Natural Language Processing
Course code: CSE3015

Module 4
NLP Using Deep Learning

Prepared by

Dr. Venkata Rami Reddy Ch

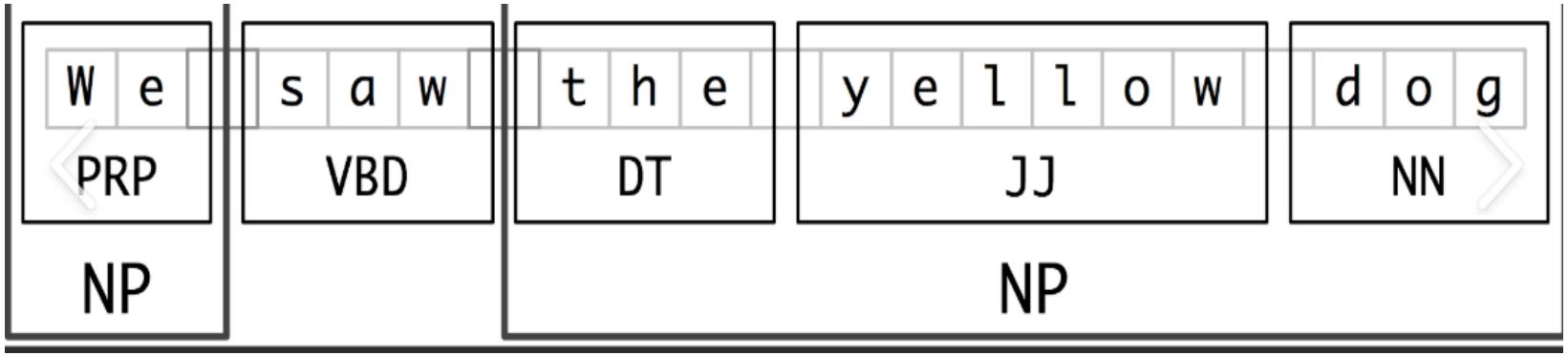
SCOPE

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.

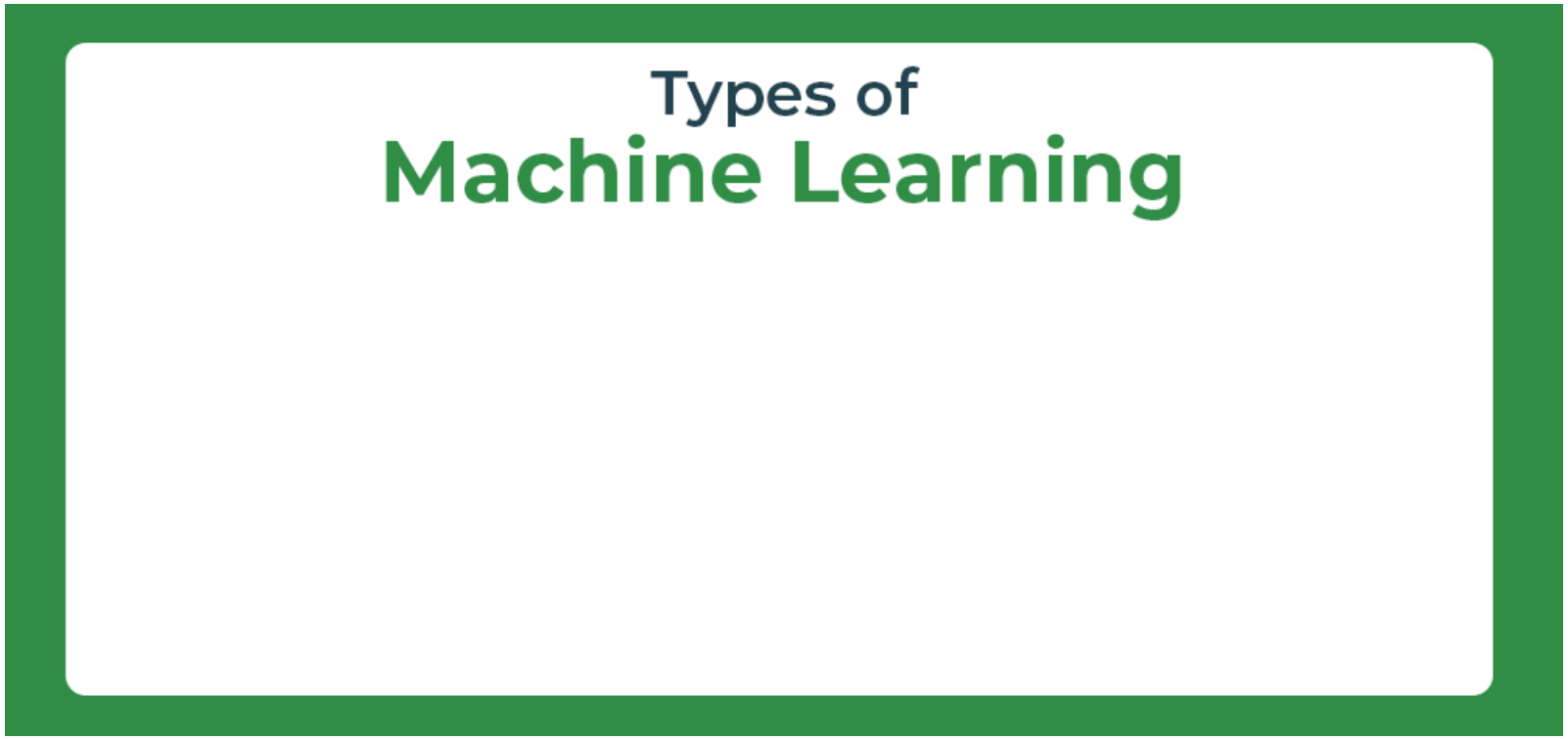
Chunking in NLP

```
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

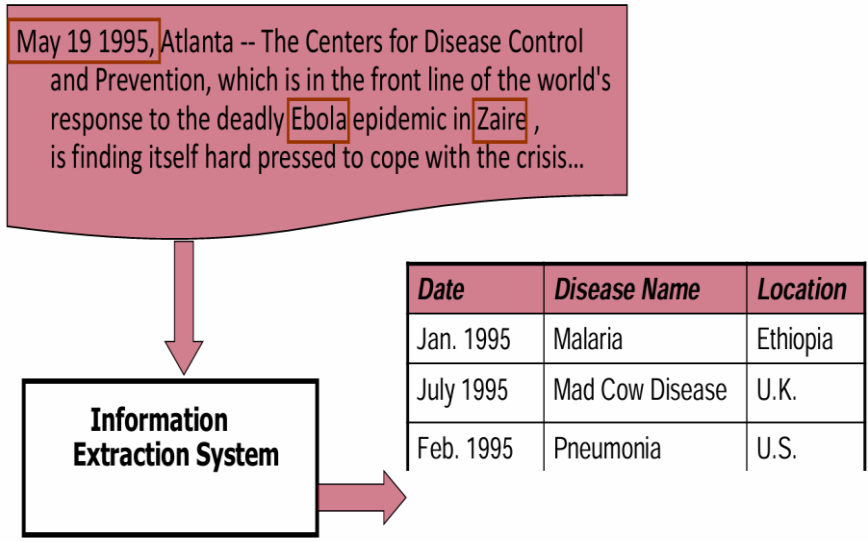
