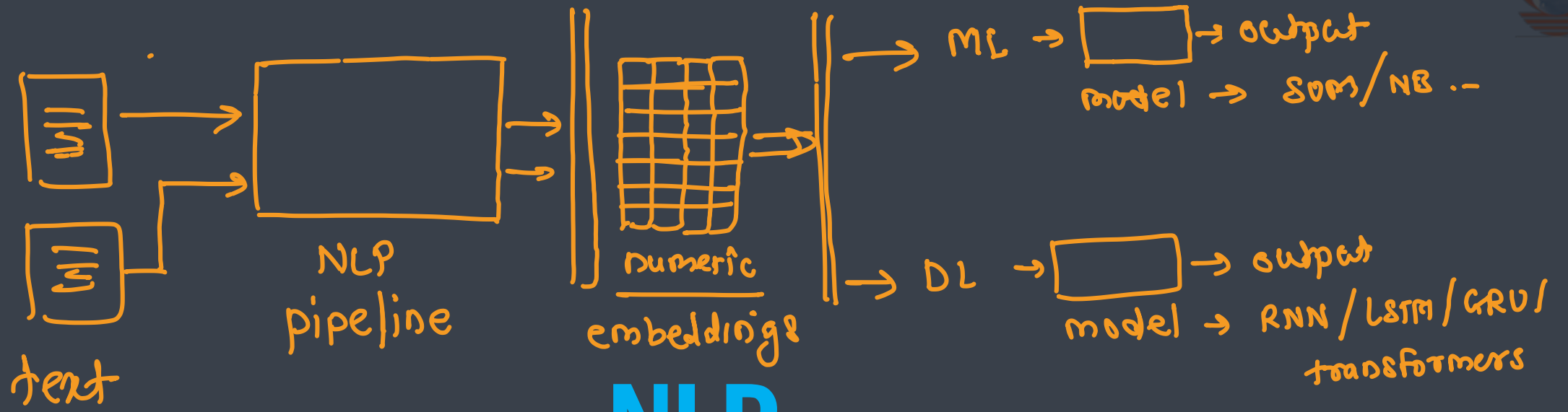
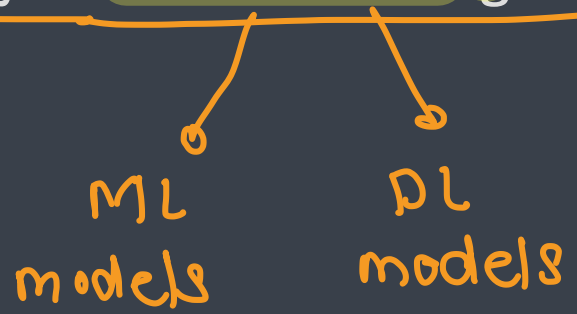

text
NLP
pipeline
numeric
embeddings
ML
model
output
SVM/NB ...
DL
model
output
RNN / LSTM / GRU /
transformers

NLP

What is NLP?

- Natural Language Processing (NLP) is a field of artificial intelligence (AI) that focuses on the interaction between computers and humans through natural language. → English - other language
- Its goal is to enable computers to understand, interpret, and respond to human language in a way that is both meaningful and useful. → context → output → sentiment, QA etc.
- The ultimate objective of NLP is to read, decipher, understand, and make sense of the human languages in a manner that is valuable → machine can understand
- Most NLP techniques rely on machine learning to derive meaning from human languages



Uses cased of NLP

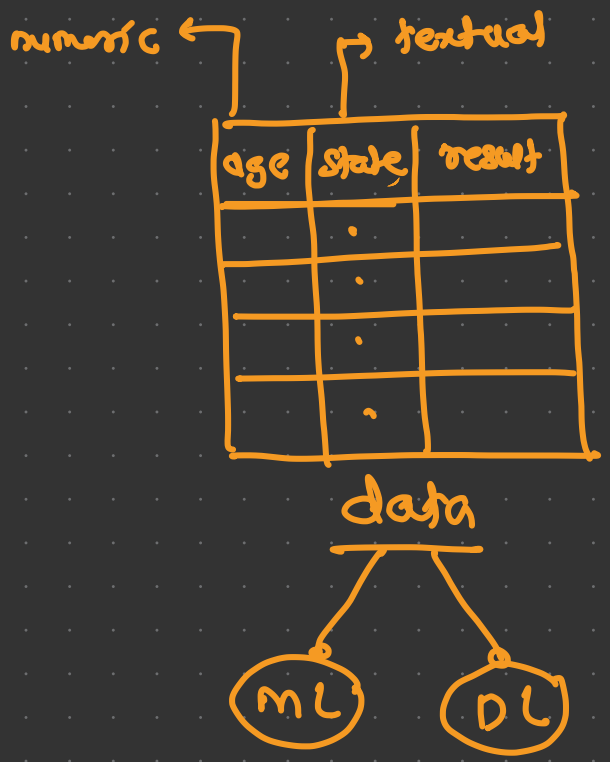
→ sentiment analysis – [+ve, -ve, neutral]

- **Text Analysis:** Understanding and extracting information from text data, including sentiment analysis, topic modeling, and keyword extraction.
- **Machine Translation:** Automatically translating text from one language to another (e.g., Google Translate) → DL model → transformers → LSTM / GPU
- **Speech Recognition:** Converting spoken language into written text, enabling applications like virtual assistants (e.g., Siri, Alexa)
- **Chatbots and Conversational Agents:** Designing systems that can engage in dialogue with users, providing responses to queries and performing tasks → similar to humans
- **Information Retrieval:** Enhancing search engines to retrieve relevant information based on user queries
- **Text Generation:** Creating coherent and contextually relevant text, as seen in applications like summarization and creative writing → Gen AI → GAN
- **Named Entity Recognition (NER):** Identifying and classifying key entities (like people, organizations, and locations) within a text → basis of Event Extraction

p summarization, event Extraction

Libraries for NLP

- Natural Language Toolkit (NLTK)
  - NLTK is one of the top frameworks for creating Python applications that can operate on human language data
  - Sentence identification, tokenization, lemmatization, stemming, parsing, chunking, and POS tagging are just a few of the text processing functions that it has
  - Over 50 corpora and lexical resources can be accessed through NLTK's user-friendly interfaces
- spaCy
  - Python's spaCy is an open-source NLP package
  - It allows you to create applications that process massive amounts of text because it is specifically intended for use in production environments
  - It can be used to build information extraction or natural language processing systems
  - It has word vectors and pre-trained statistical models, and can accommodate more than 49 languages for tokenization
- TextBlob
  - TextBlob provides very convenient APIs for standard NLP tasks, including POS tagging, noun phrase extraction, sentiment analysis, classification, language translation, word inflection, parsing, n-grams, and WordNet integration
  - The objects it creates can be thought of as Python strings with NLP training



Non-NLP data

corpus → data used in NLP, a huge text

↳ documents → paragraphs/sentences of huge text

```
data = """ Welcome to NLP.
NLP is the most imp task in ML.
"""
```

textual

NLP data

NLP Pipeline : collection of stages executed sequentially

(unstructured) text → pipeline → structured data / numeric data

1. **Text Preprocessing** — data cleansing ✓ — token = word — sentence tokenization → word tokenization
   - **Tokenization:** Splitting text into individual words or phrases (tokens).
   - **Lowercasing:** Converting all text to lowercase to maintain consistency.
   - **Removing Punctuation and Special Characters:** Cleaning the text to remove any non-essential elements.
   - **Stop Word Removal:** Eliminating common words (like "the," "is," "and") that may not contribute significant meaning
   - **Lemmatization:** Converting the words to their base forms → stemming
   - **NER:** Finding named entity representation → person name, organization name, date

   Apple, Apples → apple (base form)

   data = ["...", "...", "..."...]

2. **Text Representation** → conversion of words into numeric format.
   - **Word Embeddings:** Converting tokens into numerical representations (e.g., using Word2Vec, GloVe, or BERT) that capture semantic meanings → context
   - **Bag of Words (BoW):** Representing text as a vector of word counts → frequently appearing → CountVectorizer
   - **Term Frequency-Inverse Document Frequency (TF-IDF):** Weighting the importance of words based on their frequency in a document relative to their frequency across a collection of documents

NLP Pipeline

Selection of features
Extraction of features

3. Feature Engineering

- Creating additional features from the text that might help in modeling, such as:
- N-grams (combinations of n words)
- Part-of-speech tags (Pos tag) → verb, noun, pro-noun etc.
- Named entities - (NER)

noun
apples - apple

playing - play
verb

4. Modeling

ML
DL

- Training a Machine Learning Model: Using labeled data to train models for tasks like classification, regression, or clustering → grouping similar words
- Using Pre-trained Models: Leveraging existing models (like BERT or GPT) for specific tasks, often fine-tuning them on domain-specific data → transfer learning → langChain → Ollama

5. Post-Processing

- Interpreting Model Outputs: Converting raw predictions into actionable insights (e.g., converting sentiment scores into categories). → building newer model
- Visualizing Results: Creating visual representations of the data and findings. → WordCloud

NLP Pipeline

6. **Evaluation** → classification
   - **Metrics:** Assessing model performance using metrics such as accuracy, precision, recall, F1-score, etc.
   - **Cross-Validation:** Ensuring the model's robustness by testing it on different subsets of data.

7. **Deployment** → deployment of application in production env. [making it accessible to end user]
   - Integrating the model into applications or systems to make it accessible for real-world use, such as chatbots or recommendation systems → cloud, native app, mobile app

8. **Monitoring and Maintenance**
   - Continuously monitoring the model's performance and updating it as needed based on new data or changing requirements

Text Preprocessing

Conversion of corpus (documents) into tokens

Text Preprocessing

corpus → document ⇒ corpus = ["...", "...", "..."]

raw

Text Document

document → sentences

sentences → words

noun, verb, adjective

brings word to base form

- Sentence Segmentation
- Tokenization
- Parts of Speech Tagging
- Lemmatization
  - stemming

unwanted words / noise

dependency graph

person name, org name etc.

