Natural Language Processing

Naeemullah Khan
naeemullah.khan@kaust.edu.sa

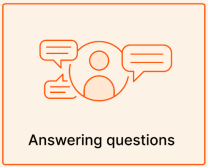
KAUST Academy
King Abdullah University of Science and Technology

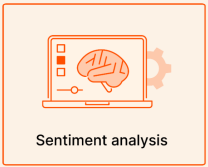
June 23, 2025

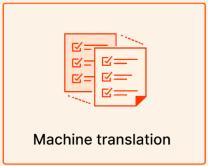
Table of Contents

Natural language processing (NLP) tasks

NLP is the process through which AI is taught to **understand the rules and syntax of language**, programmed to **develop complex algorithms to represent those rules**, and then made to **use those algorithms to carry out specific tasks** like these.







Motivation

- **Why NLP?**
  - Humans speak different languages in many forms: text, speech, emojis!
  - Computers don't "understand" — they deal with 1s and 0s.
  - NLP bridges the gap between human language and machines.
- **Examples:**
  - Google Translate
  - ChatGPT
  - Siri/Alexa
  - Email spam filters
- **Goal:** Help computers understand, interpret, and even generate human language.

Learning Outcomes

By the end of this session, you'll be able to:

- Understand key concepts and challenges in NLP.
- Know how RNN, LSTM, GRU work (without getting stuck in math!).
- Explore how attention and transformers changed the NLP game.
- Get hands-on intuition to build simple NLP models.
- Be ready to dive into modern language models (like GPT, BERT, etc.)

NLP: **Basics**

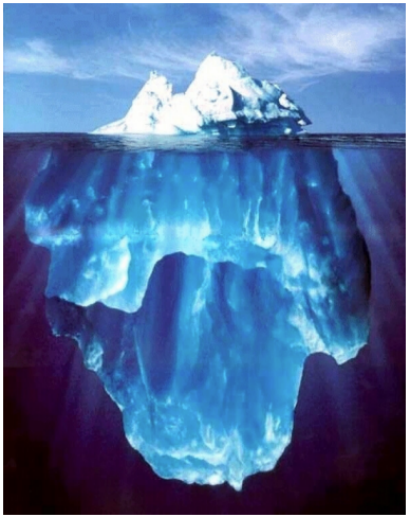
What is NLP?

- NLP stands for Natural Language Processing.
- It helps computers understand and work with human languages, like English or Arabic.
- With NLP, computers can read, listen, write, or even talk like humans.
- Examples: chatbots, language translators, voice assistants.

Text Data is Superficial

An iceberg is a large piece of freshwater ice that has broken off from a snow-formed glacier or ice shelf and is floating in open water.

.. But Language is Complex



An iceberg is a large piece of freshwater ice that has broken off from a snow-formed glacier or ice shelf and is floating in open water.

NLP Basics – What It's All About

Teaching machines to **understand and generate human language**

- **Text preprocessing**:
  Cleaning and preparing raw text data (removing noise, tokenization, normalization).
- **Understanding meaning**:
  Extracting meaning from text using techniques like part-of-speech tagging, named entity recognition, and sentiment analysis.
- **Language modeling**:
  Building models that can predict or generate text, such as autocomplete or next-word prediction.
- **Translation, summarization, etc.**:
  Enabling applications like machine translation, text summarization, question answering, and more.

Text data is everywhere: tweets, reviews, articles, chats! NLP helps us make sense of this vast information.

Core NLP Tasks

Task	What It Does	Example
Tokenization	Split text into words	`"I love NLP"` → `["I", "love", "NLP"]`
POS Tagging	Label grammar tags	`"Dogs bark"` → [Noun, Verb]
Named Entity Recognition	Find names, places, etc.	`"Christopher Nolan lives in Los Angeles"`
Sentiment Analysis	Detect mood	`"This movie was amazing!"` → Positive
Machine Translation	Language to language	English → French

Core NLP Tasks and Examples

Core NLP Tasks (cont.)

Text Preprocessing

- **Lowercase everything**:
  Example: `"NLP"` $\rightarrow$ `"nlp"`
- **Removing stop words**:
  Eliminate common words that carry little meaning (e.g., "is", "the", "and") to focus on important content.
- **Stemming/Lemmatization**:
  Reduce words to their root or base form.
  Example: `"running"` $\rightarrow$ `"run"`
- **Vectorization**:
  Transform words or documents into numerical representations for machine learning models:
  - **Bag of Words**: Counts word occurrences in a document.
  - **TF-IDF (Term Frequency-Inverse Document Frequency)**: Weighs words by importance across documents.
  - **Word2Vec**: Learns dense vector representations capturing word meaning and

What is Vocabulary in NLP?

- **Vocabulary** = All unique words in your dataset
- Example: `"I love NLP and NLP loves me"` $\rightarrow$ Vocabulary = {`"I"`, `"love"`, `"NLP"`, `"and"`, `"loves"`, `"me"`}
- More data = Bigger vocabulary = Harder to process!

Tip: Rare words may not help; common words may not mean much.

The Problem with Sparse Representations

Traditional approach: One-hot encoding

- Example: `"NLP"` $\rightarrow$ [0, 0, 1, 0, 0, 0, 0...]

Problem:

- High-dimensional (thousands of words!)
- Sparse (mostly 0s)
- No meaning in structure (no relation between `"king"` and `"queen"`)
- Not efficient for learning

Feature extraction = Turning text into numbers

- Count how often each word appears (**Term Frequency**)

Example:

Text	`"great product"`	`"bad product"`
Word: `"great"`	1	0
Word: `"bad"`	0	1

Use this to find patterns in sentiment, spam, etc.

Positive & Negative Frequencies

Suppose you're classifying reviews:

Positive Reviews: `["amazing", "good", "great"]`
Negative Reviews: `["bad", "awful", "terrible"]`

Count how often each word appears in each class.

Example table:

Word	Positive Count	Negative Count
good	20	1
bad	1	30

Helps models detect the "tone" (**sentiment clues**) of new text.

Feature Extraction Techniques

- **Bag of Words (BoW)**: Just counts word frequencies in each document.
- **TF-IDF (Term Frequency-Inverse Document Frequency)**: Adjusts for how "unique" or important a word is in a document compared to all documents.
  - Words like `"the"`, `"is"` are less important.
- **Word Embeddings (later)**: Add meaning and capture relationships between words (e.g., similarity, analogy).

N-Gram Models – Capture Word Pairs!

BoW ignores word order: `"not good"` = `"good"`

N-grams fix this:

- **1-gram:** `"I"`, `"love"`, `"cats"`
- **2-gram:** `"I love"`, `"love cats"`

Helps detect phrases:

- `"not good"` ≠ `"very good"`

Bi-grams and tri-grams make models more context-aware!

NLP is all about uncertainty!

We ask: **What is the probability that this sentence is positive?**

Example:

- $P(\text{Positive} \mid \text{“great product”}) = 0.92$
- $P(\text{Negative} \mid \text{“great product”}) = 0.08$

Bayes' Rule:

$$P(\text{Class} \mid \text{Words}) = \frac{P(\text{Words} \mid \text{Class}) \cdot P(\text{Class})}{P(\text{Words})}$$

- Predict class (e.g., spam or not) given the words
- Simple, powerful tool used in Naive Bayes classifiers
- Helps models reason with uncertainty

Simple, fast probabilistic model

Assumes:

- Words are independent given the class (a "naive" assumption)

Example:

- `"This movie is awesome"` $\rightarrow P(\text{Positive})$ high
- `"This movie is boring"` $\rightarrow P(\text{Negative})$ high

Works well with BoW, TF-IDF, and N-grams

Logistic Regression for Text Classification

We're already familiar with probability-based classifiers, and have seen logistic regression applied in computer vision tasks.

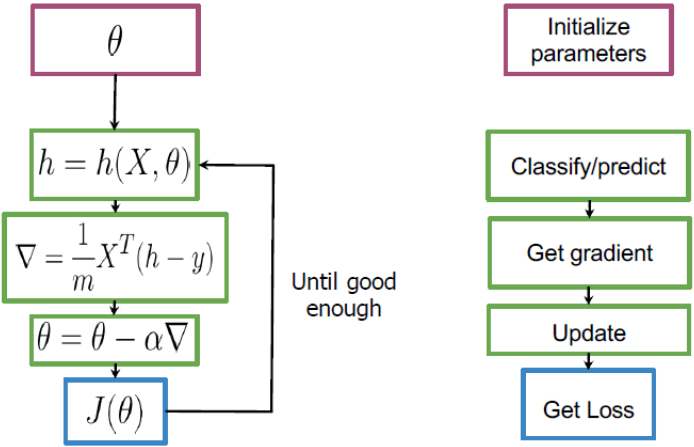
- Here, logistic regression takes word features (from BoW or TF-IDF)
- It learns weights to predict if text belongs to a certain class, such as:
  - Positive or Negative
  - Spam or Not Spam
- Outputs a probability:

$$P(\text{Class} = 1 \mid \text{features}) = \sigma(w_1 x_1 + w_2 x_2 + \ldots + b)$$

- Words like "awesome" get high weights for positive class

Intuition:

- Each word contributes a "vote" toward a class.
- The model learns which words are most important for classification.