Entities(NER)

- 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']}

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

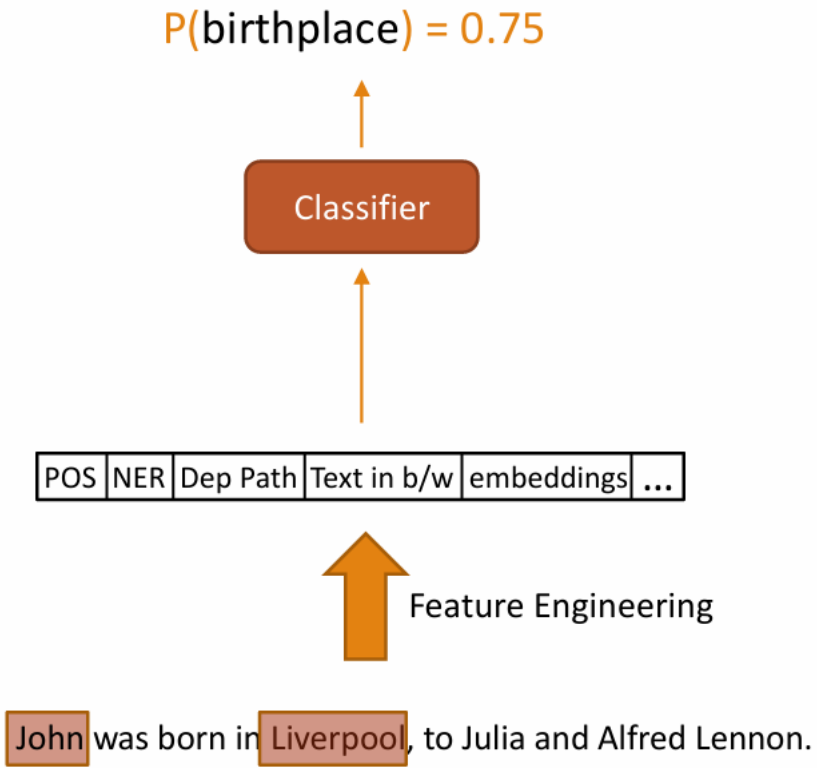
- Choose a supervised learning model:
  - **Traditional ML models**: SVM, Decision Trees, Random Forest, Logistic Regression.
  - **Deep Learning models**: CNN, BiLSTM, Transformer-based models (BERT, RoBERTa).