- Stop Words
- Dependency Parsing
- Noun Phrases
- Named Entity Relationship

Data Structure representing parsed text

structured

Sentence Segmentation

- Sentence tokenization is the process of splitting a piece of text into individual sentences
- This is an important step in natural language processing (NLP) and text analysis, as it allows for a more structured understanding of text by isolating distinct units of meaning → sentences
- The process typically involves identifying sentence boundaries, which can be defined by punctuation marks like periods, exclamation points, and question marks
- For example, the text "Hello! How are you? I hope you are well." would be tokenized into the sentences: ["Hello!", "How are you?", "I hope you are well."]
- **Techniques**
  - **Rule-Based Approaches:** Simple methods that rely on predefined rules (e.g., looking for punctuation).
  - **Machine Learning:** More advanced methods that use trained models to predict sentence boundaries based on context and patterns in the text. meaning
  - **Hybrid Approaches:** Combining both rule-based and machine learning techniques for better accuracy.

Word Tokenization

- Word tokenization is the process of breaking down a text into individual words, or token
- This is a crucial step in natural language processing (NLP) and text analysis, as it allows for the handling of text data in a structured way
- The process typically involves splitting a string of text based on whitespace and punctuation
- For example, the sentence "Hello, world!" would be tokenized into the tokens: ["Hello", "world"]
- **Types of Tokenization**
  - **Word Tokenization:** Splitting text into individual words. It often considers punctuation and may handle contractions (e.g., "don't" → "do", "n't").
  - **Subword Tokenization:** Dividing words into smaller units or subwords, which can help handle rare or complex words. Techniques like Byte Pair Encoding (BPE) are commonly used.
  - **Character Tokenization:** Breaking text down to individual characters, which can be useful in specific applications like language modelling.

Predicting Parts of Speech for Each Token

lexical analysis → checking syntax

sl = "the boy goes to school" ⇒ the school goes to boy

- Part-of-Speech (POS) tagging is the process of assigning a part of speech to each word in a sentence, based on its definition and context →
- The parts of speech include categories such as nouns, verbs, adjectives, adverbs, pronouns, prepositions, conjunctions, and interjections
- **Lexical Analysis**: Each word in a sentence is analyzed to determine its most likely part of speech based on its context. → structure
- **Contextual Clues**: The meaning of a word can change depending on its surrounding words, so context plays a crucial role in accurate tagging

- **Tagging Techniques**
  - **Rule-Based Tagging:** Uses hand-crafted rules and dictionaries to assign POS tags. While simple, this approach can be limited by the rules' comprehensiveness. → dictionary lookup
  - **Statistical Tagging:** Involves using probabilistic models, such as Hidden Markov Models (HMMs), which use training data to learn the likelihood of certain tags following others. → probabilities
  - **Machine Learning Approaches:** Modern techniques utilize machine learning algorithms, such as Conditional Random Fields (CRFs) or deep learning models (e.g., LSTMs), to achieve higher accuracy.

Text Lemmatization

- Lemmatization is the process of reducing a word to its base or root form, known as a "lemma."
- Unlike stemming, which simply truncates words to their root form (often resulting in non-words), lemmatization considers the morphological analysis of the words, ensuring that the root form is a valid word in the language
- **Dictionary Lookup:** Lemmatization typically involves looking up words in a dictionary or a morphological analysis database to find their base forms.
- **Part of Speech Tagging:** It often requires knowing the part of speech of the word, as the lemma can differ based on its grammatical role. For example, "better" as an adjective lemmatizes to "good," while as a verb it may stay as "better."
- **Lemmatization vs. Stemming**
  - **Lemmatization:** Produces meaningful base forms and takes into account the context and grammar of the word.
  - **Stemming:** Usually involves simpler, rule-based reductions and may not always result in a valid word (e.g., "running" might stem to "run," but "better" might stem to "better" or "bett").

reviews	type
this res is good	1
this res is bad	0
service is awesome	1
⋮	⋮

BoW

this is a good restaurant

⇓

"this" "is" "a" "good" "rest"

tokens

[good: 5 , bad: 4, ..]

⇓

good:	→	1
bad:	→	0

Identifying Stop Words

- Stop words are common words in a language that are often filtered out or removed during natural language processing (NLP) and text analysis → they do not contribute in getting meaning of sentence
- These words typically include prepositions, conjunctions, articles, and other functional words that do not carry significant meaning by themselves. Examples of stop words in English include:
  - **Articles**: "a," "an," "the"
  - **Prepositions**: "in," "on," "at," "by"
  - **Conjunctions**: "and," "or," "but"
  - **Pronouns**: "he," "she," "it," "they"

Dependency Parsing → dependency tree → semantic nature of text

- Dependency parsing is a process in natural language processing (NLP) that involves analyzing the grammatical structure of a sentence to establish relationships between words
- This technique identifies how words depend on each other within a sentence, creating a tree-like structure that shows the syntactic relationships
- **Approaches** ← traditional
  - **Rule-Based Methods:** Use a set of predefined grammatical rules to identify dependencies.
  - **Statistical Methods:** Utilize probabilistic models trained on annotated corpora to learn and predict dependencies based on patterns in the data.
  - **Neural Networks:** Modern approaches often employ deep learning techniques, such as recurrent neural networks (RNNs) or transformer models, to achieve state-of-the-art results.

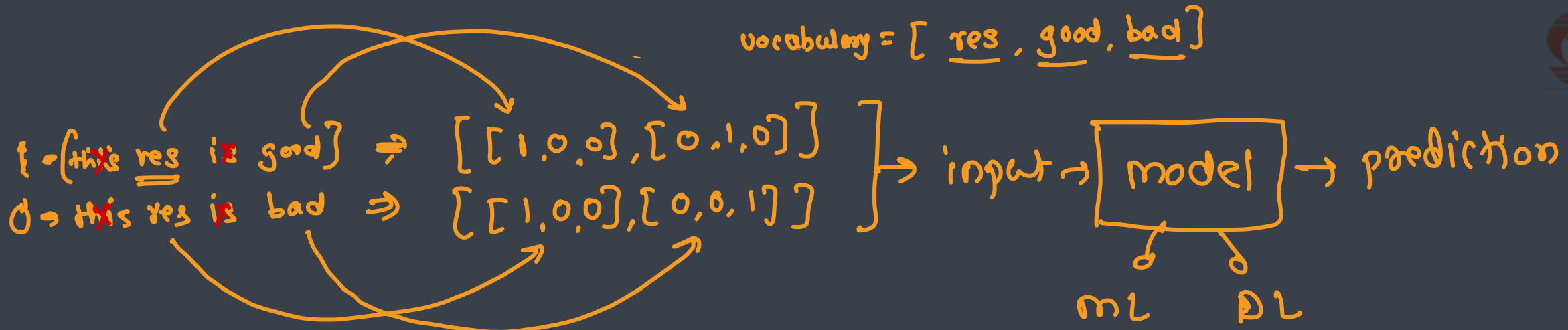
models to perform action

Finding Noun Phrases

- A noun phrase (NP) is a group of words that functions in a sentence as a noun
- It typically includes a noun (or pronoun) and its modifiers, which can be adjectives, determiners, or other phrases that provide additional information about the noun
- Noun phrases can serve various roles in a sentence, such as the subject, object, or complement
- **Structure**
  - **Head Noun:** The main noun in the phrase (e.g., "dog" in "the big dog").
  - **Modifiers:** Words or phrases that provide more detail about the noun, such as adjectives, articles, or prepositional phrases (e.g., "the big" and "in the park" in "the big dog in the park").
- **Examples**
  - Simple NP: "The cat"
  - NP with Adjectives: "The fluffy white cat"
  - NP with a Prepositional Phrase: "The cat on the roof"
  - NP with a Relative Clause: "The cat that I saw yesterday"

Named Entity Recognition (NER)

- Named Entity Recognition (NER) is a subtask of natural language processing (NLP) that involves identifying and classifying named entities in text into predefined categories
- These entities can include names of people, organizations, locations, dates, quantities, monetary values, and other specific identifiers
- **Common Categories**
  - **People:** Names of individuals (e.g., "Albert Einstein")
  - **Organizations:** Names of companies, institutions, or groups (e.g., "Google", "United Nations")
  - **Locations:** Geographical entities (e.g., "New York", "Mount Everest")
  - **Dates:** Specific dates or time expressions (e.g., "January 1, 2023")
  - **Miscellaneous:** Other entities such as products, events, and artworks.
- **Approaches** ↪ dictionary lookup
  - **Rule-Based Methods:** Use handcrafted rules and regular expressions to identify entities based on specific patterns.
  - **Statistical Models:** Employ probabilistic models like Hidden Markov Models (HMMs) or Conditional Random Fields (CRFs) to recognize and classify entities based on context.
  - **Deep Learning:** Utilize neural network architectures, such as BiLSTMs and transformers (e.g., BERT), to achieve state-of-the-art performance in entity recognition.



