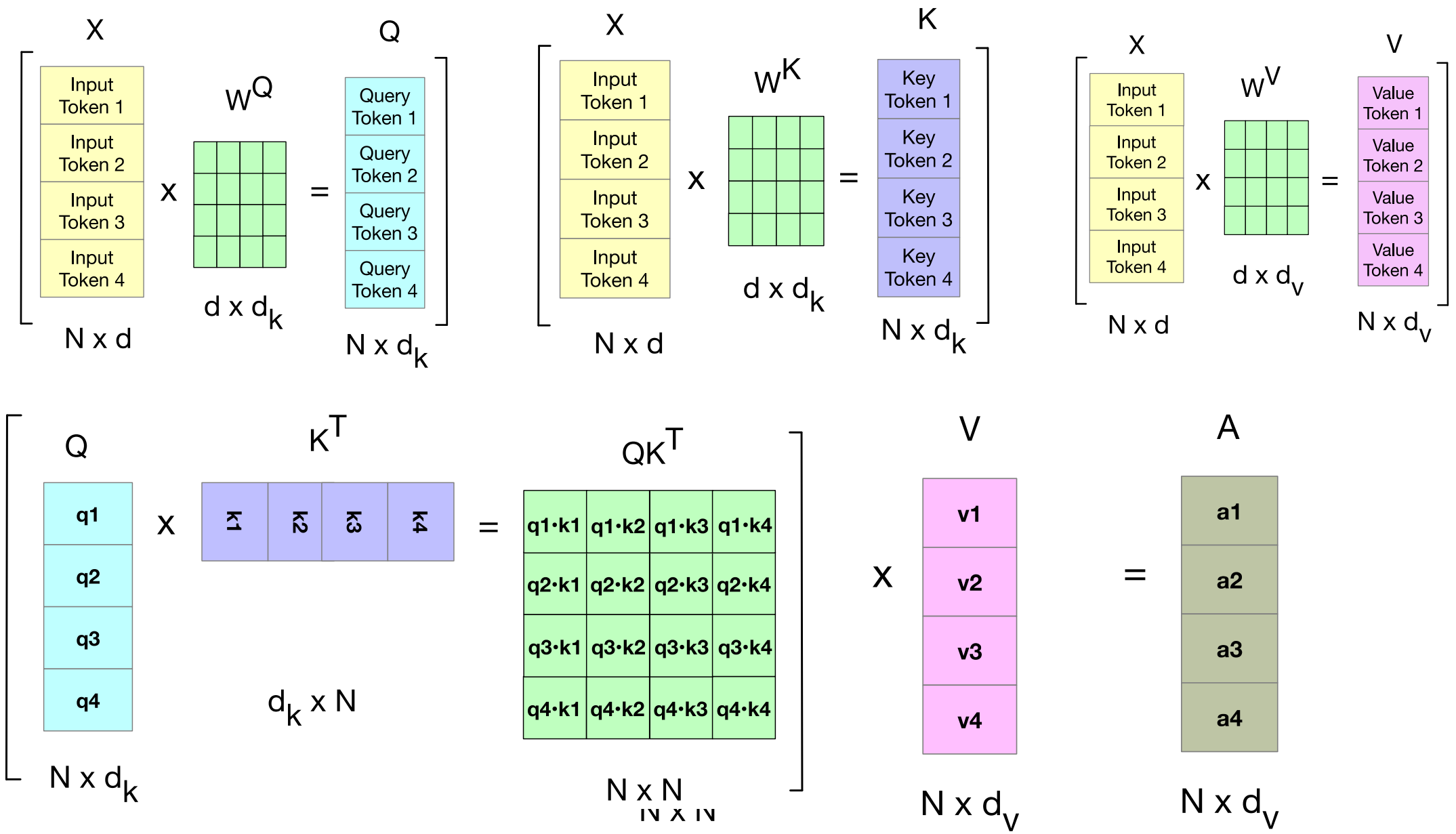
$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j))$$