Training a Logistic Regression Model


اكاديمية كاوست
KAUST ACADEMY


LMH
Lady Margaret Hall

Word Vectors – Adding Meaning!

Unlike BoW, word vectors (embeddings) capture meaning

Words are mapped to dense vectors in low-dimensional space.

Example: 300-dimensional vector: `"cat"` → [0.1, -0.3, ..., 0.2]

Similar words have similar vectors:

- `"king"` and `"queen"` are close in vector space
- Captures relationships and analogies (e.g., `"man"` : `"woman"` :: `"king"` : `"queen"`)

Word embeddings add context and meaning to NLP models!

Popular Word Vector Models

- **Word2Vec (Google):**
  Trains using context words ("You shall know a word by the company it keeps")
- **GloVe (Stanford):**
  Uses word co-occurrence in large corpus
- **FastText (Facebook):**
  Captures subword info, works better for rare words

Used for initializing deep models like RNNs, Transformers

Recap – NLP Basics

- **Vocabulary and word counts** are key features for representing text.
- **Sparse data** (like one-hot vectors) limits early NLP models.
- **Naive Bayes** and **Logistic Regression** are fast and effective for text classification.
- **N-grams** add word order and context.
- **Word vectors** (embeddings) bring semantic meaning to models.

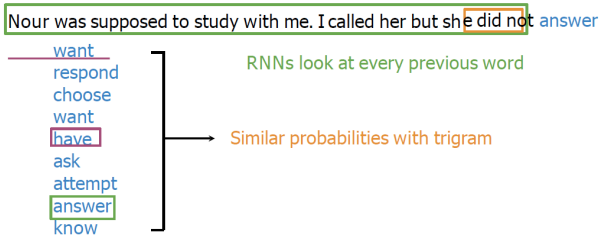
Now, let's dive into neural networks for language...

NLP: **Recurrent Neural Networks (RNNs)**

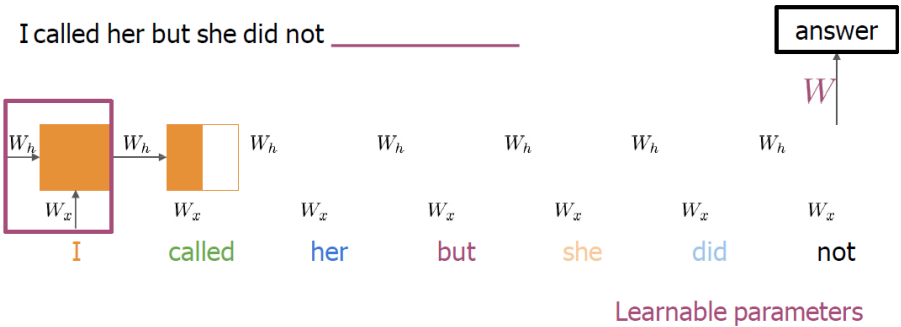
Limitations of Traditional Approaches

- **Bag-of-Words and n-gram models:**
  - Ignore word order and long-range dependencies.
  - Fixed-size context window limits understanding of sequence.
  - Consume more memory and computational resources for larger contexts.
- **Feedforward Neural Networks:**
  - Cannot handle variable-length input sequences.
  - Lack memory of previous inputs.

How RNNs Helps

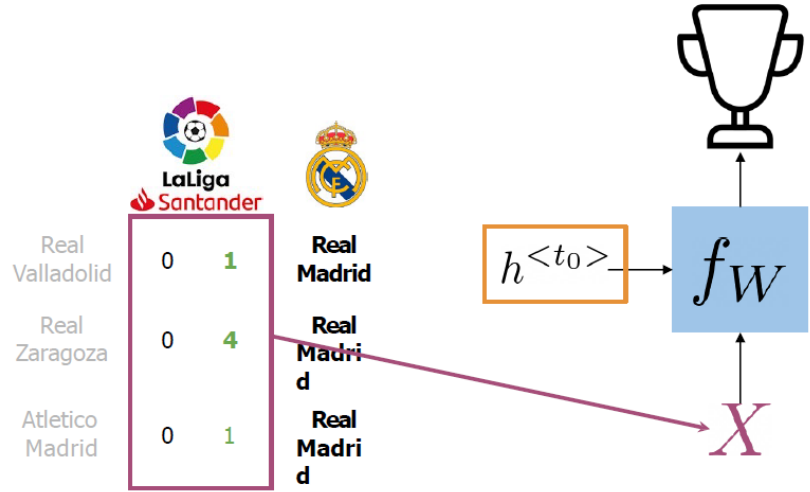


- Designed to process sequential data of arbitrary length.
- Maintain a hidden state to capture context and dependencies over time.
- Better suited for tasks like language modeling, translation, and sequence prediction.

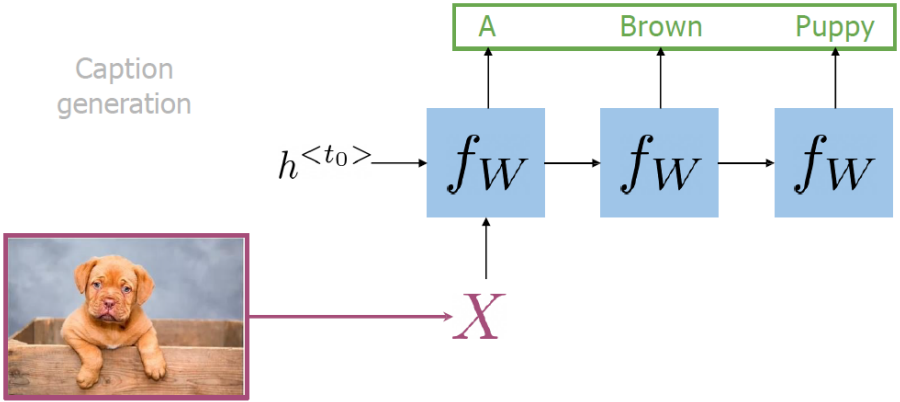


- RNNs model relationships among distant words in a sequence.
- Many computations in RNNs share parameters across time steps, enabling efficient learning and generalization.

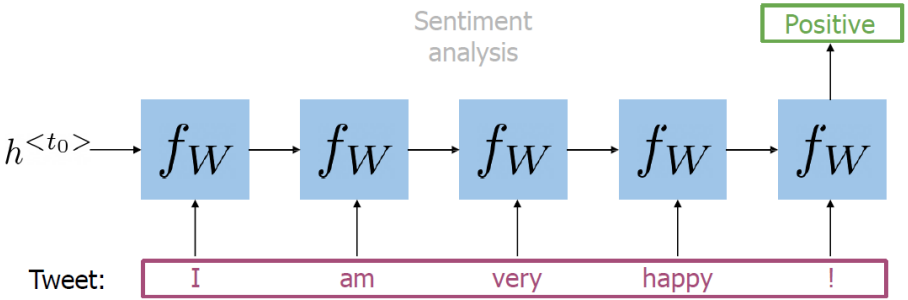
One to One


LaLiga
Santander
Real Valladolid
0
1
Real Madrid
Real Zaragoza
0
4
Real Madrid
Atletico Madrid
0
1
Real Madrid
$h^{<t_0>}$
f_W
X

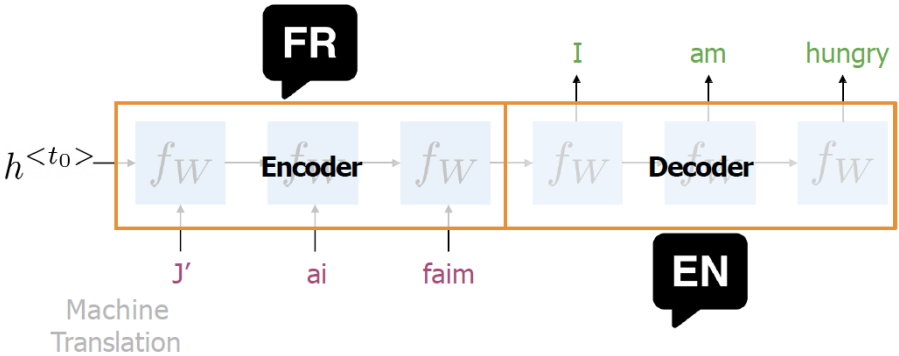
One to Many



Many to One


Sentiment analysis
Positive
$h^{<t_0>}$
fw
fw
fw
fw
fw
Tweet:
I
am
very
happy
!