Text Representation

conversion of textual data to numeric format

Text Representation

- Text representation in natural language processing (NLP) refers to the methods and techniques used to convert textual data into numerical formats that can be processed by machine learning algorithms
- Effective text representation is crucial for various NLP tasks, such as classification, translation, and sentiment analysis
- Here are some of the key methods and concepts involved in text representation
  - **Bag of Words (BoW)**
  - Term Frequency-Inverse Document Frequency (TF-IDF)
  - Word Embeddings
  - Contextualized Word Embeddings
  - **Sentence and Document Embeddings**
  - **Feature Engineering**
  - Dimensionality Reduction

- one Hot Encoding

One-Hot Encoding → binary vectors

- Represents each word as a binary vector with a length equal to the size of the vocabulary
- Each position corresponds to a specific word, with a 1 in the position of the word and 0s elsewhere
- E.g.
  - For example, if your vocabulary is ["cat", "dog", "fish"], the one-hot encodings would be:
  - "cat" → [1, 0, 0]
  - "dog" → [0, 1, 0]
  - "fish" → [0, 0, 1]

I love cats → [I, cats, dogs, love, white]

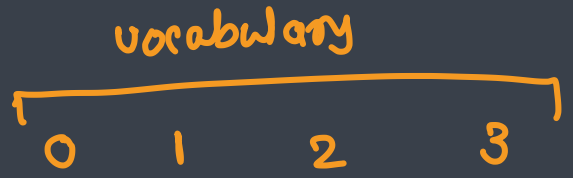
[[1 0 0 0 0], [0 0 0 1 0], [0 1 0 0 0]]

[1 0 0 0 0] → I

- **Advantages**
  - **Simplicity:** It's easy to understand and implement
  - **No Ordinality Assumption:** One-hot encoding does not impose any ordinal relationships among the words, which can be beneficial in many applications
- **Limitations**
  - **High Dimensionality:** If the vocabulary size is large, the resulting vectors can be very high-dimensional and sparse (mostly zeros), leading to inefficiency
  - **Lack of Semantic Information:** One-hot encoding does not capture any semantic relationships (↳ grammer) between words. For instance, "cat" and "dog" are treated as completely unrelated, even though they are both animals
  - **Increased Computational Load:** The high-dimensional representation can lead to increased computational costs for storage and processing

Bag of Words (BoW) → collection of unique words from corpus : CountVectorizer

- It is a popular model in natural language processing (NLP) used to represent text data.
- It simplifies the text into a collection of words without considering the order or grammar.
- **Steps to Create a Bag of Words Model:**
  - **Text Preprocessing:** This involves cleaning the text, which may include removing punctuation, converting to lowercase, and eliminating stop words (common words like "and," "the," etc.).
  - **Vocabulary Creation:** Compile a list of unique words (the vocabulary) from the entire text dataset.
    - **Vector Representation:** For each document (or sentence), create a vector that counts the occurrences of each word from the vocabulary. Each position in the vector corresponds to a word, and the value at that position represents its count in the document.
- **Example:**
  - Consider two sentences:
    - "I love cats."
    - "I love dogs."



document = review
review = "I love cats. cats are white..."
↳ [1, 1, 2, 0]

  - Assuming our vocabulary is ["I", "love", "cats", "dogs"], the BoW representation would be:
    - "I love cats." → [1, 1, 1, 0]
    - "I love dogs." → [1, 1, 0, 1]

- "food" "is" "not" "good" = ' food is good '

unigram = [food, not, good]

= [1 1 1] , [1 0 1]

bigram = [(food, not), (not good)]

= [1, 1) [0 0]

trigram = [(food not good)]

= [1] [0]

N-Gram

- In natural language processing (NLP), an **n-gram** is a contiguous sequence of n items (usually words or characters) from a given text or speech
- N-grams are used to analyze the structure and context of text data
- They help capture relationships and patterns that can be useful for various NLP tasks such as text classification, language modelling, and machine translation
- **Types of N-grams**
  - **Unigram**
    - An n-gram of size 1, which consists of individual words.
    - Example: From the sentence "I love cats," the unigrams are ["I", "love", "cats"]
  - **Bigram**
    - An n-gram of size 2, which consists of pairs of consecutive words.
    - Example: From the same sentence, the bigrams are ["I love", "love cats"]
  - **Trigram**
    - An n-gram of size 3, which consists of triplets of consecutive words
    - Example: ["I love cats"]
  - **Higher-order n-grams**
    - You can have n-grams of size 4, 5, or more, depending on your analysis needs.

N-grams

How are | you?
(x) (y)

How are you?
x y

- **Applications of N-grams**
  - **Text Prediction:** N-grams can be used to predict the next word in a sentence based on the previous words.
  - **Spam Detection:** N-grams help in identifying patterns in text that are characteristic of spam or non-spam content.
  - **Sentiment Analysis:** By analyzing n-grams, one can capture phrases that convey sentiment, improving the accuracy of sentiment classification models.
  - **Machine Translation:** N-grams help in understanding the context and relationships between words in different languages.
- **Limitations of N-grams**
  - **Sparsity:** As the size of n increases, the number of possible n-grams grows exponentially, leading to sparse representations, especially in large vocabularies.
  - **Contextuality:** N-grams can lose context, as they do not capture the meaning or relationship beyond their immediate sequence.

Term Frequency-Inverse Document Frequency (TF-IDF)

- It is a statistical measure used to evaluate the importance of a word in a document relative to a collection of documents (corpus)
- It combines two components: **Term Frequency (TF)** and **Inverse Document Frequency (IDF)**
- TF-IDF is widely used in text mining and information retrieval to transform text into a meaningful representation for machine learning models
- **Applications of TF-IDF**
  - **Information Retrieval:** Enhances search engines by ranking documents based on their relevance to the search query.
  - **Text Classification:** Serves as a feature representation for machine learning algorithms.
  - **Topic Modelling:** Helps identify significant topics in a collection of documents.

Term Frequency-Inverse Document Frequency (TF-IDF)

- Components of TF-IDF
  - Term Frequency (TF)
    - Measures how frequently a term appears in a document
    - TF gives higher values to terms that appear more frequently in a document
    - $tf(t,d) = \frac{number\ of\ times\ term\ t\ appears\ in\ document\ d}{total\ number\ of\ terms\ in\ doument\ d}$
  - Inverse Document Frequency (IDF)
    - Measures how important a term is across the entire corpus
    - $IDF(t) = log(\frac{total\ number\ of\ documents}{number\ of\ documents\ containing\ term\ t})$
  - TF-IDF
    - The final TF-IDF score for a term t in a document d is calculated as:
      - TF-IDF (t, d) = TF (t, d) * IDF (t)

Word Embeddings

male = = king : 28598851 embedding queen = female

- Word embeddings are a type of word representation that allows words to be represented as dense vectors in a continuous vector space
- Unlike traditional methods such as one-hot encoding or Bag of Words, which produce high-dimensional and sparse vectors, word embeddings capture semantic meanings and relationships between words in a more efficient and meaningful way
- Key Characteristics of Word Embeddings
  - Dense Representation
    - Word embeddings typically reduce the dimensionality of the representation (e.g., from a vocabulary size of thousands to a vector of size 100-300)
  - Semantic Similarity
    - Words with similar meanings are represented by vectors that are close together in the embedding space. For example, "king" and "queen" might be close to each other, while "king" and "apple" would be far apart
  - Contextual Relationships
    - Word embeddings can capture various linguistic relationships. For example, the difference between the embeddings for "king" and "queen" can be similar to the difference between "man" and "woman."

Word Embeddings - Models

- **Word2Vec**
  - Developed by Google, Word2Vec can create word embeddings using two model architectures
    - **CBOW** predicts a target word based on its surrounding context words
    - **Skip-Gram** does the opposite, predicting the surrounding context words given a target word
- **GloVe (Global Vectors for Word Representation)**
  - Developed by Stanford, GloVe uses matrix factorization techniques on the word co-occurrence matrix to produce embeddings.
  - It leverages global word-word co-occurrence statistics from a corpus
- **FastText**:
  - Developed by Facebook, FastText represents words as bags of character n-grams
  - This allows it to capture subword information, making it effective for morphologically rich languages and handling out-of-vocabulary words
- **Contextualized Embeddings**
  - Models like ELMo, BERT, and GPT generate embeddings that are context-dependent, meaning the same word can have different embeddings based on its usage in a sentence

Word Embeddings - Applications

- **Text Classification**
  - Word embeddings serve as features for machine learning models to improve classification accuracy
- **Sentiment Analysis**
  - They help capture the nuanced meanings of words, aiding in determining sentiment
- **Machine Translation**
  - Embeddings assist in translating words and phrases across different languages
- **Information Retrieval**
  - They improve search algorithms by understanding semantic meanings

NLP

Text processing → tokenization, stop words .....

text representation → one hot, BoW, Ngram, TF-IDF, Word Embeddings

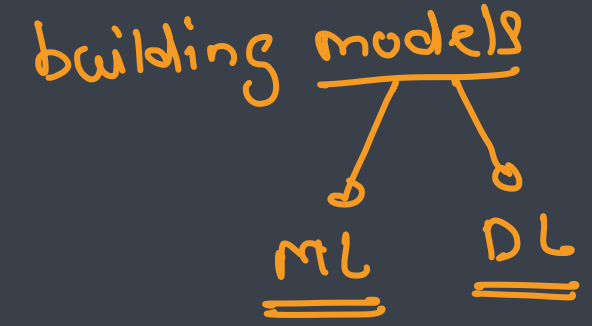
Modelling

→ ML : NB, SVM, KNN

→ DL : RNN → LSTM, GRU → Encoder, Decoder, bidirectional encoder, transformers ***

LLM

Modelling



is_active_member	has_cr_card	...

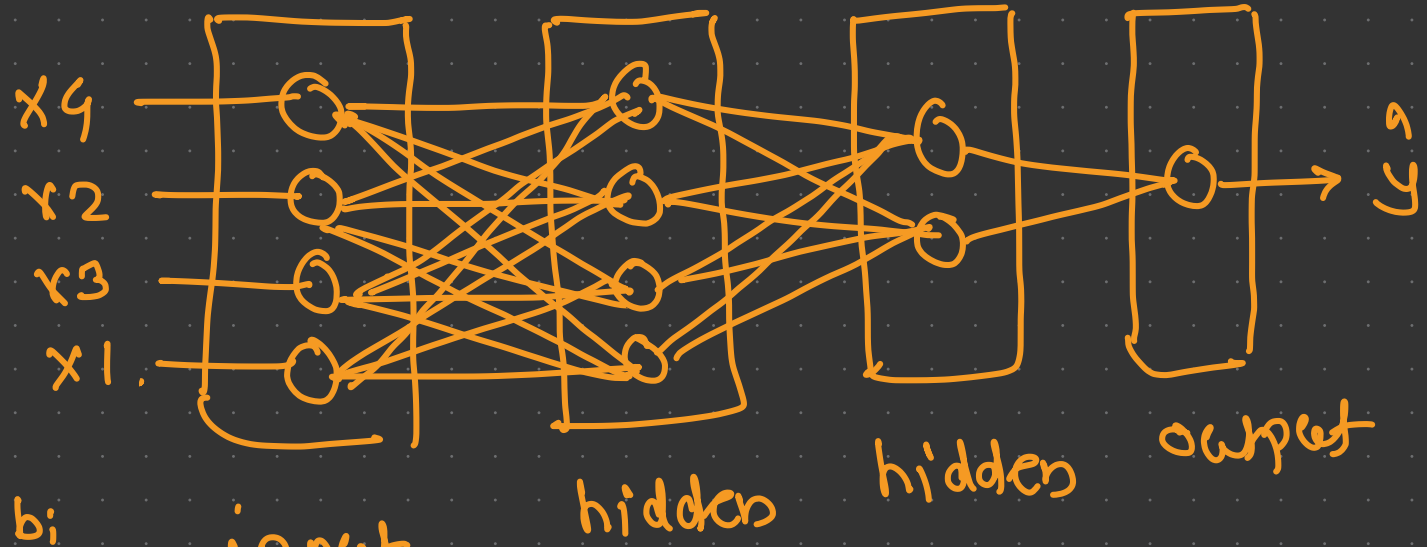
$y = \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_0$

