# Distributed Optimization for Machine Learning

## Lecture 19 - Transformers: Architecture, Parameters, and Memories


Tianyi Chen

School of Electrical and Computer Engineering
Cornell Tech, Cornell University


November 10, 2025

# Course project summary: Major themes

**Top 4 Focus areas:**

1. **LLM Parameter-Efficient Fine-Tuning (PEFT)**
2. **Federated/decentralized optimization**
3. **System & protocol robustness**
4. **Specialized topics** (e.g., Quantum FL, ADMM)

---

- **Dominant topic: Low-Rank Adaptation (LoRA)**
  - Why full fine-tuning is impractical and how LoRA addresses it.
- **Related focus:** Quantization in LoRA; memory-latency tradeoffs.

# Theme 2: Federated and decentralized optimization

**Core challenge:** Convergence, communication, and heterogeneity in distributed learning.

- **Focus:** Adaptive consensus under data heterogeneity.
- **Algorithms:** Comparing Local SGD vs. Mini-Batch SGD.
- **Theory:** Invertibility and injectivity in distributed solutions.

- **Focus:** Hierarchical FL and parallelism.
- **Adaptation:** Adaptive and learned aggregation under non-IID data.
- **Application (Research):** Federated transfer learning for **drug discovery**.
- **System:** Privacy-preserving FL.

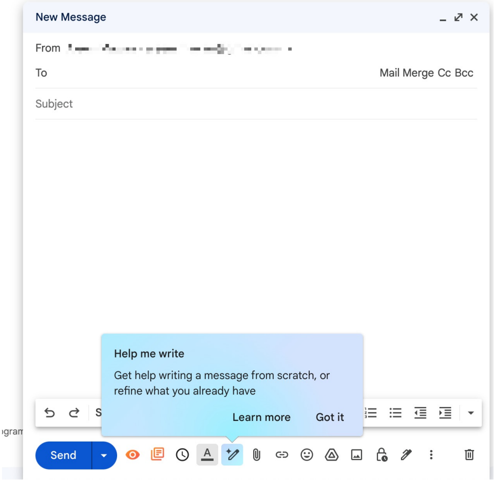
---

**Decentralized LLM Projects (Research/Educational):** Theoretical and practical foundations of distributed optimization for LLMs.

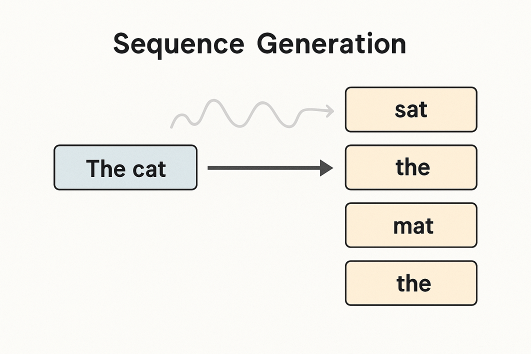
# Themes 3 & 4: Specialized topics

- **Byzantine resilience:** Detection of **Traitors** in distributed optimization.
- **System reliability:** Making distributed training reproducible via **consensus protocols**.
- **Optimization tools:** Understanding **ZeRO** (Zero Redundancy Optimizer).
- **LLM serving:** Queueing for efficient LLM serving.
- **Quantum FL:** Introduction to quantum federated learning.
- **ADMM/GNNs:** Using ADMM for distributed training of graph neural networks.
- **Hardware (Research):** In-memory training on analog devices.
- **Multi-objective (Research):** Tuning LLMs to balance criteria using parallel implementation.

# Generative AI in everyday life



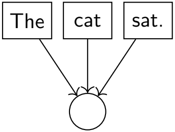
# Many of the tasks involve sequence generation



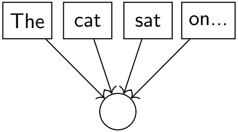
# Table of Contents

# Motivations of sequential modeling

Consider the sequential data and why standard MLPs are a poor fit.




Requires 3 inputs

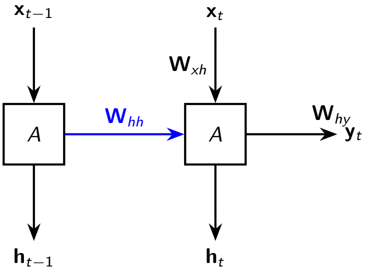



Requires a different architecture!

- **Variable length data:**
  MLPs require a fixed-size input vector, but sequential data like sentences can be any length.
- **No sense of order:**
  It has no built-in notion that "cat" comes after "The," losing crucial contextual information.
- **No parameter sharing:**
  The weights learned for the first word are separate from the weights for the third word.

# Revisit recurrent neural networks (RNNs)

**RNNs** process sequences by maintaining a hidden state $\mathbf{h}_t$ that acts as a memory, passed from one time step to the next.