Step 4: Multiply by Values

- Each word's final representation is obtained by multiplying the **attention weights** with **V**:

$$\mathbf{a}_i = \sum_{j \le i} \alpha_{ij}\mathbf{v}_j$$

Schematic of the attention computation for a single attention head in parallel



Multi-head Attention

- Instead of using **one attention**, Transformers use **multiple attention heads** that learn different aspects of the text.

$$
\begin{aligned}
\mathbf{q}_i^c = \mathbf{x}_i\mathbf{W}^{\mathbf{Qc}};\quad \mathbf{k}_j^c = \mathbf{x}_j\mathbf{W}^{\mathbf{Kc}};\quad \mathbf{v}_j^c &= \mathbf{x}_j\mathbf{W}^{\mathbf{Vc}};\quad \forall\, c \quad 1 \le c \le h \\
\text{score}^c(\mathbf{x}_i, \mathbf{x}_j) &= \frac{\mathbf{q}_i^c \cdot \mathbf{k}_j^c}{\sqrt{d_k}} \\
\alpha_{ij}^c &= \text{softmax}(\text{score}^c(\mathbf{x}_i, \mathbf{x}_j)) \;\; \forall j \le i \\
\mathbf{head}_i^c &= \sum_{j \le i} \alpha_{ij}^c \mathbf{v}_j^c \\
\mathbf{a}_i &= (\mathbf{head}^1 \oplus \mathbf{head}^2 \ldots \oplus \mathbf{head}^h)\mathbf{W}^O \\
\text{MultiHeadAttention}(\mathbf{x}_i, [\mathbf{x}_1, \cdots, \mathbf{x}_N]) &= \mathbf{a}_i
\end{aligned}
$$

feedforward layer & layer normalization

- The feedforward layer is a fully-connected 2-layer network, i.e., one hidden layer, two weight matrices. The weights are the same for each token position i

$$\text{FFN}(\mathbf{x}_i) = \text{ReLU}(\mathbf{x}_i\mathbf{W}_1 + b_1)\mathbf{W}_2 + b_2$$

- At two stages in the transformer block we normalize the vector that can be used to improve training performance in deep neural networks by keeping the values of a hidden layer in a range that facilitates gradient-based training.

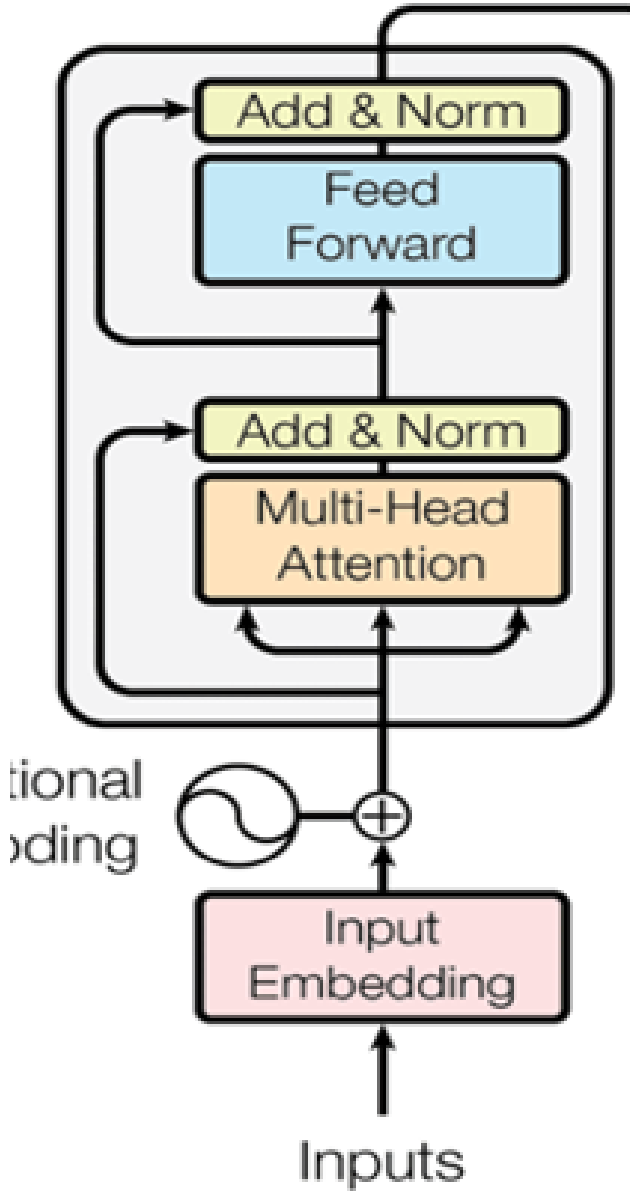
$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$$