$y = \beta_2 x_2 + \beta_1 x_1 + \dots + \beta_0$

x1	x2	y
2	3	1
3	6	.
4	2	.

x1	x2	x3	x4

y



$y = 2x_1 + 3x_2 + 2$

$y = 3x_2 + 2x_1 + 2$

$sl = \sum w_i x_i + b_i$

Vanilla ANN

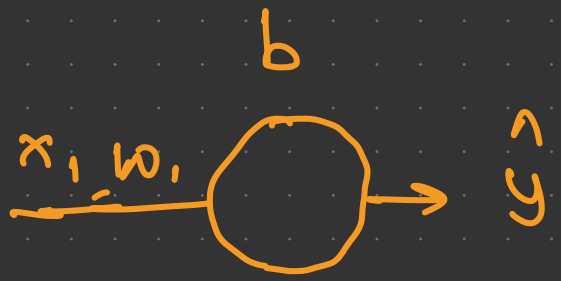
model = [weights + biases]

89%

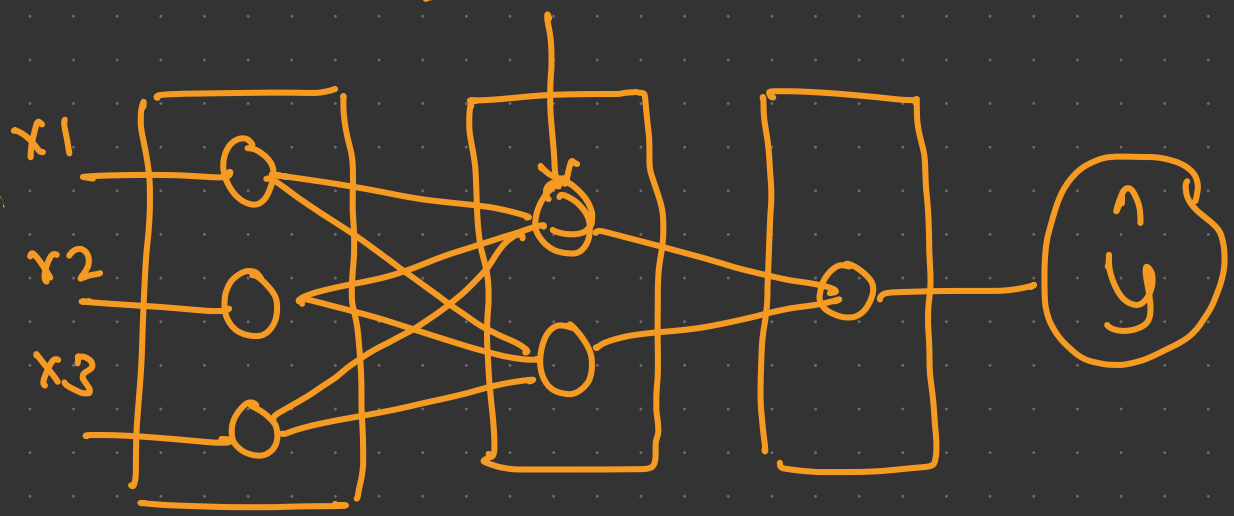
boy goes to school

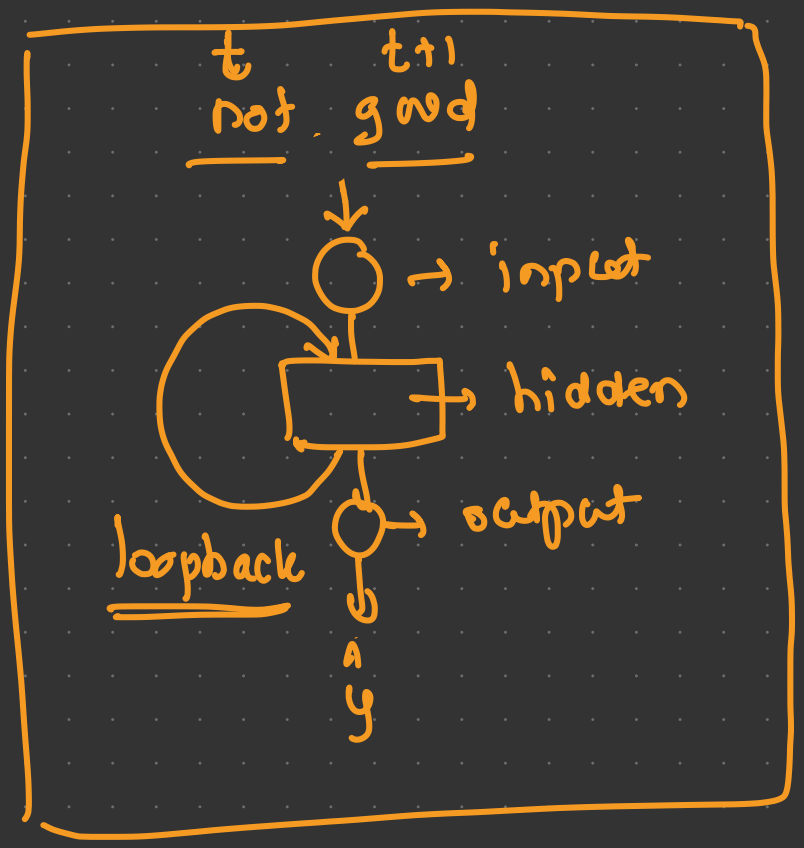
[[1 0 0] [0 1 0] [0 0 1]]