Many to Many


FR
I
am
hungry
$h^{<t_0>}$
fw
Encoder
fw
fw
fw
Decoder
fw
J'
ai
faim
EN
Machine
Translation

- **Propagation through time:**

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

where x_t is the input at time t, h_{t-1} is the previous hidden state, W_{xh} and W_{hh} are weight matrices, b_h is the bias, and σ is an activation function (e.g., tanh or ReLU).

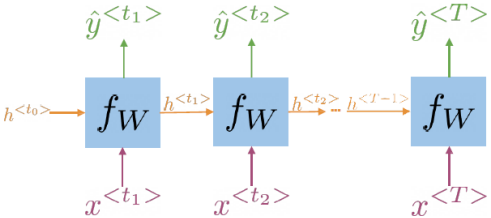
- **Making predictions:**

$$y_t = \phi(W_{hy}h_t + b_y)$$

where W_{hy} is the output weight matrix, b_y is the output bias, and ϕ is an activation function (e.g., softmax for classification).

Key idea: The hidden state h_t acts as memory, allowing information to flow from previous time steps to influence current predictions.

A Vanilla RNN

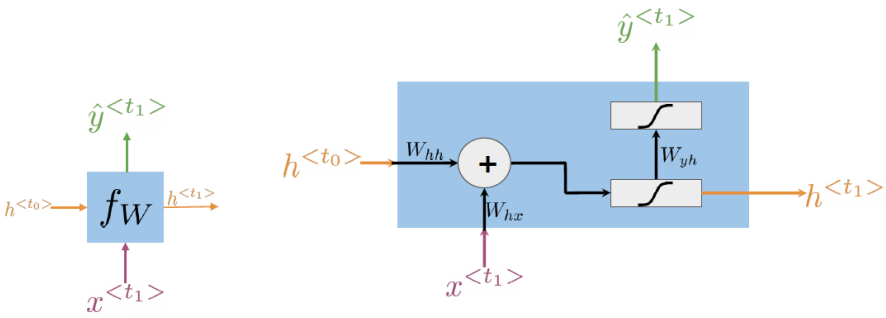


$$h^{<t>} = g(W_h[h^{<t-1>}, x^{<t>}] + b_h)$$

$$\hat{y}^{<t>} = g(W_{yh}h^{<t>} + b_y)$$

$$h^{<t>} = g(W_{hh}h^{<t-1>} + W_{hx}x^{<t>} + b_h)$$

A Vanilla RNN (cont.)



$$h^{<t>} = g(W_{hh}h^{<t-1>} + W_{hx}x^{<t>} + b_h)$$

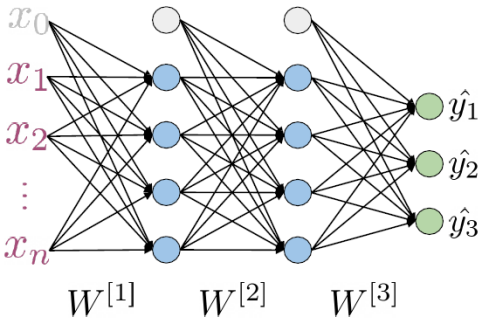
$$\hat{y}^{<t>} = g(W_{yh}h^{<t>} + b_y)$$

- Hidden states propagate information through time.

A Vanilla RNN (cont.)

- Basic recurrent units have two inputs at each time: $x^{\langle t \rangle}$ (current input) and $h^{\langle t-1 \rangle}$ (previous hidden state).

Cross Entropy Loss



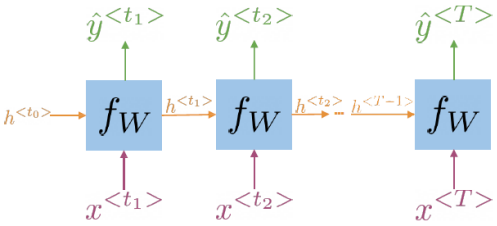
K - classes or possibilities

Either 0 or 1

$$J = -\sum_{j=1}^{K} y_j \log \hat{y}_j$$

Looking at a single example (x, y)

Cross Entropy Loss for RNNs



$$h^{<t>} = g(W_h[h^{<t-1>}, x^{<t>}] + b_h)$$
$$\hat{y}^{<t>} = g(W_{yh}h^{<t>} + b_y)$$

$$J = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{K} y_j^{<t>} \log \hat{y}_j^{<t>}$$

Average with respect to time

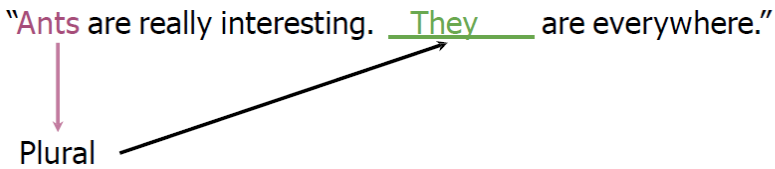
For RNNs the loss function is just an average through time!

NLP: **Gated Recurrent Unit (GRU)**

GRU: Overview

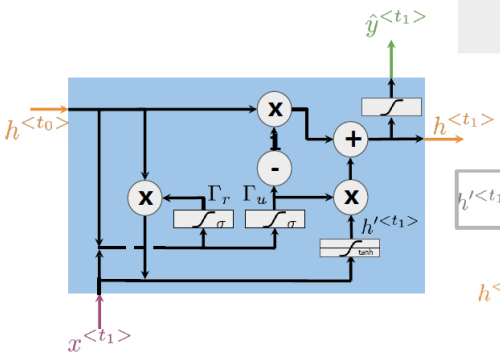
- GRU is a type of RNN designed to handle long-range dependencies.
- Combines the forget and input gates into a single update gate.
- Uses a reset gate to control how much past information to forget.

GRU: Overview (cont.)



Relevance and update gates to remember important prior information

GRU: Overview (cont.)



Gates to keep/update relevant information in the hidden state

$$\Gamma_r = \sigma(W_r[h^{<t_0>}, x^{<t_1>}] + b_r)$$
$$\Gamma_u = \sigma(W_u[h^{<t_0>}, x^{<t_1>}] + b_u)$$

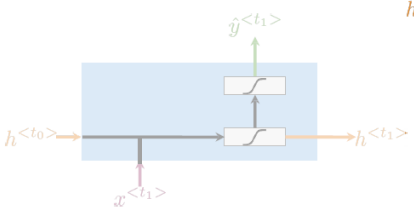
$$h'^{<t_1>} = \tanh\left(W_h[\Gamma_r * h^{<t_0>}, x^{<t_1>}] + b_h\right)$$

Hidden state candidate

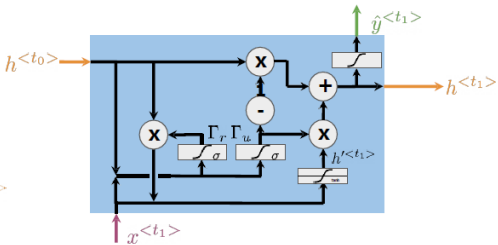
$$h^{<t_1>} = (1 - \Gamma_u) * h^{<t_0>} + \Gamma_u * h'^{<t_1>}$$

$$\hat{y}^{<t_1>} = g(W_y h^{<t_1>} + b_y)$$

Vanilla RNN vsGRUs



$$h^{<t>} = g(W_h[h^{<t-1>}, x^{<t>}] + b_h)$$
$$\hat{y}^{<t>} = g(W_{yh}h^{<t>} + b_y)$$



$$\Gamma_u = \sigma(W_u[h^{<t_0>}, x^{<t_1>}] + b_u)$$
$$\Gamma_r = \sigma(W_r[h^{<t_0>}, x^{<t_1>}] + b_r)$$
$$h'^{<t_1>} = \tanh(W_h[\Gamma_r * h^{<t_0>}, x^{<t_1>}] + b_h)$$
$$h^{<t_1>} = (1 - \Gamma_u) * h^{<t_0>} + \Gamma_u * h'^{<t_1>}$$
$$\hat{y}^{<t_1>} = g(W_y h^{<t_1>} + b_y)$$

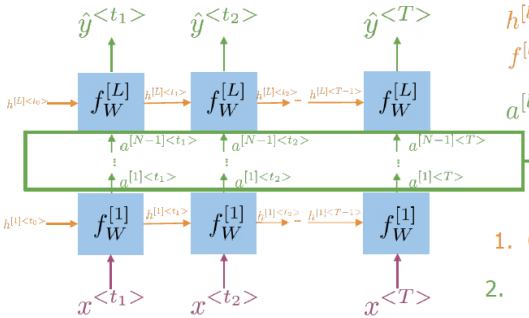
GRU: Key Properties

- GRUs "decide" how to update the hidden state:
  - The update gate controls the degree to which the hidden state is updated with new information.
  - The reset gate determines how much of the previous hidden state to forget.
  - This gating mechanism allows the GRU to adaptively capture dependencies of varying lengths.