$$\sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2}$$

$$\text{LayerNorm}(\mathbf{x}) = \gamma\,\frac{(\mathbf{x} - \mu)}{\sigma} + \beta$$

Putting all together

$$
\begin{aligned}
\mathbf{T}^1 &= \text{MultiHeadAttention}(\mathbf{X}) \\
\mathbf{T}^2 &= \mathbf{X} + \mathbf{T}^1 \\
\mathbf{T}^3 &= \text{LayerNorm}(\mathbf{T}^2) \\
\mathbf{T}^4 &= \text{FFN}(\mathbf{T}^3) \\
\mathbf{T}^5 &= \mathbf{T}^4 + \mathbf{T}^3 \\
\mathbf{H} &= \text{LayerNorm}(\mathbf{T}^5)
\end{aligned}
$$



decoder

- The decoder is used to produce the output, i.e., to generate the sentence in the target language, capturing the same meaning as the sentence in the source language.
- Therefore, the decoder takes the context representation from the encoder as one of the inputs.

Component	Function
Input Embedding	Converts tokens to dense vectors
Positional Encoding	Adds sequence order information
Masked Multi-Head Self-Attention	Allows tokens to attend to past tokens only
Cross Attention(Encoder-Decoder Attention)	Helps the decoder focus on relevant encoder outputs
Feed-Forward Network (FFN)	Applies non-linearity for better representation
Add & Norm (Residual Connections + Layer Norm)	Stabilizes training and prevents gradient vanishing
Final Linear & Softmax Layer	Generates the final output token probabilities

Masked Multi-Head Attention

- The output at a certain position can only depend on the words on the previous positions.
- Masked Multi-Head Attention allows tokens to attend to past tokens and itself.
- A mask is applied to prevent attending to future tokens.
- The mask is an upper triangular matrix with **negative infinity (-∞) values** in places where attention should be blocked
- This forces the attention mechanism to only consider past and current positions.

$$\mathbf{A} = \text{softmax}\left(\text{mask}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\right)\mathbf{V}$$

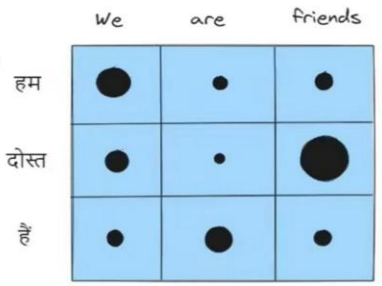
N

q1·k1	−∞	−∞	−∞
q2·k1	q2·k2	−∞	−∞
q3·k1	q3·k2	q3·k3	−∞
q4·k1	q4·k2	q4·k3	q4·k4

N

Cross-Attention

- Cross attention allows the decoder **to attend to the encoder's output**.
- It Helps the decoder to focus on relevant encoder outputs
- The cross-attention in the decoder uses the partially generated sequence as the query and the context representation from the encoder as the key and value.
- For example, if we are translating "We are friends" into Hindi as "हम दोस्त हैं,".
- Cross-attention generates query vectors from the output sequence (Hindi), while key and value vectors are derived from the input sequence (English).
- cross-attention calculates similarity/attention scores between the query vectors from the output sequence and the key vectors from the input sequence.
- These scores are then used to weight the value vectors from the input sequence.
- In the end, this process helps the model determine how similar or related words from the output sequence (Hindi) are to words from the input sequence (English).