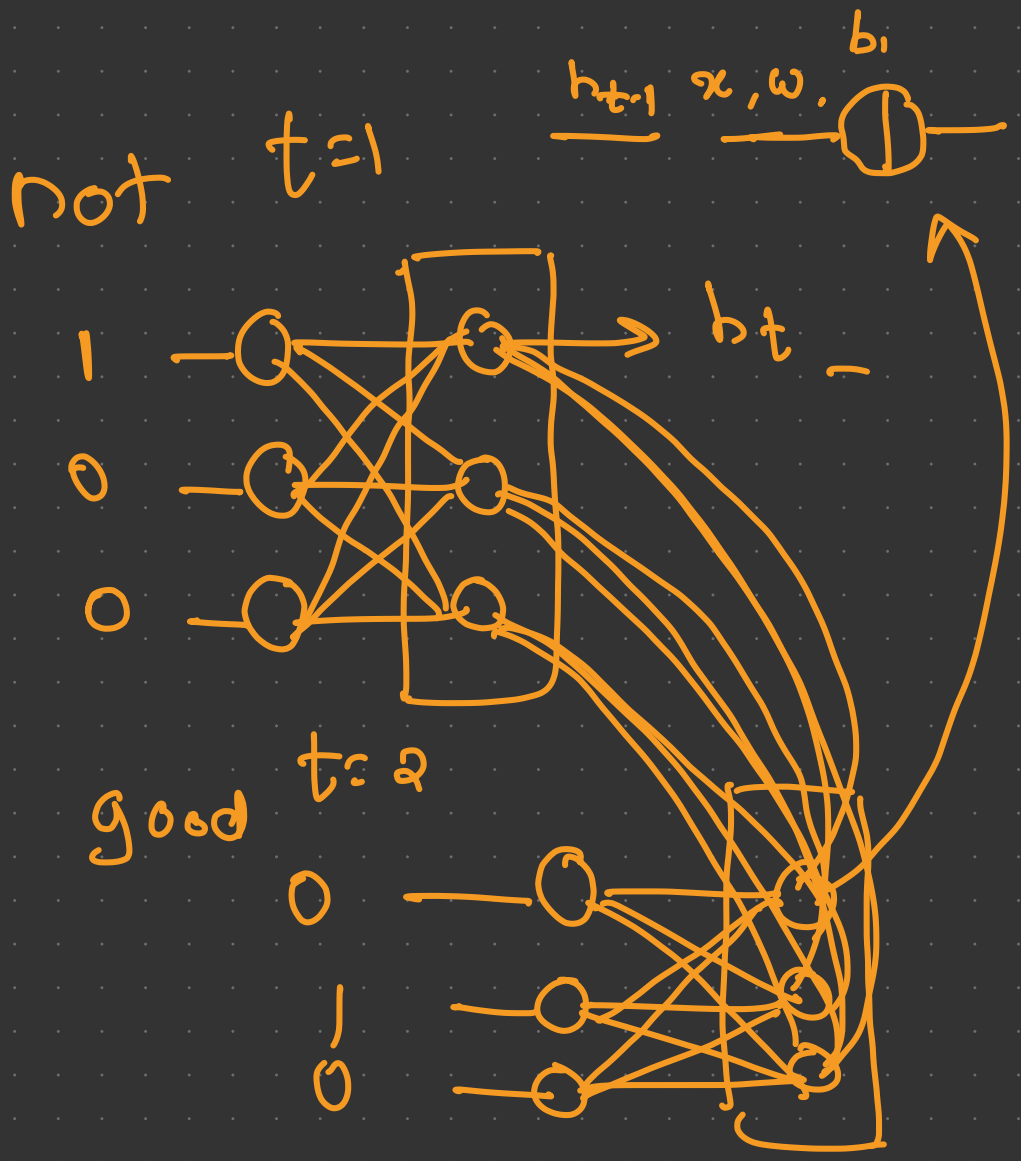
school goes to boy



neuron



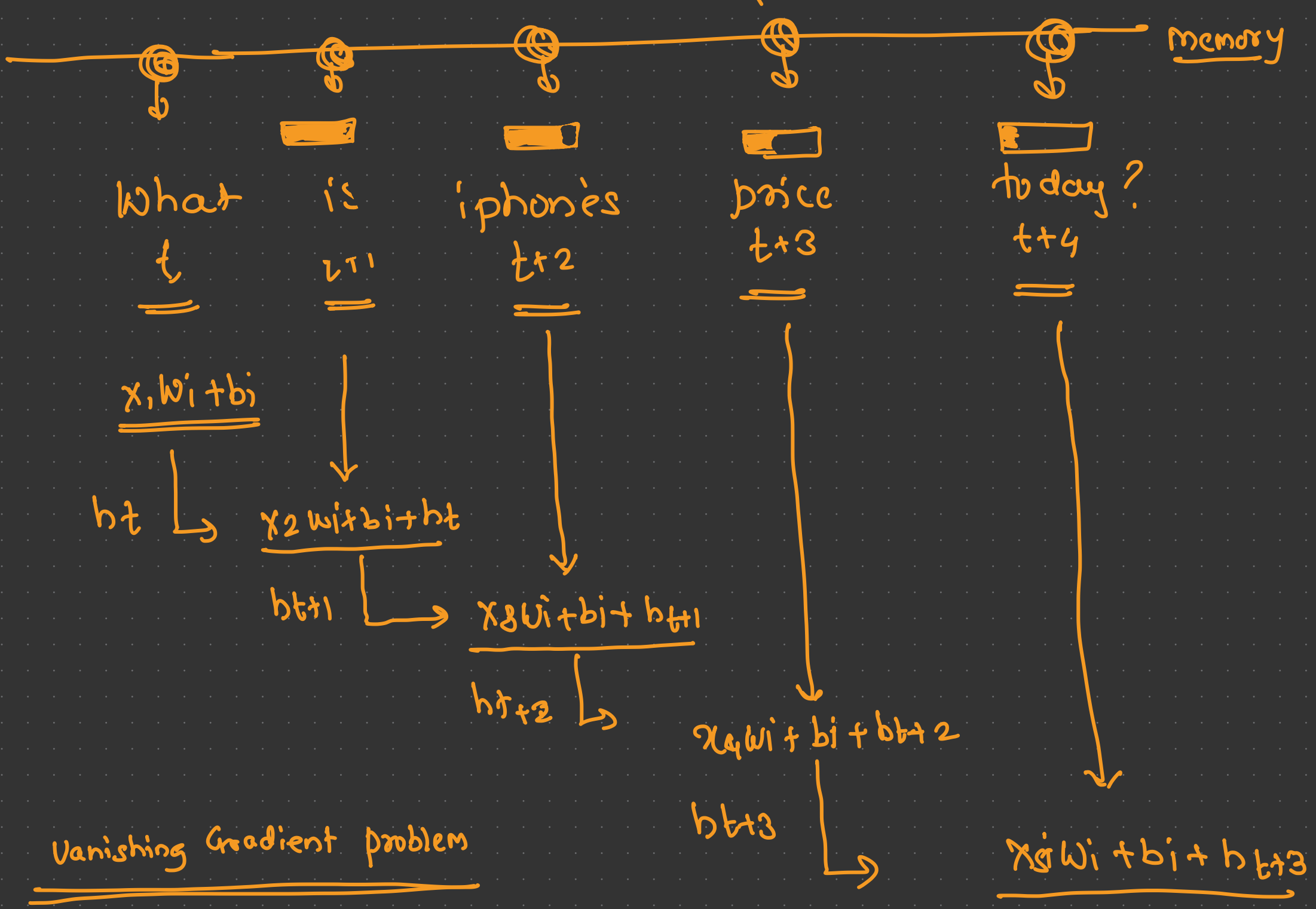




h_{t-1} $x_i w_i$ b_i

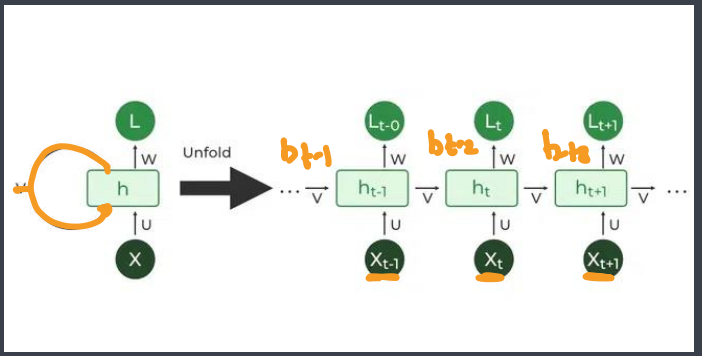
find ~~is~~ not good

sl function = $\Sigma w_i x_i + b + h_{t-1}$


memory
What
t
is
t+1
iphones
t+2
price
t+3
today ?
t+4
x1 w1 + b1
ht
x2 w1 + b1 + ht
ht+1
x3 w1 + b1 + ht+1
ht+2
x4 w1 + b1 + ht+2
ht+3
x5 w1 + b1 + ht+3
Vanishing Gradient problem

RNN

- A Recurrent Neural Network (RNN) is a type of artificial neural network designed for processing sequences of data
- Unlike traditional feedforward neural networks, RNNs have connections that loop back on themselves, allowing them to maintain a hidden state that can capture information about previous inputs
- This makes RNNs particularly suited for tasks like time series prediction, natural language processing, and any application where the order of inputs is important
- The key features of RNNs include:
  - **Sequential Processing:** RNNs process input sequences one element at a time while maintaining a hidden state that updates with each input.
  - **Memory:** They can remember information from previous inputs, which helps in understanding context in sequences.
  - **Training:** RNNs are typically trained using backpropagation through time (BPTT), a method that handles the temporal dependencies in the data.



Architecture of RNN

- The architecture of a Recurrent Neural Network (RNN) is designed to handle sequential data through the use of loops in its structure.
- **Input Layer**
  - The input layer receives sequences of data, where each input can be a vector representing a single time step (e.g., a word embedding in natural language processing)
- **Hidden Layers**
  - **Recurrent Connections**: The hidden layer(s) contain recurrent connections, meaning that the output from the previous time step is fed back into the network along with the current input. This creates a loop, allowing the network to maintain a hidden state over time
  - **Hidden State**: At each time step, the hidden state is updated based on the current input and the previous hidden state
- **Output Layer**
  - The output layer produces predictions for each time step or a single output for the entire sequence, depending on the task (e.g., classification, sequence generation)
- **Activation Functions**
  - Non-linear activation functions (like tanh or ReLU) are commonly used to introduce non-linearity into the network, which helps it learn complex patterns
- **Loss Function and Optimization**
  - A loss function (e.g., categorical cross-entropy for classification tasks) measures the difference between the predicted outputs and the actual targets. An optimizer (like Adam or SGD) is used to update the weights during training

RNN Advantages

- **Handling Sequential Data**
  - RNNs are specifically designed to process sequences, making them ideal for tasks like time series prediction, natural language processing, and speech recognition.
- **Memory of Previous Inputs**
  - The architecture allows RNNs to maintain a hidden state that carries information from previous time steps. This ability to remember past inputs helps capture context and relationships in the data.
- **Flexible Input Lengths**
  - RNNs can accept input sequences of varying lengths, making them suitable for tasks where the input size is not fixed, such as sentences of varying lengths in NLP.
- **Parameter Sharing**
  - RNNs share parameters across different time steps, reducing the number of parameters that need to be learned and allowing the model to generalize better across different inputs.
- **End-to-End Learning**
  - RNNs can be trained end-to-end, allowing the model to learn directly from raw input data to the final output, which simplifies the training process.
- **Adaptability to Different Tasks**
  - RNNs can be adapted to a variety of tasks by adjusting the output layer, whether it's for classification, regression, or sequence generation.
- **Rich Representations**
  - RNNs can learn complex representations of the input data, capturing nuanced patterns and dependencies over time.

RNN Limitations

- **Vanishing and Exploding Gradients**
  - During training, especially with long sequences, gradients can either diminish (vanishing) or grow excessively (exploding). This makes it challenging to learn long-range dependencies effectively.
- **Difficulty with Long-Term Dependencies**
  - While RNNs can maintain a hidden state, they often struggle to remember information from earlier time steps when the sequences are long. This can lead to loss of context and information over time.
- **Sequential Processing**
  - RNNs process data sequentially, which can be slower compared to parallel processing methods like convolutional neural networks (CNNs). This makes training and inference time-consuming for long sequences.
- **Limited Memory Capacity**
  - The fixed size of the hidden state means that RNNs have a limited capacity to remember information. This can be a significant limitation in tasks requiring the retention of a lot of contextual information.
- **Complexity in Training**
  - RNNs can be harder to train effectively due to their recurrent structure, which can complicate the optimization process.
- **Sensitivity to Initial Conditions**
  - The performance of RNNs can be sensitive to the initialization of weights and the choice of hyperparameters, leading to variability in results
- **Overfitting**
  - RNNs, particularly deeper architectures, can overfit to training data, especially if the dataset is small or not diverse enough

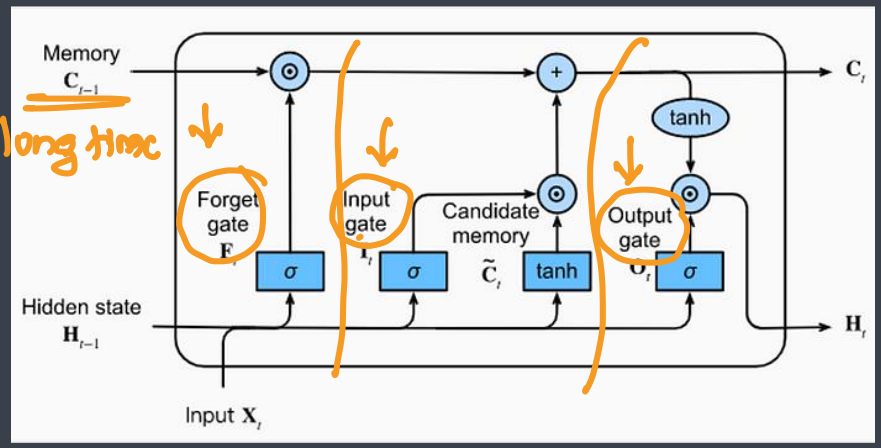
Variants of RNN

- To address some of the limitations of vanilla RNNs, several advanced architectures have been developed:
  - **LSTM (Long Short-Term Memory)**
    - Incorporates memory cells and gating mechanisms (input, forget, and output gates) to better capture long-term dependencies and manage the flow of information
  - **GRU (Gated Recurrent Unit)**
    - A simplified version of LSTMs, using fewer gates while still effectively handling long-range dependencies
  - **Bidirectional RNNs**
    - Process the input sequence in both forward and backward directions, allowing the model to have access to past and future context