- Train the model on labelled data to learn patterns.

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

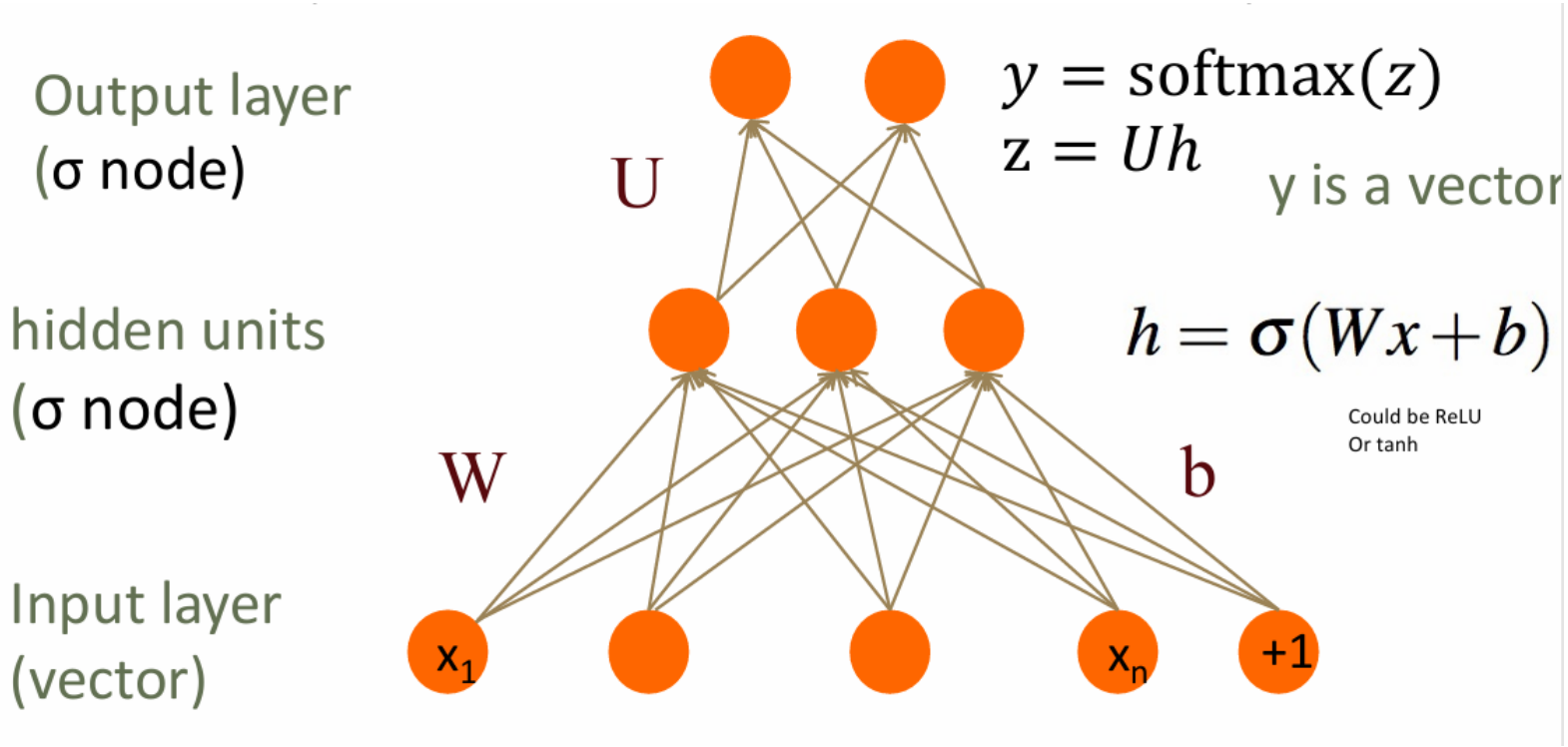


```
import nltk
import spacy
import numpy as np
from sklearn.svm import SVC
from sklearn.feature_extraction import DictVectorizer
from sklearn.pipeline import make_pipeline

# 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"))
```

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.

Cont..

- 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.

Recurrent neural networks

Key Features of RNNs:

Sequential Processing – Unlike traditional neural networks, RNNs maintain a hidden state(memory) that captures information from previous time steps.

Weight Sharing – The same set of weights is applied at each time step, reducing the number of parameters.