GRU: Key Properties (cont.)

- GRUs help preserve important information:
  - By selectively updating and resetting, GRUs can retain relevant information over long sequences.
  - This helps mitigate the vanishing gradient problem common in standard RNNs.
  - As a result, GRUs are effective for tasks requiring memory of long-term context.

Deep RNNs



$$h^{[l]<t>} = f^{[l]}(W_h^{[l]}[h^{[l]<t-1>}, a^{[l-1]<t>}] + b_h^{[l]})$$

$$a^{[l]<t>} = f^{[l]}(W_a^{[l]} h^{[l]<t>} + b_a^{[l]})$$

Intermediate layers and activations

1. Get hidden states for current layer
2. Pass the activations to the next layer

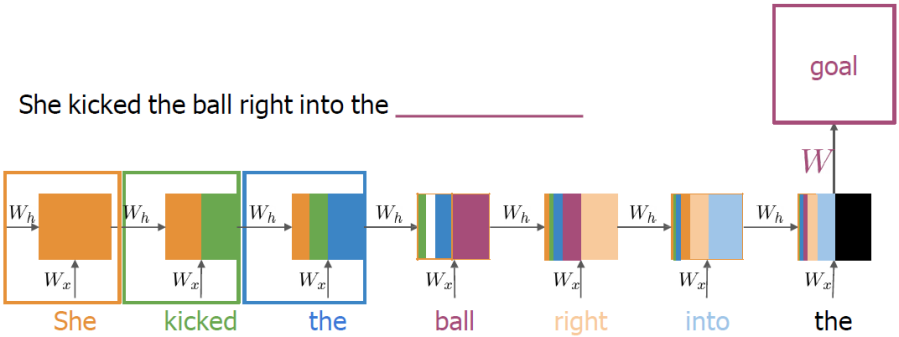
Deep RNNs have more than one layer, which helps in complex tasks.

RNNs: Advantages

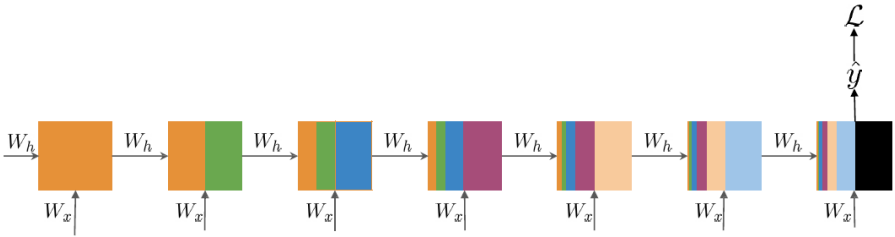
- Captures dependencies within a short range.
- Takes up less RAM than other n-gram models.

RNNs: Disadvantages

- Struggles to capture long-term dependencies.
- Prone to vanishing or exploding gradients.


goal
She kicked the ball right into the _____
W
W_h
W_x
She
kicked
the
ball
right
into
the
Learnable parameters

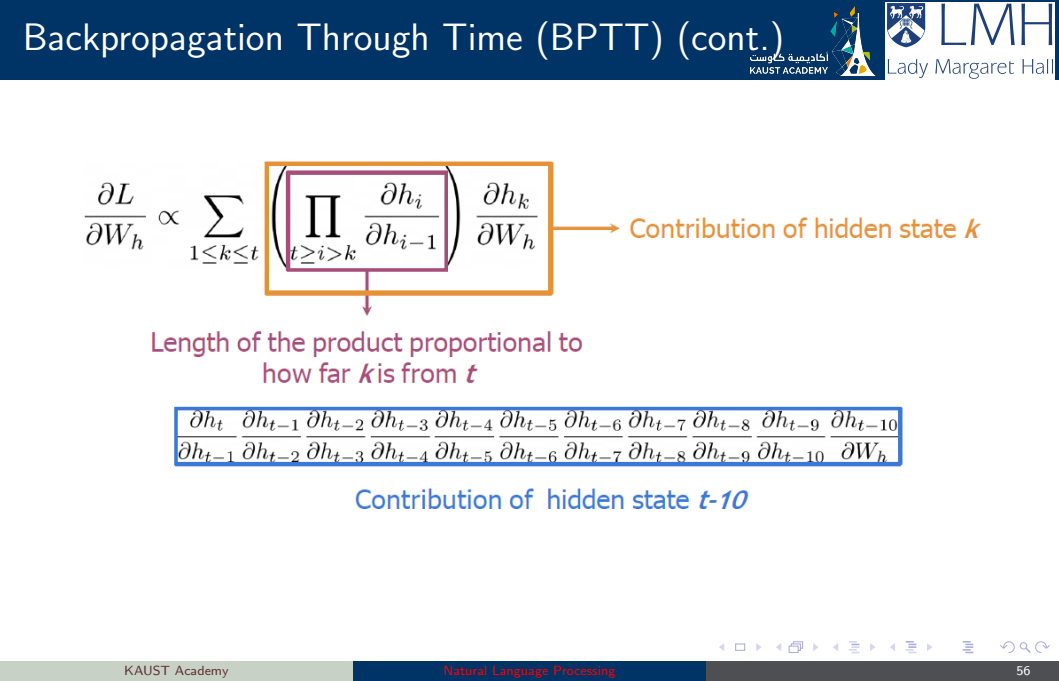
Backpropagation Through Time (BPTT)



W_x
W_h
Same at every step

$$\frac{\partial L}{\partial W_h} \propto \sum_{1 \leq k \leq t} \left(\prod_{t \geq i > k} \frac{\partial h_i}{\partial h_{i-1}} \right) \frac{\partial h_k}{\partial W_h}$$

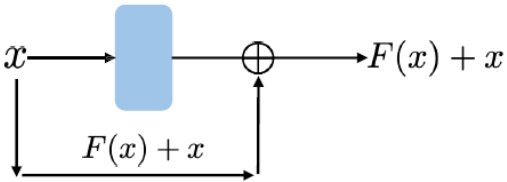
Gradient is proportional to a sum of partial derivative products


$\frac{\partial L}{\partial W_h} \propto \sum_{1 \le k \le t} \left(\prod_{t \ge i > k} \frac{\partial h_i}{\partial h_{i-1}} \right) \frac{\partial h_k}{\partial W_h}$
Contribution of hidden state k
Length of the product proportional to how far k is from t
$\frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \frac{\partial h_{t-2}}{\partial h_{t-3}} \frac{\partial h_{t-3}}{\partial h_{t-4}} \frac{\partial h_{t-4}}{\partial h_{t-5}} \frac{\partial h_{t-5}}{\partial h_{t-6}} \frac{\partial h_{t-6}}{\partial h_{t-7}} \frac{\partial h_{t-7}}{\partial h_{t-8}} \frac{\partial h_{t-8}}{\partial h_{t-9}} \frac{\partial h_{t-9}}{\partial h_{t-10}} \frac{\partial h_{t-10}}{\partial W_h}$
Contribution of hidden state t-10

- Identity RNN with ReLU activation

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Gradient clipping
- Skip connections