LSTM

Gate - mathematical calculation

- Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) specifically designed to address the limitations of standard RNNs, particularly their difficulties in capturing long-range dependencies in sequential data
- LSTMs are widely used in various NLP tasks, such as:
  - Sentiment analysis
  - Text generation
  - Machine translation
  - Speech recognition



- Architecture
  - **Memory Cells:** LSTMs use memory cells to maintain information over long periods. These cells can store information and keep it for as long as needed.
  - **Gates:** LSTMs have three types of gates that control the flow of information:
    - **Input Gate:** Decides how much of the incoming information to store in the memory cell
    - **Forget Gate:** Determines what information should be discarded from the memory cell
    - **Output Gate:** Controls how much of the memory cell's content should be output to the next layer or time step
  - **Cell State:** The cell state is a key feature of LSTMs, allowing them to carry information across many time steps without it being significantly altered, thus preserving long-term dependencies

LSTM

- **Working** → memory cell
  - **Input Processing**
    - At each time step, the LSTM takes in the current input and the previous hidden state
  - **Gate Computations** (input gate ↑, ↑ forget gate)
    - The gates compute values that determine what information to keep or discard
    - The input gate takes the current input and the previous hidden state to create a vector that determines what new information to add to the cell state. → output (hidden state)
    - The forget gate decides what information from the cell state should be removed.
    - The output gate determines what the next hidden state should be based on the current cell state.
  - **Updating Cell State**
    - The cell state is updated based on the input and forget gate decisions
- **Advantages of LSTMs**
  - **Long-Term Dependency Capture:** LSTMs can effectively learn from long sequences, making them suitable for tasks like language modelling, machine translation, and time series forecasting.
  - **Reduced Gradient Issues:** The gating mechanisms help mitigate vanishing and exploding gradient problems common in standard RNNs.

LSTM - Limitations

- **Complexity and Training Time**
  - **Long Training Times:** LSTMs can be computationally intensive, requiring longer training times compared to simpler models.
  - **Complex Architecture:** Their complex structure (multiple gates and cell states) makes them harder to implement and tune compared to simpler models.
- **Difficulty with Long Sequences**
  - **Vanishing Gradient Problem:** Although LSTMs are designed to mitigate the vanishing gradient problem, they can still struggle with very long sequences, leading to poor learning of long-range dependencies.
  - **Memory Limitations:** They may still forget important information from earlier in a long sequence, depending on the architecture and hyperparameters used.
- **Overfitting**
  - **Sensitivity to Overfitting:** With many parameters, LSTMs can easily overfit, especially when the training dataset is small or not representative of the problem domain.
  - **Need for Regularization:** Regularization techniques such as dropout are often needed, which can complicate model training.
- **Limited Parallelization**
  - **Sequential Processing:** LSTMs process data sequentially, making them less efficient for parallelization on modern hardware compared to architectures like Transformers, which can process entire sequences at once.
- **Difficulties in Handling Irregular Time Series** 1998, 1999, 2000 , 1999, 2002, 2003, 2007
  - **Time Gaps:** LSTMs may struggle with irregularly sampled time series data where data points are not evenly spaced in time.
- **Less Effective for Short Contexts**
  - **Short Context Dependency:** In some cases, simpler architectures like GRUs or even vanilla RNNs may outperform LSTMs when dealing with shorter sequences.
- **Performance on Certain Tasks**
  - **Not Always the Best Choice:** For some tasks, especially in NLP, newer architectures like Transformers (e.g., BERT, GPT) have shown to be significantly more effective than LSTMs.

GRU

it consists of memory cell

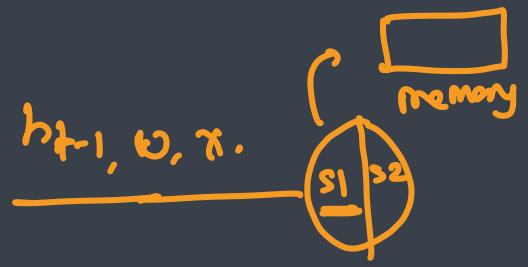
- A Gated Recurrent Unit (GRU) is a type of RNN architecture that is similar to Long Short-Term Memory (LSTM) networks but with a simplified structure
- GRUs were introduced to improve the efficiency of training while still effectively capturing dependencies in sequential data
- GRUs are widely used in various tasks, including:
  - Natural language processing (e.g., text classification, machine translation)
  - Speech recognition
  - Time series prediction
- **Advantages of GRUs**
  - **Efficiency:** GRUs are typically faster to train than LSTMs due to their simpler architecture, making them suitable for tasks with limited computational resources.
  - **Performance:** In many applications, GRUs perform comparably to LSTMs, making them a strong alternative when training speed is a consideration.

GRU

- **Architecture**
  - **Gates:** GRUs have two main gates
    - **Update Gate:** Controls how much of the previous hidden state to retain and how much of the new input to incorporate
    - **Reset Gate:** Determines how much of the previous hidden state to forget when processing the new input
  - **Single Hidden State:** Unlike LSTMs, which maintain both a cell state and a hidden state, GRUs use a single hidden state, simplifying the architecture.
  - **Combined Update:** The update gate and reset gate work together to manage information flow, allowing GRUs to maintain context while being computationally efficient.

- **Working**
  - At each time step, GRUs take the current input and the previous hidden state to compute the new hidden state
  - The reset gate decides how much of the past information to forget, and the update gate balances between retaining the previous hidden state and incorporating new information
  - The hidden state is then updated accordingly

LSTM vs GRU



- **Architecture**
  - **LSTM:**
    - Consists of three gates: **input gate**, **forget gate**, and **output gate**
    - Maintains a separate cell state in addition to the hidden state, allowing it to manage and store information over long sequences
  - **GRU:**
    - Combines the input and forget gates into a single **update gate** and uses a **reset gate**. This results in a simpler architecture
    - Does not maintain a separate cell state; instead, it directly modifies the hidden state
- **Complexity**
  - **LSTM:** More complex due to its three gates and separate cell state, leading to increased computational overhead
  - **GRU:** Simpler than LSTM, making it generally faster to compute while still being effective for many tasks
- **Performance**
  - **LSTM:** Often performs better on tasks requiring longer memory due to its ability to manage the flow of information more explicitly
  - **GRU:** Can perform comparably to LSTMs on various tasks and may sometimes outperform LSTMs, especially with smaller datasets or less complex sequences

LSTM vs GRU

- **Training Time**
  - **LSTM:** Typically requires more time to train because of its complexity
  - **GRU:** Generally faster to train due to its simpler structure, which can be advantageous in scenarios with limited computational resources
- **Use Cases**
  - **LSTM:** Commonly used in applications where long-range dependencies are critical, such as language modelling, machine translation, and speech recognition
  - **GRU:** Often used in similar applications, particularly when computational efficiency is a priority or when working with smaller datasets

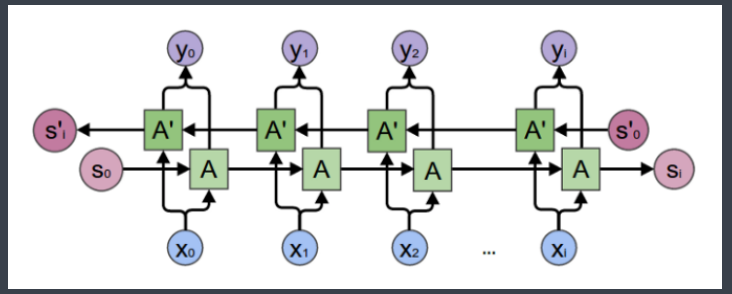
LSTM vs GRU

↳ performance → accuracy

- **When to Use LSTM**
  - **Complex Relationships**: If your data has complex dependencies and long-range relationships that need to be captured, LSTMs may perform better due to their more intricate architecture, which includes separate cell states and multiple gates
  - **Long Sequences**: For tasks involving long sequences where retaining information over extended time periods is critical, LSTMs are often preferred due to their design to handle vanishing gradient issues better than traditional RNNs
  - **Established Baseline:** If you're working on a problem where LSTMs have been the standard approach and you want to build on existing work, they may be a suitable choice
  - **Fine-Grained Control**: LSTMs provide more control over what information to keep or forget due to their architecture, which might be beneficial in applications like machine translation or text generation
- **When to Use GRU**
  - **Faster Training**: GRUs are generally faster to train because they have fewer parameters (due to the absence of a separate cell state). If you need quicker experimentation and iterations, GRUs can be advantageous
  - **Sufficient Performance**: For many tasks, GRUs perform comparably to LSTMs. If you find that GRUs yield satisfactory results, they can be a simpler alternative
  - **Less Complex Data**: If the relationships in your data are not overly complex or if you're working with shorter sequences, GRUs can be an effective choice
  - **Resource Constraints**: When computational resources are limited, and model size or training time is a concern, GRUs can provide a lightweight alternative while still maintaining performance

Bidirectional RNN

- A **Bidirectional RNN (BRNN)** is an extension of the traditional Recurrent Neural Network (RNN) architecture that allows the model to capture context from both past and future states when processing sequences



- How it works
  - **Two RNNs**
    - A bidirectional RNN consists of two separate RNNs
      - **Forward RNN:** Processes the input sequence from the beginning to the end (left to right)
      - **Backward RNN:** Processes the input sequence from the end to the beginning (right to left)
  - **Combined Outputs**
    - The outputs from both RNNs at each time step are combined, typically through concatenation or averaging
    - This allows the model to utilize information from both directions
  - **Context Awareness**
    - By incorporating information from both past (via the forward RNN) and future (via the backward RNN), bidirectional RNNs can capture more nuanced contextual information
    - This is particularly beneficial for tasks where the meaning of a word or token depends on both preceding and following elements in the sequence

Bidirectional RNN

- **Advantages of Bidirectional RNNs**
  - **Improved Contextual Understanding:** They can better understand sequences where the context is influenced by elements that appear later in the sequence. For instance, in language tasks, the meaning of a word can be affected by words that follow it.
  - **Enhanced Performance:** In many NLP tasks, such as sentiment analysis, named entity recognition, and machine translation, bidirectional RNNs often outperform unidirectional RNNs due to their ability to capture richer context.