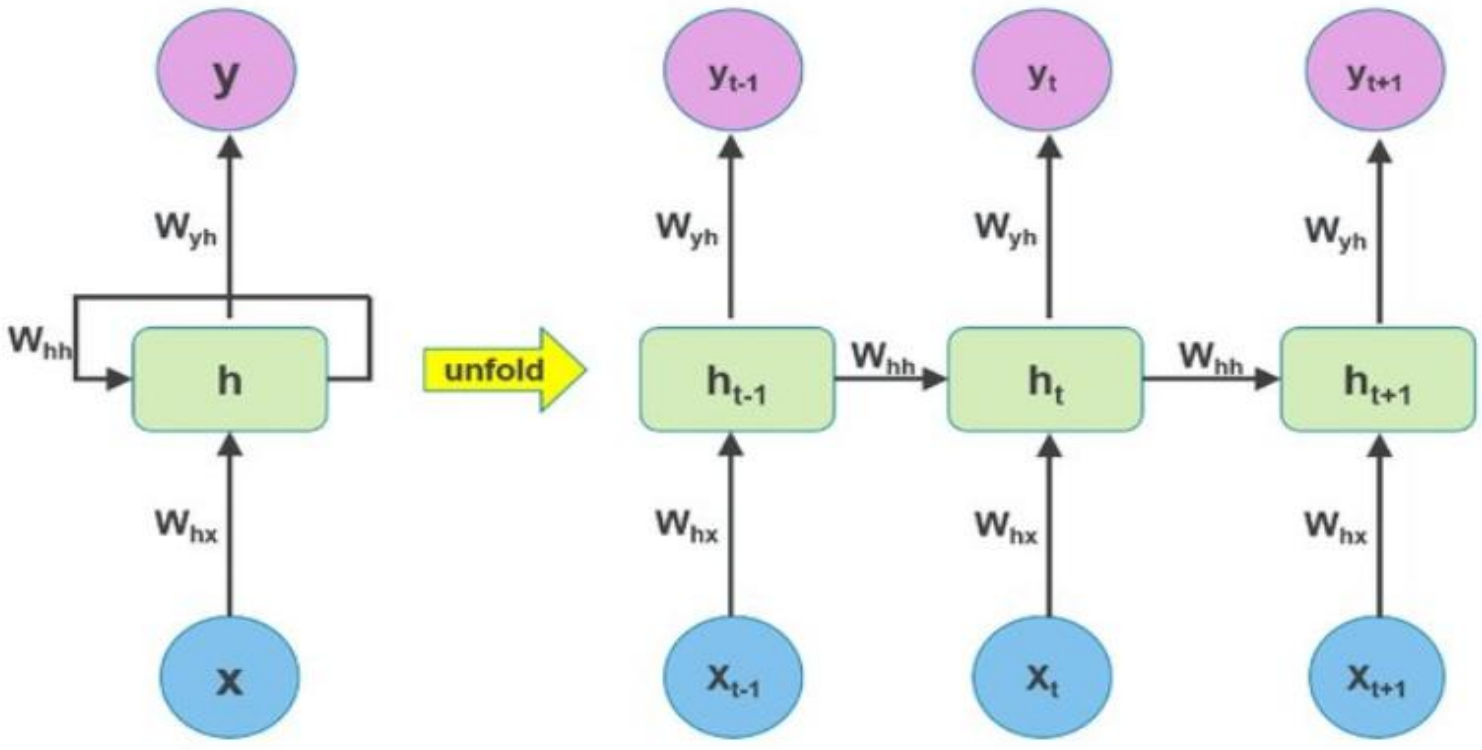
Backpropagation Through Time (BPTT) – A variant of backpropagation used to update weights in RNNs.

Temporal Dependencies – RNNs capture relationships over time, enabling them to recognize patterns in time-series data and text.

The Structure of RNN

A basic RNN consists of three layers:

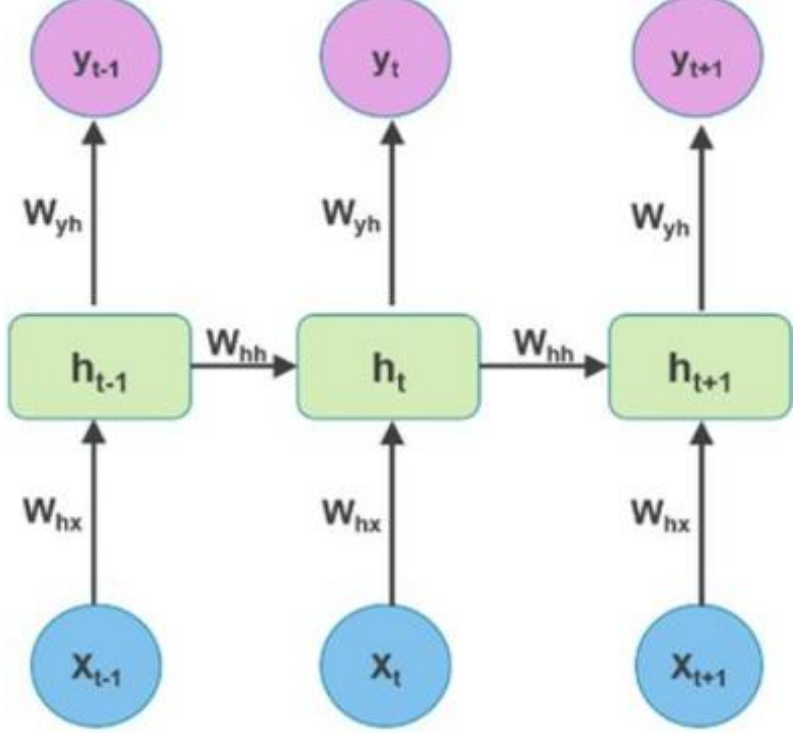
1. **Input Layer** – Takes in the current input (e.g., a word, a time-series value).