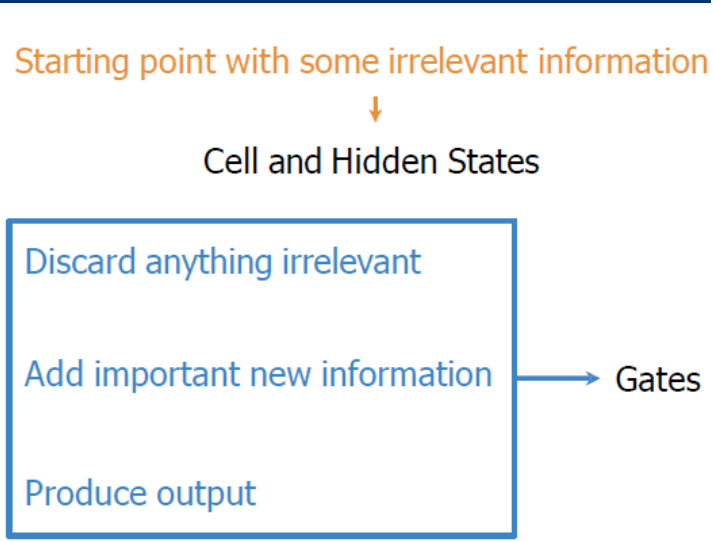


NLP: **Long Short-Term Memory (LSTM)**

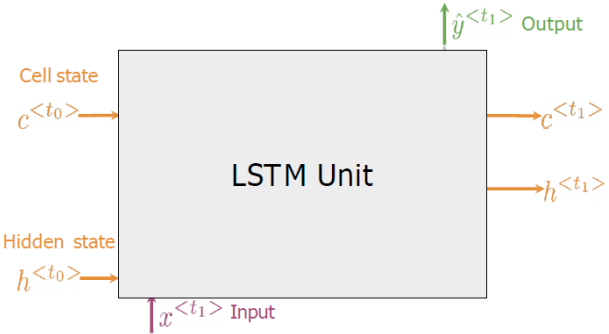
LSTM: Overview

- LSTM is a type of RNN designed to overcome the vanishing gradient problem.
- It uses a memory cell to maintain information over long periods.
- Learns when to remember and when to forget.
- **Basic anatomy:**
  - A cell state
  - A hidden state
  - Multiple gates
- LSTM has three gates: input gate, forget gate, and output gate.
- Gates allow gradients to avoid vanishing and exploding.

LSTMs: Based on Previous Understanding



Gates in LSTM



1.Forget Gate: information that is no longer important

2. Input Gate: information to be stored

3.Output Gate: information to use at current step

Applications of LSTMs

Next-character prediction

Chatbots

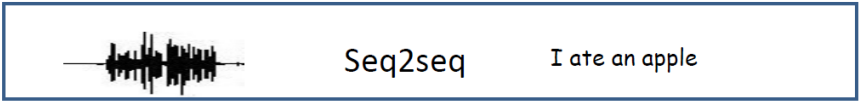
Music composition

Image captioning

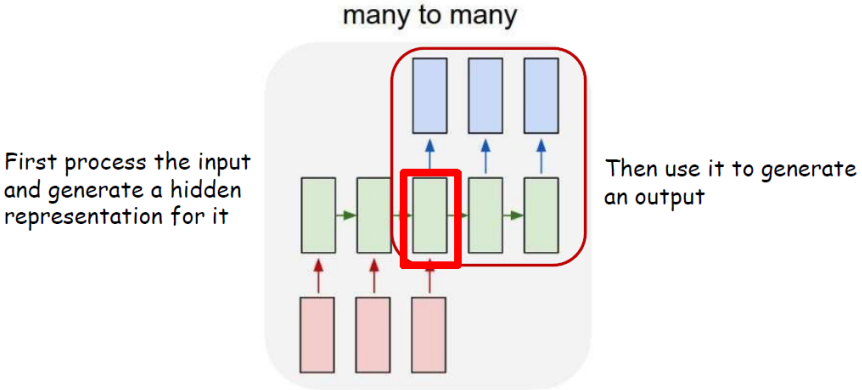
Speech recognition

NLP: **Attention Mechanism**

Sequence to Sequence Learning

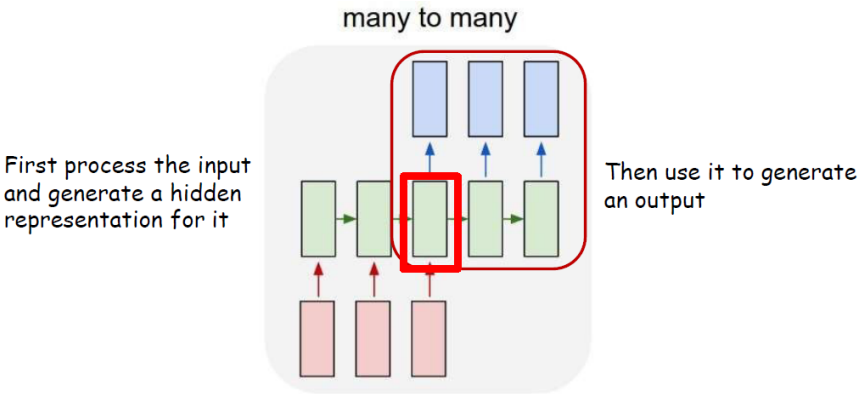


- Sequence goes in, sequence comes out
- No notion of "time synchrony" between input and output
  - May even not even maintain order of symbols
    - E.g. "I ate an apple" → "Ich habe einen apfel gegessen"
  - Or even seem related to the input
    - E.g. "My screen is blank" → "Please check if your computer is plugged in."



Delayed Sequence to Sequence Problem:

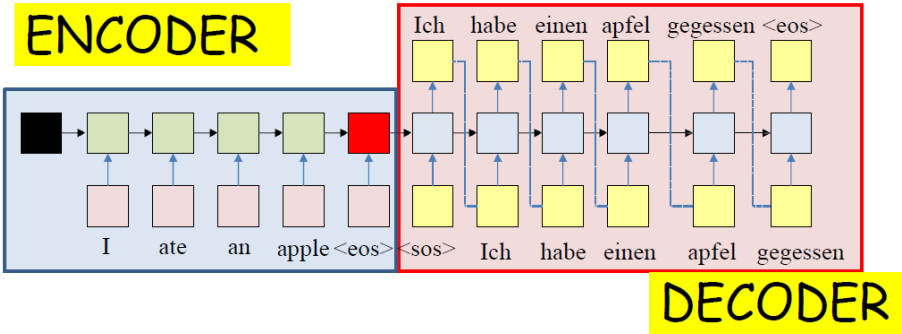
Problem: Each word that is output depends only on the current hidden state, and not on previous outputs.



Delayed Sequence to Sequence Problem:

Delayed *self-referencing* sequence-to-sequence: Each word that is output depends on the current hidden state, and also on previous outputs.

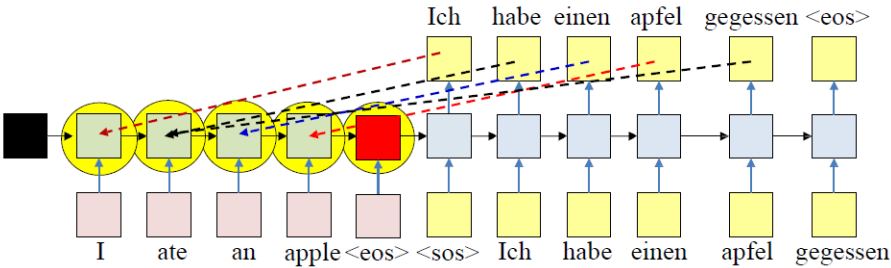
The "simple" translation model



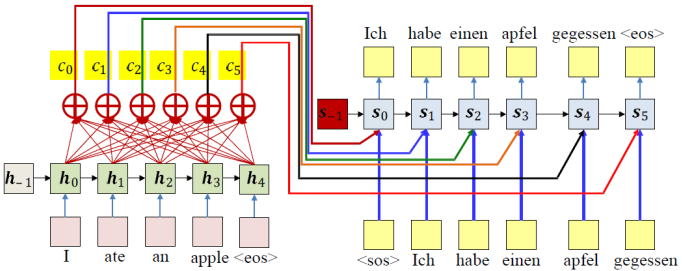
The recurrent structure that:

- extracts the hidden representation from the input sequence is the **encoder**.
- utilizes this representation to produce the output sequence is the **decoder**.

Problems with the "simple" translation model

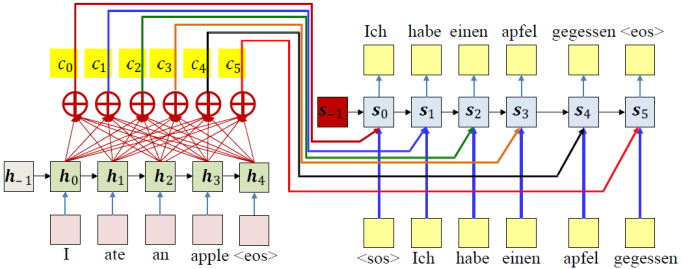


- The model has to compress all the information from the input sequence into a single fixed-size vector.
- This can lead to loss of important information, especially for long sequences.
- The model struggles with long-range dependencies and context retention.