- **Applications**
  - **Natural Language Processing (NLP):** For tasks like sentiment analysis, language modeling, and text classification.
  - **Speech Recognition:** To improve the understanding of spoken language by considering both preceding and succeeding audio frames.
  - **Time Series Prediction:** In cases where future data points can help predict past values.

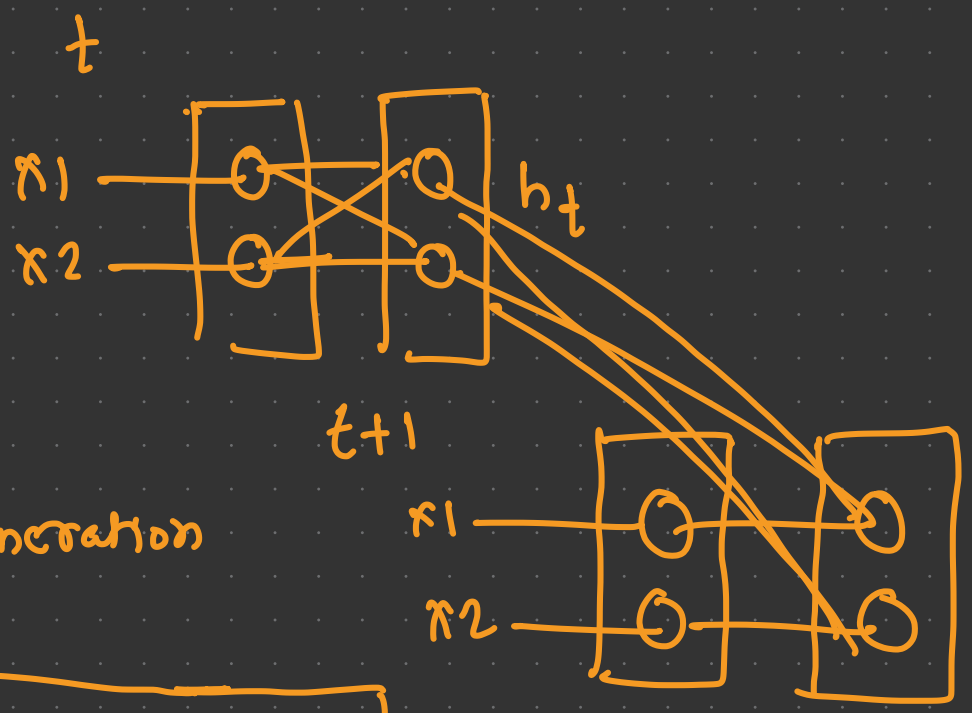
Types of RNN

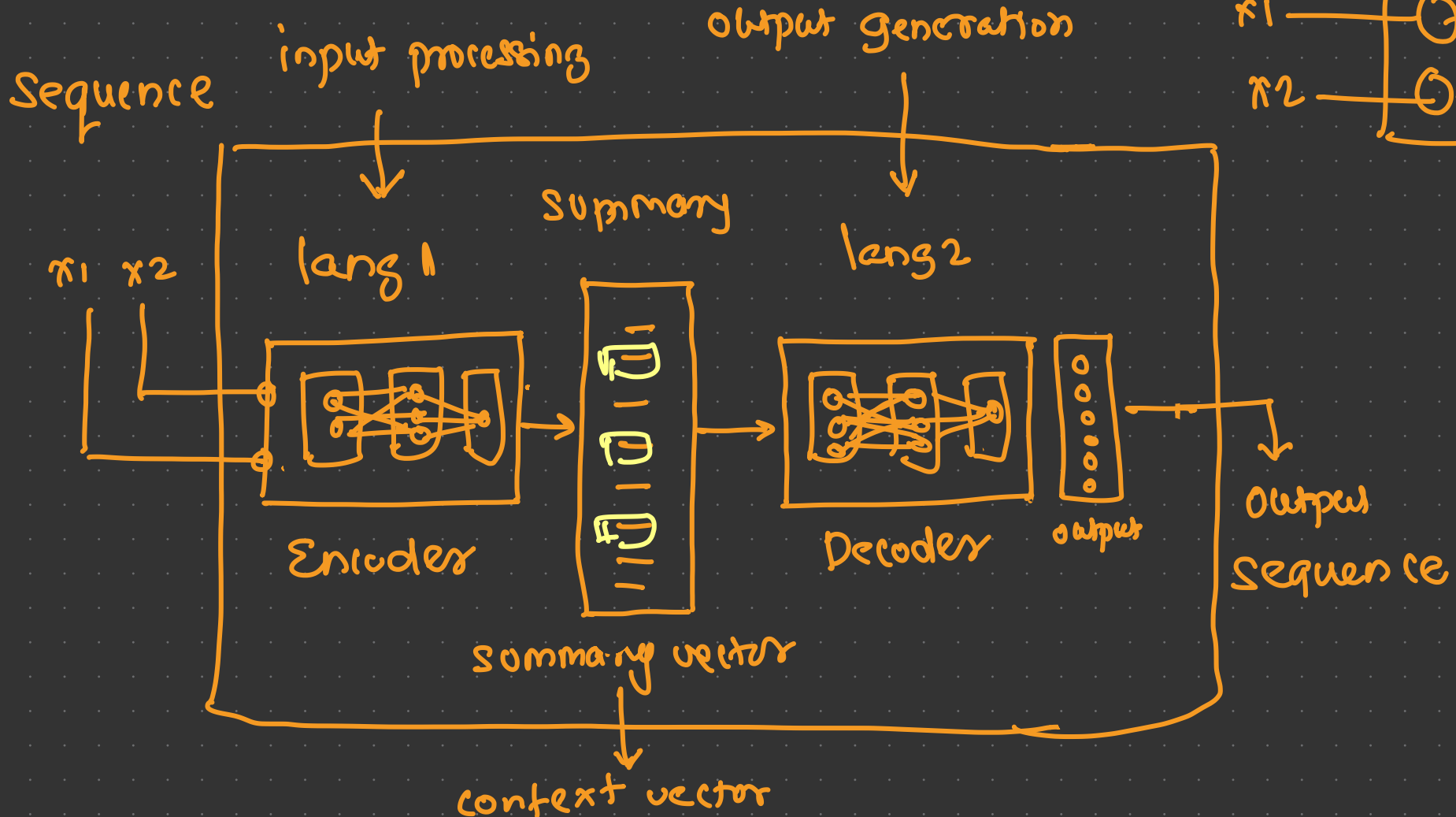
input output

1. one to one RNN → sentiment analysis
2. one to many RNN → generating data
3. many to one RNN → sentiment analysis
4. many to many RNN → generating data

Encode → translate from one to another

Decode → opposite of encode





Encoder – Decoder

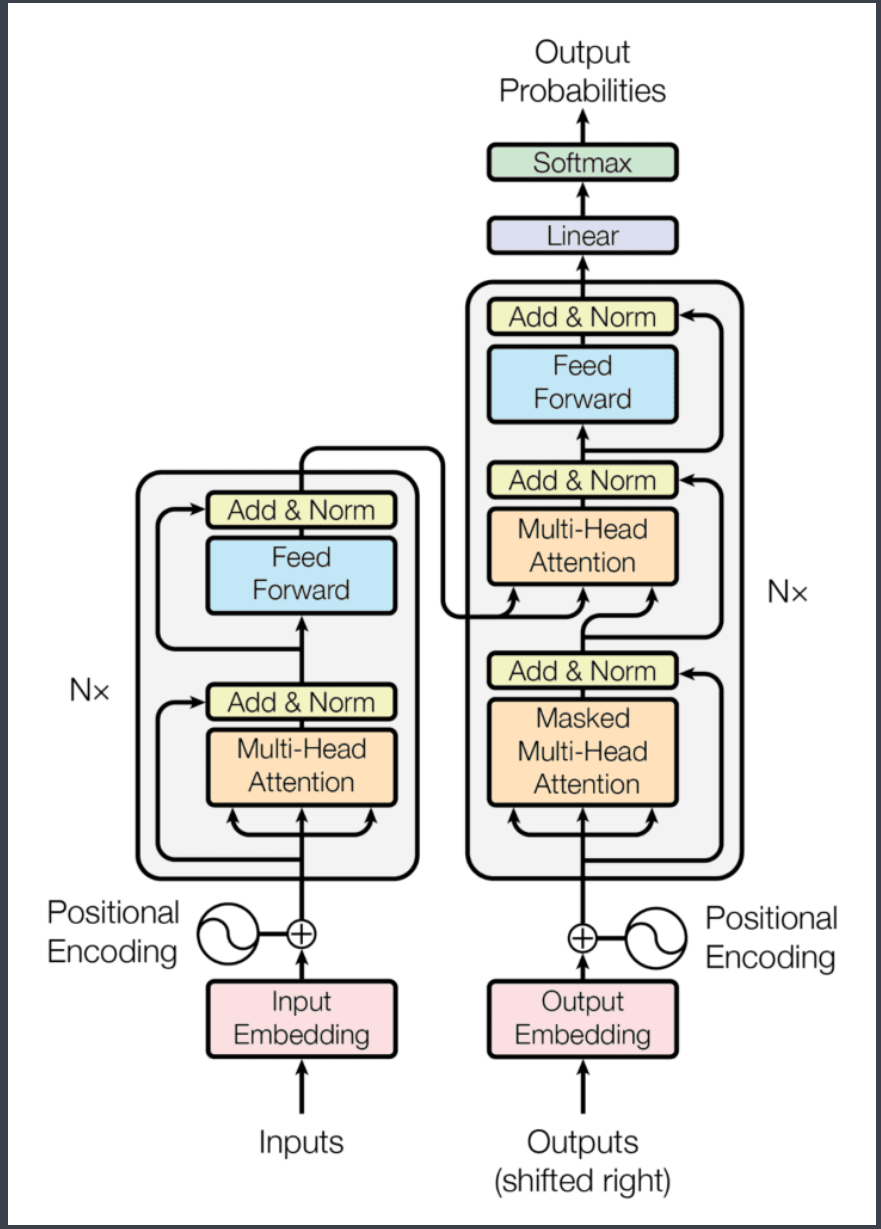
- The **encoder-decoder architecture** is a neural network framework commonly used for sequence-to-sequence tasks, such as machine translation, text summarization, and image captioning
- By separating the input processing (encoder) from the output generation (decoder), and often incorporating attention mechanisms, it enables models to achieve state-of-the-art performance in a variety of applications
- This architecture is foundational in many modern neural network designs, including those based on transformers
- It consists of two main components: the **encoder** and the **decoder**
- **Encoder**
  - **Function:** The encoder processes the input sequence and compresses the information into a fixed-size context vector (or a sequence of vectors).
  - **Structure:**
    - The encoder typically consists of one or more layers of RNNs, LSTMs, GRUs, or transformers.
    - Each input token (e.g., a word in a sentence) is embedded into a continuous vector representation. These embeddings are then fed into the encoder.
  - **Output:** The final hidden state (or states) of the encoder serves as the context vector, representing the entire input sequence. This vector encapsulates the relevant information from the input that the decoder will use.