2. **Hidden Layer (Memory Unit)** – Stores past information and updates itself at each time step.
3. **Output Layer** – Produces the final prediction based on the hidden state.



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 processes input data in a sequence, maintaining a hidden state that gets updated at each 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 will be given as:

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**).

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

After obtaining outputs y_t at each time step, the loss function $\mathcal{L}$ is calculated over the entire sequence:

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

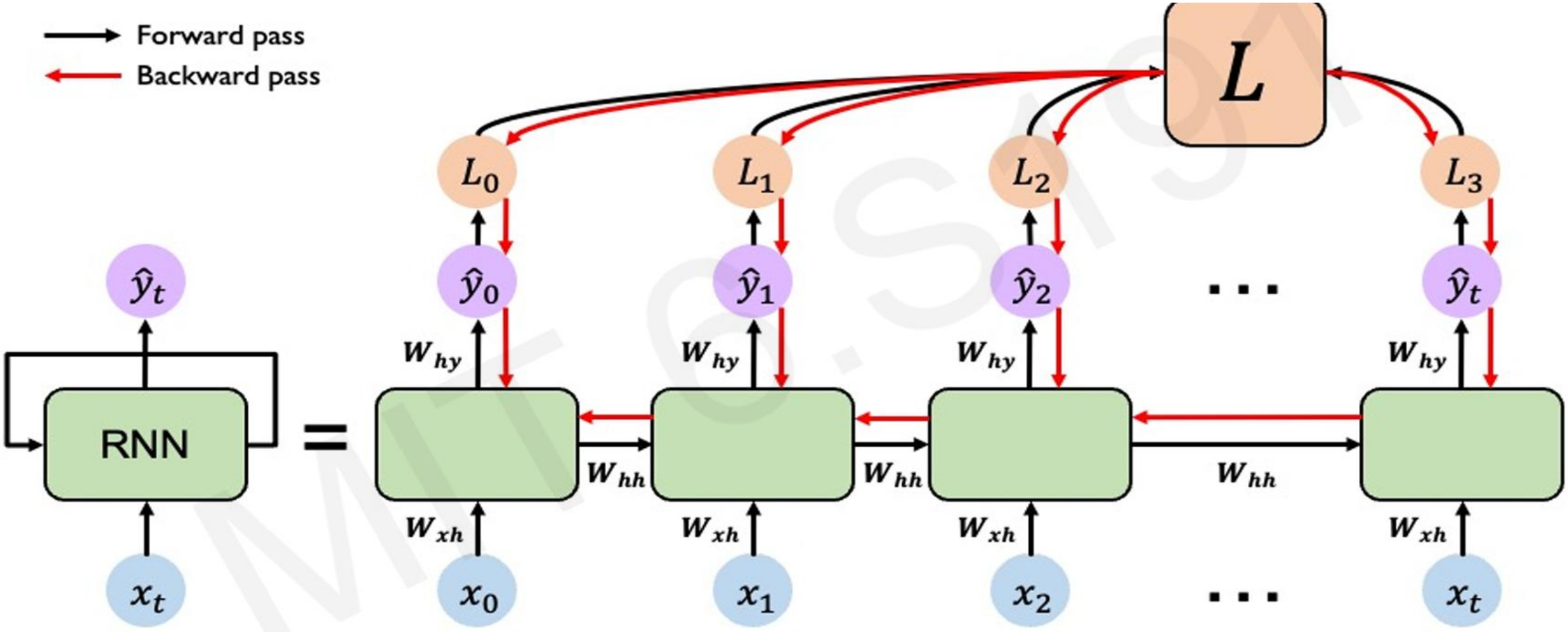
where:

- $\hat{y}_t$ is the true output (ground truth).
- y_t is the predicted output.
- $\mathcal{L}_t$ is typically **cross-entropy loss** for classification.

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):

- Since the hidden states are dependent on previous time steps, we compute gradients **recursively** using the chain rule.