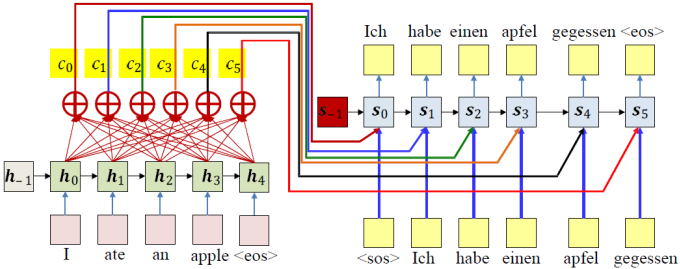
$$c_t = \frac{1}{N}\sum_{i}^{N} w_i(t) h_i$$

- **Attention weights:** The weights $w_i(t)$ are dynamically computed as functions of the decoder state and the encoder outputs.



$$c_t = \frac{1}{N}\sum_{i}^{N} w_i(t) h_i$$

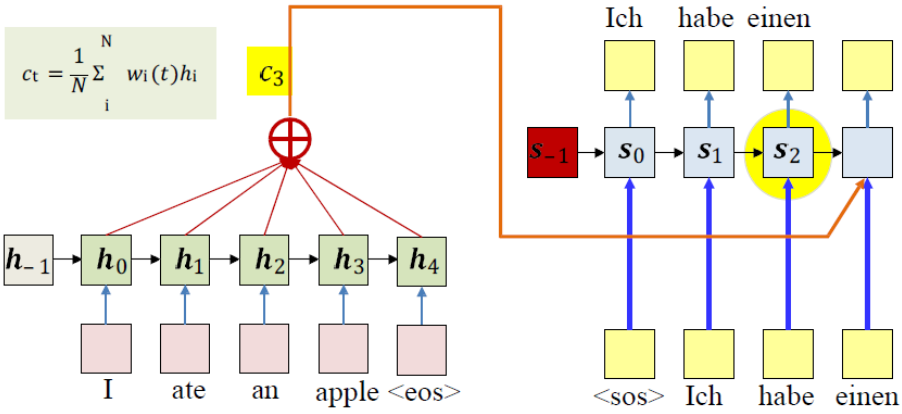
- **Attention weights:** The weights $w_i(t)$ are dynamically computed as functions of the decoder state and the encoder outputs.
- **Intuition:** If the model is well-trained, these weights will automatically "highlight" the most relevant parts of the input sequence for each output step.



$$c_t = \frac{1}{N}\sum_{i}^{N} w_i(t) h_i$$

- **Attention weights:** The weights $w_i(t)$ are dynamically computed as functions of the decoder state and the encoder outputs.
- **Intuition:** If the model is well-trained, these weights will automatically "highlight" the most relevant parts of the input sequence for each output step.
- **How are they computed?**

Attention Weights at time t



Attention Weights at time t

- **What do attention weights do?** They tell the model which input words to pay more attention to when making each output word.

Attention Weights at time t

- **What do attention weights do?** They tell the model which input words to pay more attention to when making each output word.
- The **attention** weight $w_i(t)$ determines how much focus the model places on the i-th input position when producing the output at time t.

Attention Weights at time t

- **What do attention weights do?** They tell the model which input words to pay more attention to when making each output word.
- The **attention** weight $w_i(t)$ determines how much focus the model places on the i-th input position when producing the output at time t.
- The primary information used to compute $w_i(t)$ is the encoder hidden state h_i (at input position i) and the decoder state s_{t-1} (at the previous output time step).

Attention Weights at time t

- **What do attention weights do?** They tell the model which input words to pay more attention to when making each output word.
- The **attention** weight $w_i(t)$ determines how much focus the model places on the i-th input position when producing the output at time t.
- The primary information used to compute $w_i(t)$ is the encoder hidden state h_i (at input position i) and the decoder state s_{t-1} (at the previous output time step).
- The attention score is typically computed as:

$$w_i(t) = a(h_i, s_{t-1})$$

where $a(\cdot)$ is a function (such as a dot product, additive, or other scoring function).

Attention Weights at time t

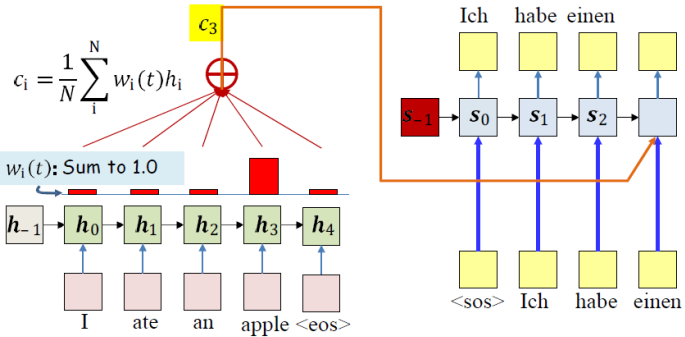
- **What do attention weights do?** They tell the model which input words to pay more attention to when making each output word.
- The **attention** weight $w_i(t)$ determines how much focus the model places on the i-th input position when producing the output at time t.
- The primary information used to compute $w_i(t)$ is the encoder hidden state h_i (at input position i) and the decoder state s_{t-1} (at the previous output time step).
- The attention score is typically computed as:

$$w_i(t) = a(h_i, s_{t-1})$$

where $a(\cdot)$ is a function (such as a dot product, additive, or other scoring function).

- Sometimes, the input word at time t is also used, but often omitted for simplicity.

Requirement on attention weights



- The attention weights $w_i(t)$ must satisfy the following properties:
  - They are non-negative: $w_i(t) \geq 0$ for all i.
  - They sum to 1: $\sum_{i=1}^{n} w_i(t) = 1$, where n is the length of the input sequence.

Requirement on attention weights (cont.)

- These properties ensure that the attention weights can be interpreted as probabilities, allowing the model to focus on different parts of the input sequence.
- The non-negativity ensures that the model does not "ignore" any part of the input sequence, while the summation to 1 ensures that the model can distribute its attention across the input sequence.

Two-step computation of attention weights

1. **Compute raw attention scores:** For each input position i, compute a score $e_i(t)$ (which can be positive or negative) using a scoring function:

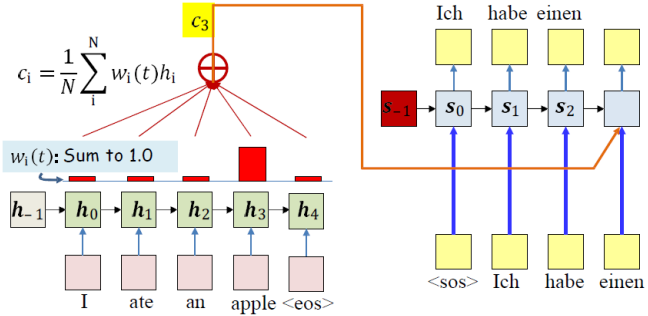
$$e_i(t) = a(h_i, s_{t-1})$$

2. **Normalize with softmax:** Convert the raw scores into a probability distribution using the softmax function:

$$w_i(t) = \frac{\exp(e_i(t))}{\sum_{j=1}^{n} \exp(e_j(t))}$$

This ensures all $w_i(t) \geq 0$ and $\sum_{i=1}^{n} w_i(t) = 1$.

Attention Weights at time t



- Typical options for $g()$ (**variables in red must learned**)

$$g(\boldsymbol{h}_i, \boldsymbol{s}_{t-1}) = \boldsymbol{h}_i^T \boldsymbol{s}_{t-1}$$

$$g(\boldsymbol{h}_i, \boldsymbol{s}_{t-1}) = \boldsymbol{h}_i^T \boldsymbol{W}_g \boldsymbol{s}_{t-1}$$

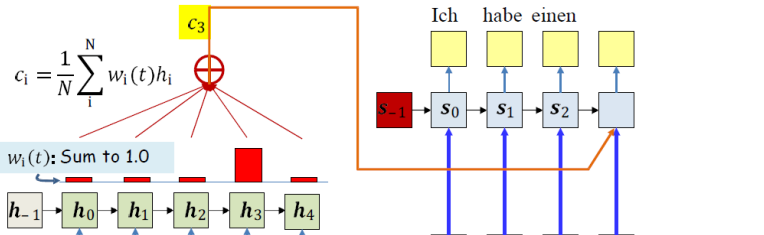
$$g(\boldsymbol{h}_i, \boldsymbol{s}_{t-1}) = \boldsymbol{v}_g^T \boldsymbol{tanh}\left(\boldsymbol{W}_g \begin{bmatrix} \boldsymbol{h}_i \\ \boldsymbol{s}_{t-1} \end{bmatrix}\right)$$

$$g(\boldsymbol{h}_i, \boldsymbol{s}_{t-1}) = MLP([\boldsymbol{h}_i, \boldsymbol{s}_{t-1}])$$

$$e_i(t) = g(\boldsymbol{h}_i, \boldsymbol{s}_{t-1})$$

$$w_i(t) = \frac{\exp(e_i(t))}{\sum_j \exp(e_j(t))}$$

Attention Weights at time t (cont.)