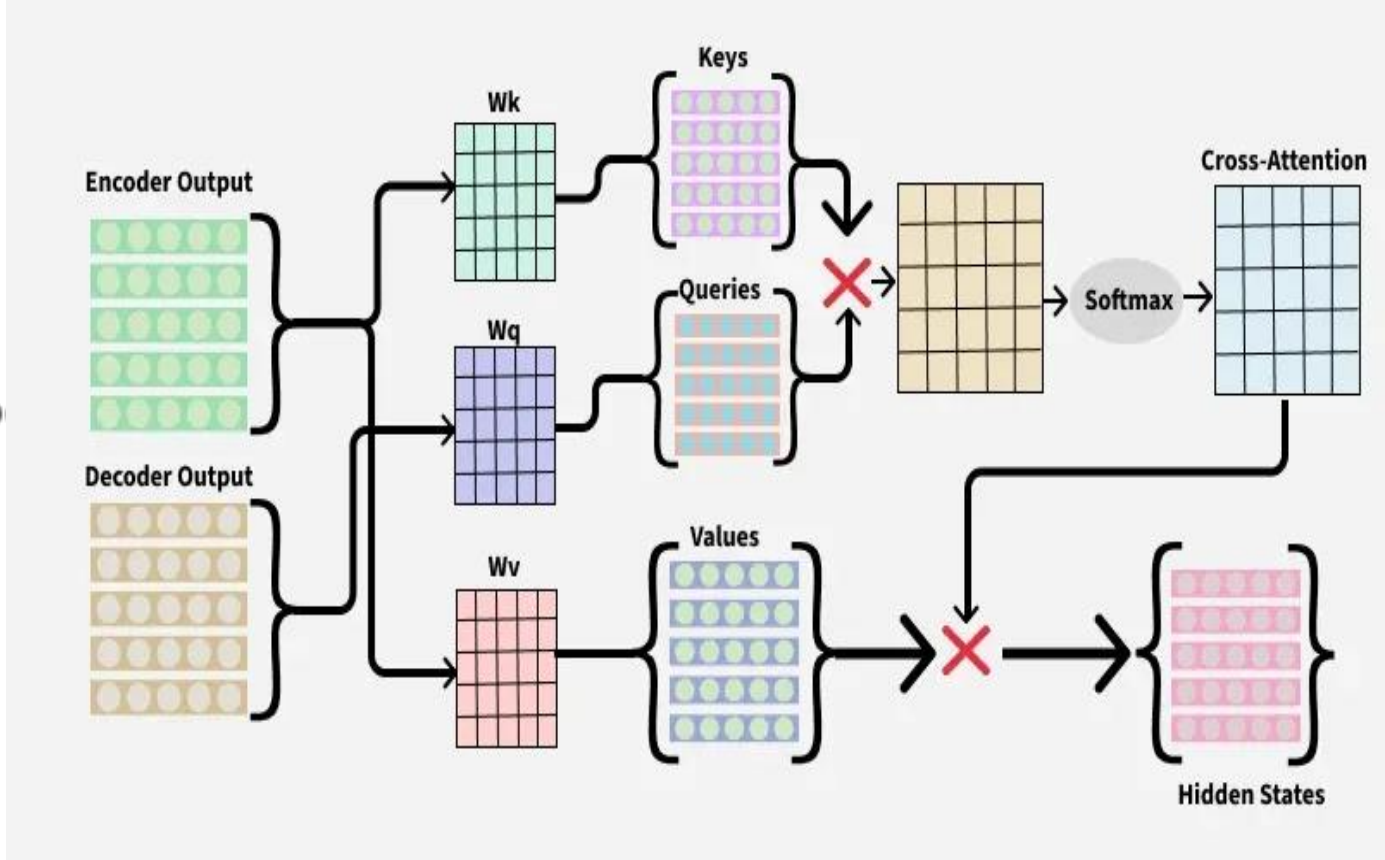
How Cross-Attention Works

- The encoder's output serves as the **Key (K) and Value (V)**, while the decoder's hidden states are used as the **Query (Q)**.