1. **Compute gradient of the loss with respect to output weights W_{yh}:**

$$\frac{\partial \mathcal{L}}{\partial W_{yh}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial W_{yh}}$$

2. **Compute gradient of the loss with respect to hidden state h_t:**

$$\delta h_t = \frac{\partial \mathcal{L}_t}{\partial h_t} + \frac{\partial \mathcal{L}_{t+1}}{\partial h_{t+1}} \cdot \frac{\partial h_{t+1}}{\partial h_t}$$

3. **Compute gradient of the loss with respect to recurrent weights W_{hh}:**

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_{t=1}^{T} \delta h_t \cdot h_{t-1}^{T}$$

4. **Compute gradient of the loss with respect to input weights W_{hx}:**

$$\frac{\partial \mathcal{L}}{\partial W_{hx}} = \sum_{t=1}^{T} \delta h_t \cdot x_t^{T}$$

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

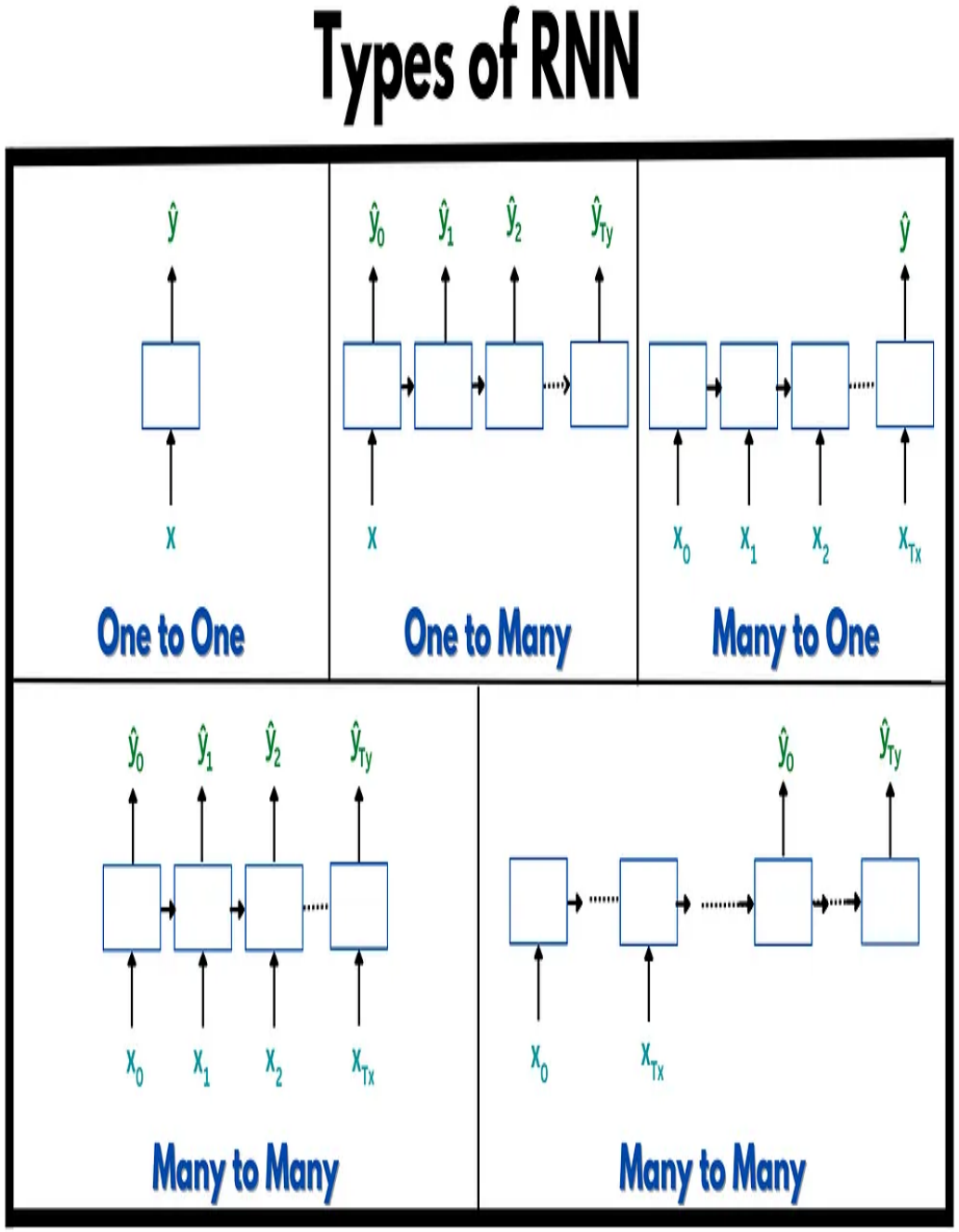
Recurrent Neural Networks (RNNs) have different architectures based on how inputs and outputs are structured.