- Typical options for $g()$ (**variables in red must learned**)

$$g(\boldsymbol{h}_i, \boldsymbol{s}_{t-1}) = \boldsymbol{h}_i^T \boldsymbol{s}_{t-1}$$

$$g(\boldsymbol{h}_i, \boldsymbol{s}_{t-1}) = \boldsymbol{h}_i^T \boldsymbol{W}_g \boldsymbol{s}_{t-1}$$

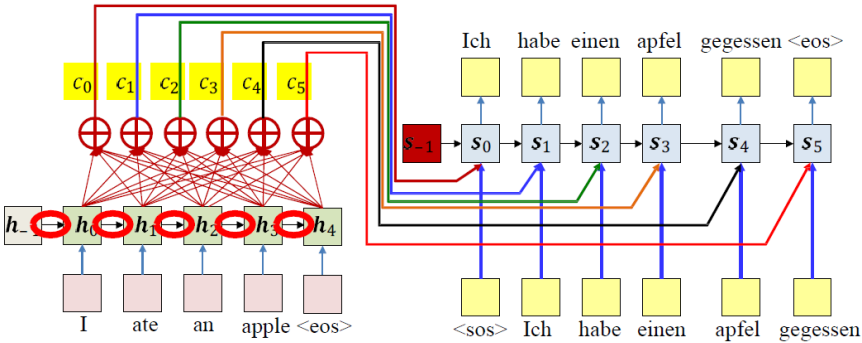
$$g(\boldsymbol{h}_i, \boldsymbol{s}_{t-1}) = \boldsymbol{v}_g^T \boldsymbol{tanh}\left(\boldsymbol{W}_g \begin{bmatrix} \boldsymbol{h}_i \\ \boldsymbol{s}_{t-1} \end{bmatrix}\right)$$

$$g(\boldsymbol{h}_i, \boldsymbol{s}_{t-1}) = \boldsymbol{MLP}([\boldsymbol{h}_i, \boldsymbol{s}_{t-1}])$$

$$e_i(t) = g(\boldsymbol{h}_i, \boldsymbol{s}_{t-1})$$

$$w_i(t) = \frac{\exp(e_i(t))}{\sum_j \exp(e_j(t))}$$

Summary: Attention Models



- **Problem**: The decoder's hidden representation for each output word is influenced by *all* preceding words due to recurrence.
- The decoder effectively pays attention to the entire input sequence, not just a single word.
- If the decoder can automatically determine which input words to focus on at each step, recurrence in the input may not be strictly necessary.

NLP: **Transformers**

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

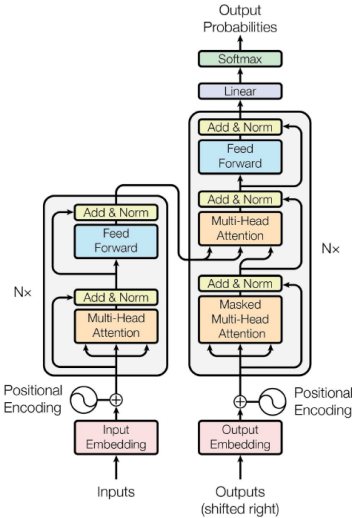
Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Transformers: Overview (cont.)

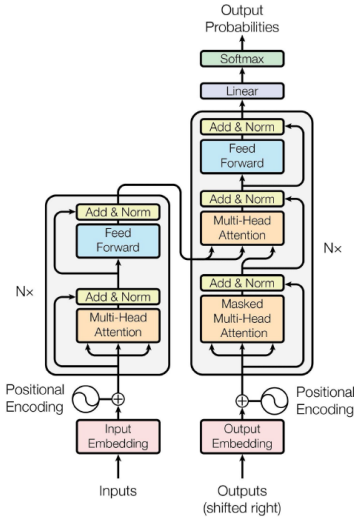
Key Features:

- Transformers revolutionized NLP by using self-attention mechanisms.
- They process entire sequences simultaneously, unlike RNNs.
- Key components:
  - Multi-head self-attention
  - Positional encoding
  - Feed-forward neural networks
- Transformers can handle long-range dependencies effectively.



Targets

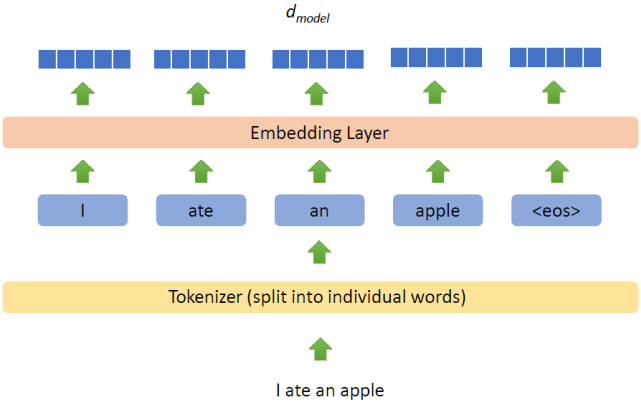
Ich habe einen Apfel gegessen

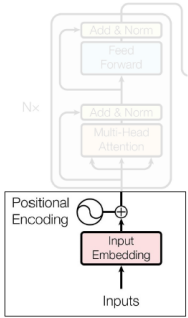


Inputs

I ate an apple

Input Embeddings

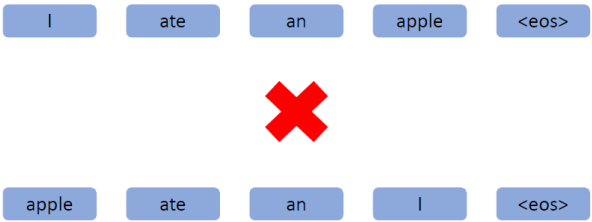


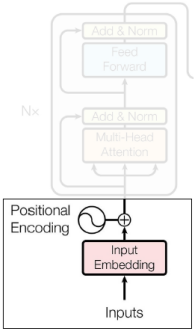


Input embeddings are dense vector representations of input tokens, allowing the model to understand semantic relationships.

Position encodings are added to input embeddings to provide information about the position of tokens in the sequence, enabling the model to capture order and structure.

Position Encodings (cont.)





Position Encodings (cont.)

Requirements for Positional Encodings

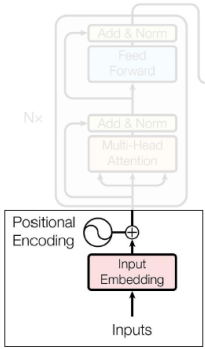
- Some representation of time? (like **seq2seq**?)
- Should be unique for each position – not cyclic
- **Bounded**

Possible Candidates

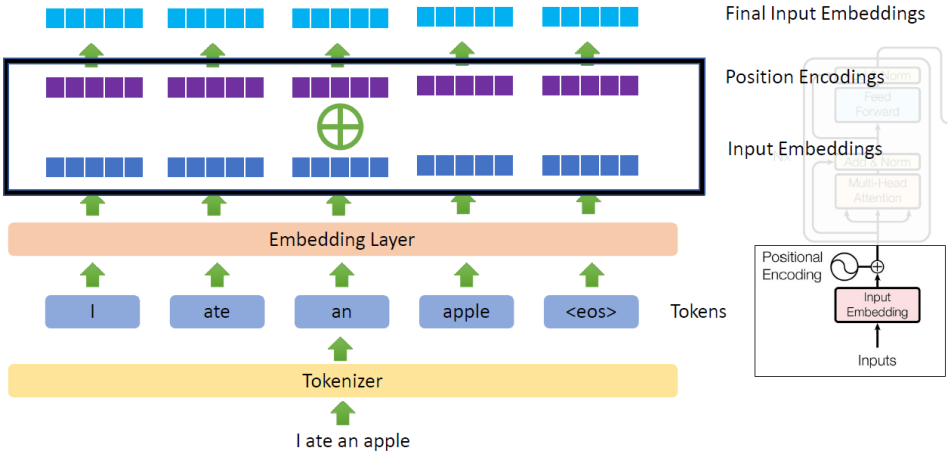
$$P(t + t') = M^{t'} \times P(t)$$

M?