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

where:

- Q = decoder's hidden states
- K, V = encoder's output
- Softmax ensures focus on the most relevant enco



Final Feedforward Network & Output Generation

- The final **FFN layer** transforms attention outputs.
- A **softmax layer** converts the decoder output into probabilities over the vocabulary.
- The token with the **highest probability** is selected as the next word.
- The decoder continues generating tokens **iteratively**.
- Process stops when an **end token (<EOS>)** is generated.

Subword Tokenization

- Subword tokenization is a method used in **NLP** to break words into smaller meaningful **subword units** instead of treating entire words as tokens.
- **Subword tokenization** is a method of breaking text into smaller units than words, but larger than characters — called *subwords*.
- It's commonly used in modern NLP models (like BERT, GPT, and T5) to handle rare and out-of-vocabulary (OOV) words effectively.

- Text: "I love programming!"

Tokens: ["I", "love", "pro", "gram", "ming", "!"]

Subword Tokenization Using Byte Pair Encoding (BPE)

- BPE is a **data compression technique** that iteratively merges the most frequent pair of bytes (or characters) in a corpus into a new symbol.
- It was first introduced for text compression, but it's now widely used for **subword tokenization** in NLP, especially in neural machine translation (NMT).
- It helps address the problem of **out-of-vocabulary (OOV) words** by breaking down words into smaller, more manageable subword units, which can be learned from data.
- **BPE** is a subword tokenization algorithm that:
  - Starts with characters.
  - Iteratively merges the most frequent pair of symbols (subwords).
  - Builds a vocabulary of subwords to better handle unknown or rare words.

- It's great for handling rare words, morphologically rich languages, and reducing [UNK] tokens in NLP models.

Subword Tokenization Using Byte Pair Encoding (BPE)

BPE Algorithm:

1.Initialize Vocabulary

- Start by representing each word in the corpus as a sequence of characters, (e.g., hello → h e l l o).
- This turns the text into a vocabulary of character-level tokens.

2. Count Pair Frequencies

- Count the frequency of each **adjacent character pair** (bigram) in the vocabulary.

3. Merge the Most Frequent Pair

- Find the most frequent pair (e.g., l l) and merge it into a new symbol (ll).Replace all instances of this pair in the vocabulary with the new symbol.

4. Update Vocabulary

- Update the vocabulary with the newly merged symbols.

5. Repeat

- Repeat steps 2–4 for a predefined number of **merge operations** (or until no more pairs can be merged)

Example

Corpus:["low", "lower", "newest"]

Step 1: Initialize Vocabulary

```
l o w          → 1
l o w e r      → 1
n e w e s t    → 1
```

Step 2: Count Symbol Pair Frequencies

Pair	Count
(l, o)	2
(o, w)	2
(w, e)	1
(e, r)	1
(n, e)	1
(e, w)	1
(w, e)	1
(e, s)	1
(s, t)	1

Subword Tokenization Using Byte Pair Encoding (BPE)

Step 3: Merge Most Frequent Pair → (l, o) → lo

lo w
lo w e r
n e w e s t

Step 4: Merge (lo, w) → low

low
low e r
n e w e s t

Step 5: Merge (e, r) → er

low
low er
n e w e s t

Step 6: Merge (e, s) → es

low
low er
n e w es t

Step 7: Merge (es, t) → est

low
low er
n e w est

Final Vocabulary (subwords learned):

```
l
o
w
e
r
n
s
t
lo      # from l + o
low     # from lo + w
er      # from e + r
es      # from e + s
est     # from es + t
```

Final Tokenized Words:

"low" → low“

lower" → low, er“

newest" → n, e, w, est