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

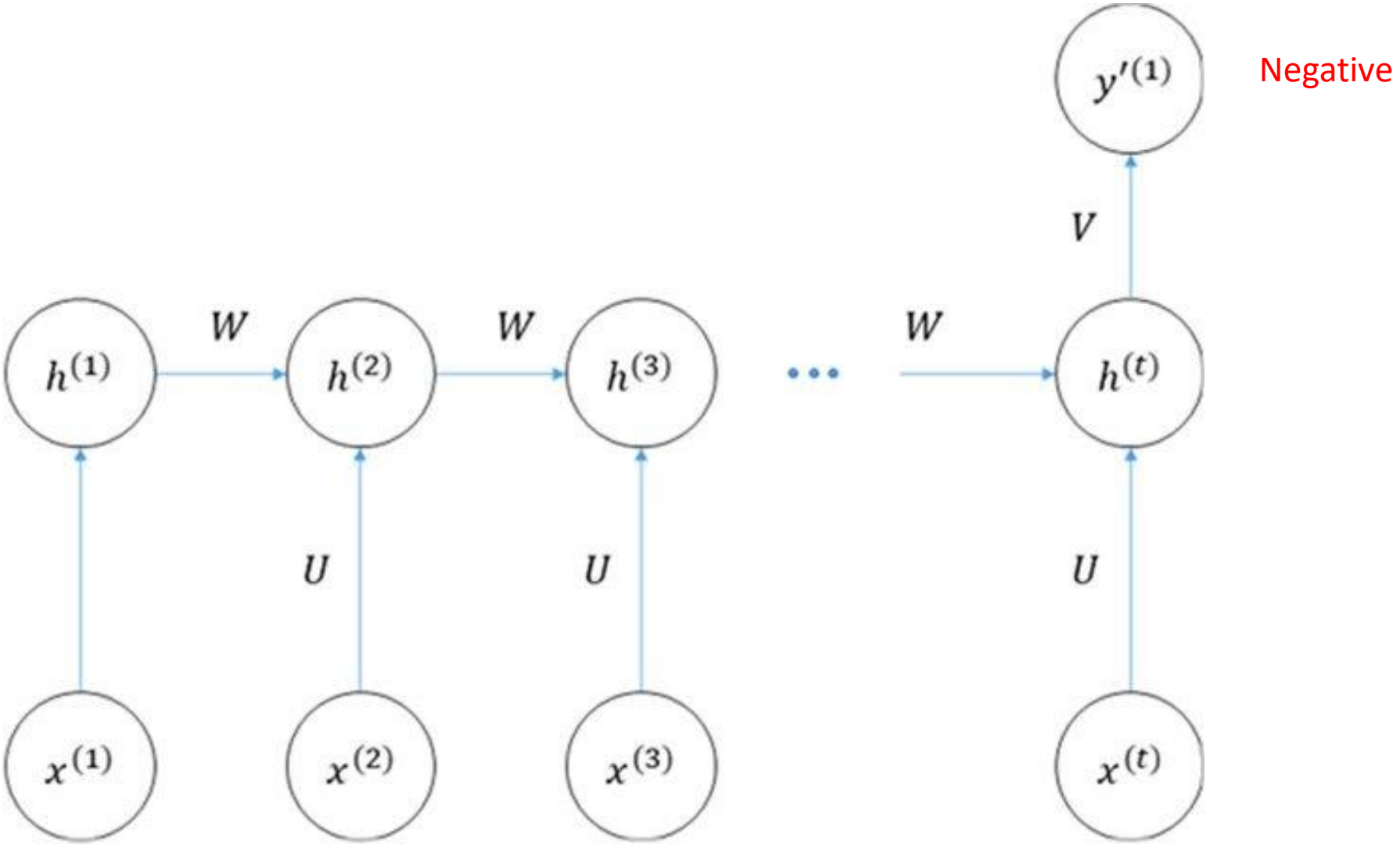
Many-to-Many (Same Length)

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



Vanishing and Exploding Gradients problem in RNNs

- The **vanishing and exploding gradient problem** occurs during backpropagation in RNNs due to repeated multiplication of gradients through multiple time steps. This makes it difficult to train deep networks effectively.

Vanishing Gradient Problem

Why It Happens

•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.

Effects

•The network struggles to learn long-range dependencies.

•The model relies mostly on recent inputs and ignores older ones.

•Training becomes slow, and weights stop updating.

Vanishing and Exploding Gradients problem in RNNs

Exploding Gradient Problem

Why It Happens

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

Effects

- Model parameters update too aggressively.
- Loss does not converge properly.
- The model may produce meaningless or repetitive outputs.