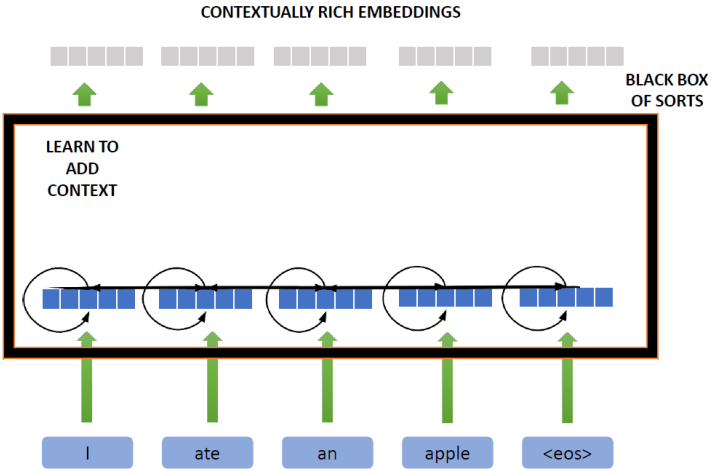
1. **Should be a unitary matrix**
2. **Magnitudes of eigen value should be 1 -> norm preserving**
3. **The matrix can be learnt**
4. **Produces unique rotated embeddings each time**

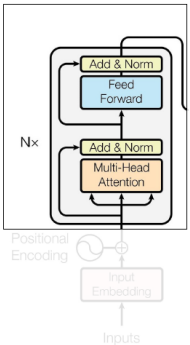


Position Encodings (cont.)



Encoder

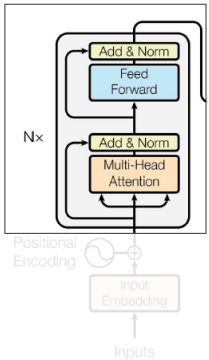




Encoder processes the input sequence, generating a set of continuous

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

- Query
- Key
- Value



Query, Key, and Value

{Query: “Order details of order_104”}

OR

{Query: “Order details of order_106”}

{Key, Value store}


```
{"order_100": {"items":"a1", "delivery_date":"a2", ...}},
{"order_101": {"items":"b1", "delivery_date":"b2", ...}},
{"order_102": {"items":"c1", "delivery_date":"c2", ...}},
{"order_103": {"items":"d1", "delivery_date":"d2", ...}},
{"order_104": {"items":"e1", "delivery_date":"e2", ...}},
{"order_105": {"items":"f1", "delivery_date":"f2", ...}},
{"order_106": {"items":"g1", "delivery_date":"g2", ...}},
{"order_107": {"items":"h1", "delivery_date":"h2", ...}},
{"order_108": {"items":"i1", "delivery_date":"i2", ...}},
{"order_109": {"items":"j1", "delivery_date":"j2", ...}},
{"order_110": {"items":"k1", "delivery_date":"k2", ...}}
```

Query, Key, and Value (cont.)

{Query: “Order details of order_104”}

OR

{Query: “Order details of order_106”}

{Key, Value store}

```
{"order_100": {"items":"a1", "delivery_date":"a2", ...}},
{"order_101": {"items":"b1", "delivery_date":"b2", ...}},
{"order_102": {"items":"c1", "delivery_date":"c2", ...}},
{"order_103": {"items":"d1", "delivery_date":"d2", ...}},
{"order_104": {"items":"e1", "delivery_date":"e2", ...}},
{"order_105": {"items":"f1", "delivery_date":"f2", ...}},
{"order_106": {"items":"g1", "delivery_date":"g2", ...}},
{"order_107": {"items":"h1", "delivery_date":"h2", ...}},
{"order_108": {"items":"i1", "delivery_date":"i2", ...}},
{"order_109": {"items":"j1", "delivery_date":"j2", ...}},
{"order_110": {"items":"k1", "delivery_date":"k2", ...}}
```

Query, Key, and Value (cont.)

Done at the same time !!

{Query: "Order details of order_104"}

OR

{Query: "Order details of order_106"}

{Key, Value store}

```
{"order_100": {"items":"a1", "delivery_date":"a2", ...}},
{"order_101": {"items":"b1", "delivery_date":"b2", ...}},
{"order_102": {"items":"c1", "delivery_date":"c2", ...}},
{"order_103": {"items":"d1", "delivery_date":"d2", ...}},
{"order_104": {"items":"e1", "delivery_date":"e2", ...}},
{"order_105": {"items":"f1", "delivery_date":"f2", ...}},
{"order_106": {"items":"g1", "delivery_date":"g2", ...}},
{"order_107": {"items":"h1", "delivery_date":"h2", ...}},
{"order_108": {"items":"i1", "delivery_date":"i2", ...}},
{"order_109": {"items":"j1", "delivery_date":"j2", ...}},
{"order_110": {"items":"k1", "delivery_date":"k2", ...}}
```

Query, Key, and Value (cont.)

{Query: "Order details of order_104"}

OR

{Query: "Order details of order_106"}

```
{"order_100": {"items":"a1", "delivery_date":"a2", ...}},
{"order_101": {"items":"b1", "delivery_date":"b2", ...}},
{"order_102": {"items":"c1", "delivery_date":"c2", ...}},
{"order_103": {"items":"d1", "delivery_date":"d2", ...}},
{"order_104": {"items":"e1", "delivery_date":"e2", ...}},
{"order_105": {"items":"f1", "delivery_date":"f2", ...}},
{"order_106": {"items":"g1", "delivery_date":"g2", ...}},
{"order_107": {"items":"h1", "delivery_date":"h2", ...}},
{"order_108": {"items":"i1", "delivery_date":"i2", ...}},
{"order_109": {"items":"j1", "delivery_date":"j2", ...}},
{"order_110": {"items":"k1", "delivery_date":"k2", ...}}
```

Query

1. Search for info

Key

1. Interacts directly with Queries
2. Distinguishes one object from another
3. Identify which object is the most relevant and by how much

Value

1. Actual details of the object
2. More fine grained

Transformers

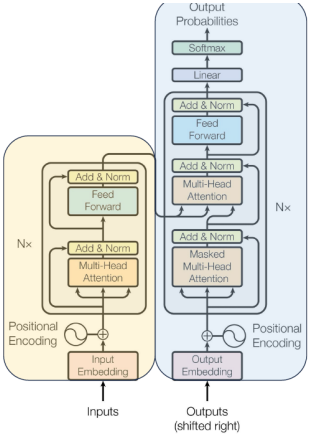
Input – input tokens
Output – hidden states

Model can see all timesteps

Does not usually output tokens, so no inherent auto-regressivity

Can also be adapted to generate tokens *by appending a module that maps hidden state dimensionality to vocab size*

Representation



Input – output tokens and hidden states*
Output – output tokens

Model can only see previous timesteps

Model is auto-regressive with previous timesteps' outputs

Can also be adapted to generate hidden states *by looking before token outputs*

Generation

1/

NLP: **Limitations**

Limitations

NLP has come a long way, but there are still some big challenges:

- **Bias:** Models can pick up and even increase unfairness from the data they learn from.
- **Needs Lots of Data:** Modern NLP needs huge amounts of labeled text, which is hard and expensive to get.
- **Ambiguity:** Language can be confusing! For example, "I saw the man with the telescope"—who has the telescope?
- **Many Languages:** It's tough to make models that work well for every language, especially those with little data.
- **Expensive to Train:** Training big models costs a lot of money and energy, so not everyone can do it.

NLP: **References**

References

[1] Jurafsky, D., Martin, J. H. (2023). *Speech and Language Processing* (3rd ed.). Draft chapters available at https://web.stanford.edu/~jurafsky/slp3/

[2] Mourri, Y., Kaiser, Ł. Natural Language Processing Specialization. DeepLearning.AI.

[3] Raj, B., Singh, R. 11-785 Introduction to Deep Learning. Carnegie Mellon University (CMU).

[4] Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14(2), 179-211.

[5] Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. *EMNLP 2014*.

References (cont.)

[6] Hochreiter, S., Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8), 1735-1780.

[7] Bahdanau, D., Cho, K., Bengio, Y. (2015). Neural Machine Translation by Jointly Learning to Align and Translate. *ICLR 2015*.

[8] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., Polosukhin, I. (2017). Attention Is All You Need. *NeurIPS 2017*.

Credits

Dr. Prashant Aparajeya

Computer Vision Scientist — Director(AISimply Ltd)

p.aparajeya@aisimply.uk

This project benefited from external collaboration, and we acknowledge their contribution with gratitude.