Encoder – Decoder

- **Decoder**
  - **Function:** The decoder generates the output sequence from the context vector produced by the encoder.
  - **Structure:**
    - Like the encoder, the decoder can also be made of RNNs, LSTMs, GRUs, or transformers
    - The decoder generates the output sequence token by token, starting with a special start token. For each token generated, the decoder uses its previous output (or the context vector) to inform the next prediction
  - **Output:** It produces a sequence of output tokens, which can be words or other data types depending on the task
- **Attention Mechanism (Optional)**
  - In many modern encoder-decoder models, an **attention mechanism** is incorporated to allow the decoder to focus on different parts of the input sequence dynamically
  - Instead of relying solely on the context vector, the decoder can attend to different hidden states of the encoder at each time step, enabling it to utilize relevant parts of the input sequence when generating each output token
- **Applications**
  - **Machine Translation:** Translating text from one language to another (e.g., English to French).
  - **Text Summarization:** Generating concise summaries from longer documents.
  - **Image Captioning:** Creating descriptive captions for images by combining visual and textual information.
  - **Speech Recognition:** Converting spoken language into text.

Transformers

- A **transformer** is a type of neural network architecture designed to handle sequential data, primarily used in natural language processing (NLP) tasks but increasingly applied in other domains like computer vision and audio processing
- Introduced in the paper "Attention is All You Need" by Vaswani et al. in 2017, transformers leverage a mechanism called **self-attention** to process input data in parallel rather than sequentially



Transformers - Components

- **Self-Attention Mechanism**
  - Self-attention allows the model to weigh the importance of different words (or tokens) in a sequence when processing each word
  - This helps the model capture relationships between words regardless of their distance from each other in the sequence.
- **Multi-Head Attention**
  - Instead of a single attention mechanism, transformers use multiple attention heads to allow the model to focus on different aspects of the input simultaneously
  - Each head learns different attention patterns, which enhances the model's ability to understand context
- **Positional Encoding:**
  - Since transformers do not have a built-in sense of order (unlike RNNs), positional encodings are added to the input embeddings
  - These encodings provide information about the position of each token in the sequence, allowing the model to understand the order of words
- **Feed-Forward Neural Networks**
  - After the attention layers, transformers include feed-forward networks that process the output of the attention mechanism independently for each position
  - This consists of two linear transformations with an activation function in between (typically ReLU)
- **Layer Normalization and Residual Connections**
  - Each sub-layer (attention and feed-forward) is followed by layer normalization and a residual connection
  - This helps stabilize training and improves gradient flow, allowing for deeper models

Transformers

- Architecture
  - Encoder
    - Composed of multiple identical layers (often 6 or more), the encoder takes the input sequence, processes it through self-attention and feed-forward networks, and generates a set of continuous representations (contextual embeddings)
  - Decoder
    - Also made up of multiple identical layers, the decoder generates the output sequence
    - It includes a masked self-attention mechanism to prevent attending to future tokens during training, as well as an additional attention layer that allows it to attend to the encoder's outputs
- Applications
  - **Machine Translation:** Converting text from one language to another.
  - **Text Summarization:** Generating concise summaries from longer texts.
  - **Question Answering:** Answering questions based on given contexts.
  - **Text Generation:** Generating human-like text based on prompts (e.g., GPT models)
- Advantages
  - **Parallelization:** Unlike RNNs, transformers can process tokens in parallel, significantly speeding up training.
  - **Long-Range Dependencies**: They excel at capturing long-range dependencies in sequences, making them effective for complex language tasks.
  - **Scalability:** Transformers scale well with data and model size, leading to improved performance with larger models.

BERT

- **BERT** (Bidirectional Encoder Representations from Transformers) is a transformer-based model developed by Google in 2018, specifically designed for understanding the context of words in search queries and various natural language processing (NLP) tasks
- Features
  - Bidirectional Context
    - Unlike previous models that read text sequences in a unidirectional manner (either left-to-right or right-to-left), BERT reads text in both directions simultaneously
    - This bidirectional approach allows it to capture a deeper understanding of context & relationships between words
  - Transformers Architecture
    - BERT is based on the transformer architecture, utilizing self-attention mechanisms to weigh the importance of different words in a sentence when making predictions
  - Pre-training and Fine-tuning
    - BERT is pre-trained on a large corpus of text (like Wikipedia and BookCorpus) using two tasks
      - **Masked Language Modeling (MLM)**: Randomly masks some words in the input and trains the model to predict those masked words based on the surrounding context.
      - **Next Sentence Prediction (NSP)**: Trains the model to understand the relationship between two sentences by predicting whether the second sentence follows the first in the text.
    - After pre-training, BERT can be fine-tuned on specific tasks like sentiment analysis, named entity recognition, or question answering

BERT

- **Applications**
  - **Sentiment Analysis:** Classifying text based on sentiment (positive, negative, neutral).
  - **Named Entity Recognition (NER):** Identifying and classifying entities in text (e.g., names, locations).
  - **Question Answering:** Providing answers to questions based on context.
  - **Text Classification:** Assigning categories to text documents.
- **Advantages of BERT**
  - **Deep Understanding of Context:** BERT's bidirectional training enables it to understand nuances and context more effectively than previous models.
  - **Transfer Learning:** The pre-training and fine-tuning process allows BERT to be applied to a variety of tasks with relatively little task-specific training data.
  - **State-of-the-Art Performance:** BERT has achieved state-of-the-art results on numerous NLP benchmarks and datasets, setting new performance standards.
- **Variants and Extensions**
  - **RoBERTa:** A variant that optimizes BERT's training process and uses more data.
  - **DistilBERT:** A smaller, faster version of BERT designed to maintain most of its performance while being more efficient.
  - **ALBERT:** A lighter and more efficient version of BERT with shared parameters across layers.

LLM

model → huge data (trillions)

- These are sophisticated machine learning models designed to understand, generate, and manipulate human language
- They very large deep learning models that are pre-trained on vast amounts of data
- The underlying transformer is a set of neural networks that consist of an encoder and a decoder with self-attention capabilities
- **Architecture of LLMs (Transformer Architecture)**
  - LLMs are primarily based on the **transformer architecture**, which consists of an encoder and decoder. However, many LLMs use only the decoder (e.g., GPT models) or just the encoder (e.g., BERT)
  - **Self-Attention Mechanism:** This allows the model to consider the entire context of a sequence when processing each word. It calculates attention scores for all words in the input, determining which words should influence each other based on their relevance
  - **Feed-Forward Networks:** After the self-attention mechanism, the output is processed through feed-forward neural networks, which add non-linearity to the model
  - **Layer Normalization and Residual Connections:** These techniques stabilize training and allow deeper networks to be trained effectively

↑ error

LLM

- **Training Process**
  - **Pre-training**
    - LLMs are initially trained on a vast corpus of text using unsupervised learning. This phase aims to capture the underlying structure and semantics of language.
    - **Masked Language Modelling (MLM):** In models like BERT, random words in sentences are masked, and the model learns to predict them based on the context. This encourages a deep understanding of word relationships.
    - **Next Sentence Prediction (NSP):** Also used in BERT, this task trains the model to understand relationships between sentences.
  - **Fine-tuning**
    - After pre-training, LLMs can be fine-tuned on specific tasks with smaller, labelled dataset
    - This step helps the model adapt to particular domains or tasks (e.g., sentiment analysis, named entity recognition)
- **Advantages**
  - **High Performance:** LLMs achieve state-of-the-art results across various benchmarks in NLP, including the GLUE and SuperGLUE benchmarks
  - **Contextual Awareness:** Their ability to consider the entire context of a sentence or passage leads to more accurate and relevant outputs.
  - **Transfer Learning:** The pre-training and fine-tuning process allows LLMs to leverage knowledge from a broad range of texts, improving performance on specific tasks even with limited data.

LLM

- Use Cases
  - **Text Generation:** LLMs like GPT-3 can generate coherent and contextually relevant text, making them useful for creative writing, content creation, and dialogue generation
  - **Question Answering:** LLMs can provide answers to questions based on provided contexts, making them suitable for customer service bots and interactive assistants
  - **Translation:** Many LLMs can translate text between languages effectively, leveraging their understanding of language structure and context
  - **Summarization:** LLMs can condense long articles or documents into shorter summaries while retaining essential information, aiding information retrieval
  - **Sentiment Analysis:** LLMs can analyze the sentiment of a text (positive, negative, neutral), which is useful for brand monitoring, customer feedback analysis, and social media monitoring
- **Examples of LLMs**
  - **GPT:** Developed by OpenAI, known for its strong text generation capabilities
  - **BERT:** Designed primarily for understanding language rather than generating it
  - **T5 (Text-to-Text Transfer Transformer):** A model that treats all NLP tasks as text generation problems
  - **Claude:** Developed by Anthropic, focusing on safe and ethical AI interactions
  - **Gemini:** Developed by Google known for its performance