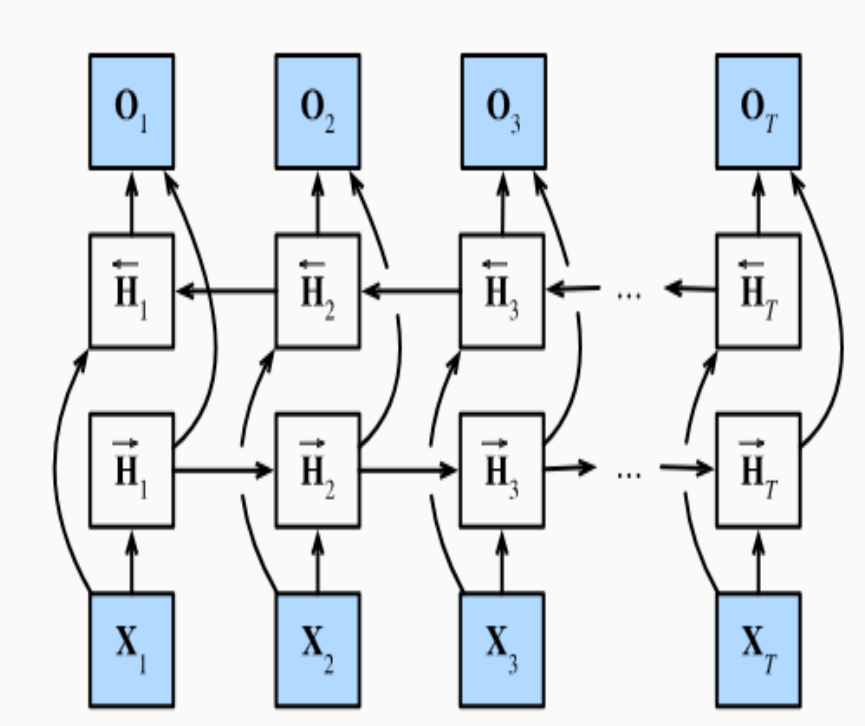
Bidirectional Recurrent Neural Networks

- A **Bidirectional Recurrent Neural Network (BiRNN)** is an extension of a standard RNN that processes data in both **forward** and **backward** directions.
- This allows the network to have **context from both past and future** time steps, making it especially useful for NLP tasks like Named Entity Recognition (NER), POS tagging, and Machine Translation.

How BiRNN Works

A BiRNN consists of two RNN layers:

- **Forward RNN** → Processes the sequence from **start to end**.
- **Backward RNN** → Processes the sequence from **end to start**.
- The outputs from both RNNs are concatenated or combined



Bidirectional Recurrent Neural Networks

Mathematical Representation
For a given input sequence X=(x1,x2,…,xT):

Forward RNN:

$$\overrightarrow{h}^{<t>} = g(W_x X^{<t>} + W_h \overrightarrow{h}^{<t-1>} + b_h)$$
$$\overrightarrow{y}^{<t>} = W_y \overrightarrow{h}^{<t>} + b_y$$

Where:
•X<t>: is the input at time step t
•W_x: is the weight matrix for the input
•W_h: is the weight matrix for the hidden state from the previous time step
•W_y: is the weight matrix from the hidden state to the output
•b_h, b_y: are the bias terms
•g: activation function, typically a non-linear function like tanh or ReLU
•y→<t>: is the output at time step t

Backward RNN:

$$\overleftarrow{h}^{<t>} = \sigma(W_x X^{<t>} + W_h \overleftarrow{h}^{<t+1>} + b_h)$$
$$\overleftarrow{y}^{<t>} = W_y \overleftarrow{h}^{<t>} + b_y$$

Where:
•X<t>: is the input at time step t (same as in the forward RNN)
•W_x, W_h, and b_h are the same matrices and bias terms used in the forward RNN but applied in the reverse order
•h←<t+1>: is the hidden state from the next time step (as we are processing backward)

Bidirectional Recurrent Neural Networks

BPTT

- In the case of a bidirectional RNN, BPTT involves two separate Backpropagation passes: one for the forward RNN and one for the backward RNN.
- During the forward pass, the forward RNN processes the input sequence in the usual way and makes predictions for the output sequence.
- These predictions are then compared to the target output sequence, and the error is backpropagated through the network to update the weights of the forward RNN.

- The backward RNN processes the input sequence in reverse order during the backward pass and predicts the output sequence.
- These predictions are then compared to the target output sequence in reverse order, and the error is backpropagated through the network to update the weights of the backward RNN.
- Once both passes are complete, the weights of the forward and backward RNNs are updated based on the errors computed during the forward and backward passes, respectively.

Bidirectional Recurrent Neural Networks

Combined Output:

There are several ways in which the outputs of the forward and backward RNNs can be merged, depending on the specific needs of the model and the task it is being used for.
Some common merge modes include:

Concatenation:

- In this mode, the outputs of the forward and backward RNNs are concatenated together, resulting in a single output tensor that is twice as long as the original input.

Sum:

- In this mode, the outputs of the forward and backward RNNs are added together element-wise, resulting in a single output tensor that has the same shape as the original input.

Average:

- In this mode, the outputs of the forward and backward RNNs are averaged element-wise, resulting in a single output tensor that has the same shape as the original input.

Maximum:

- In this mode, the maximum value of the forward and backward outputs is taken at each time step, resulting in a single output tensor with the same shape as the original input.