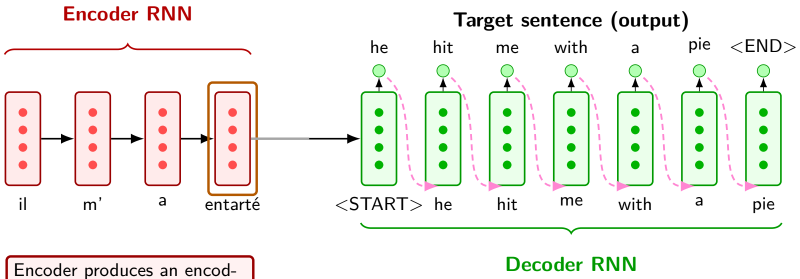

**Model** $h_\theta(\mathbf{x}_1, \ldots, \mathbf{x}_T)$**:** recurrent update for hidden state $\mathbf{h}_t$, output $\mathbf{y}_t$:

$$\mathbf{h}_t = \sigma(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h)$$
$$\mathbf{y}_t = \sigma(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y)$$

**Parameters** $\theta$**:** The shared $\{\mathbf{W}_{hh}, \mathbf{W}_{xh}, \mathbf{W}_{hy}\}$ and biases at every step.

# Sequence-to-sequence (Seq2Seq) model

## Seq2Seq first appears in Machine Translation (MT)




Encoder produces an encoding of the source sentence.

Decoder RNN is a Language model that generates the target sentence, conditioned on encoding.

*Test time:* decoder output is fed as next step's input.

# Sequence-to-sequence (Seq2Seq) is versatile!

**Seq2Seq is useful for more than just MT today! Many genAI tasks**

**can be phrased as sequence-to-sequence:**

- **Summarization:** Long text → short text (summary).
- **Dialogue:** Previous utterances → next utterance (response).
- **Parsing:** Input text → output parse as sequence (structured data).
- **Code generation:** Natural language → Python code (or other programming language).

# Seq2Seq as a conditional language model

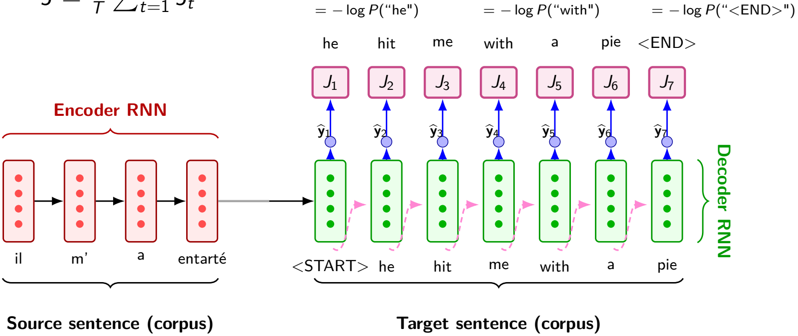
- The **sequence-to-sequence model** is an example of a **Conditional Language Model**
  - **Language Model** because the decoder is predicting the next word of the target sentence $\mathbf{y}$
  - **Conditional** because its predictions are *also* conditioned on the source sentence $\mathbf{x}$

- NMT (Neural Machine Translation) directly calculates the probability of the next word:

$$\underbrace{P(y_T | y_1, \ldots, y_{T-1}, \mathbf{x})}_{\text{Probability of next target word, given target words so far and source sentence } \mathbf{x}}$$

# Training a neural machine translation system

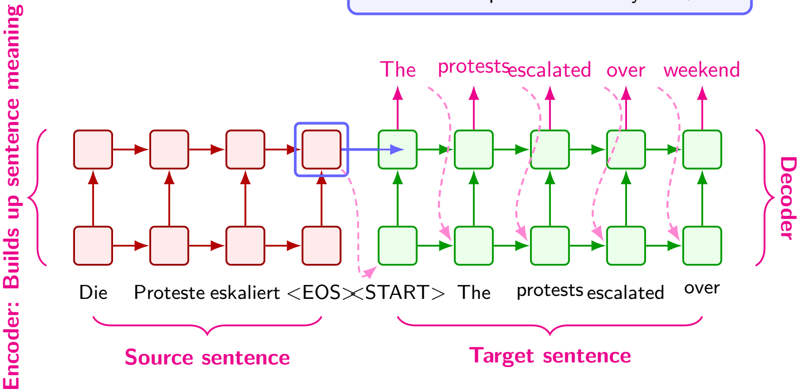
$$J = \frac{1}{T}\sum_{t=1}^{T} J_t$$




Seq2seq is optimized as a **single system**:
backpropagation flows **end-to-end across encoder and decoder.**

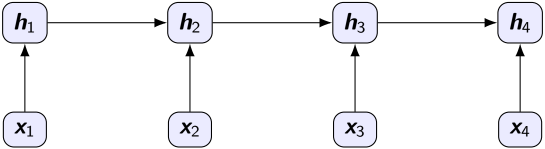
# Multi-layer deep encoder-decoder machine translation net

The hidden states from RNN layer $i$ are the inputs to RNN layer $i+1$



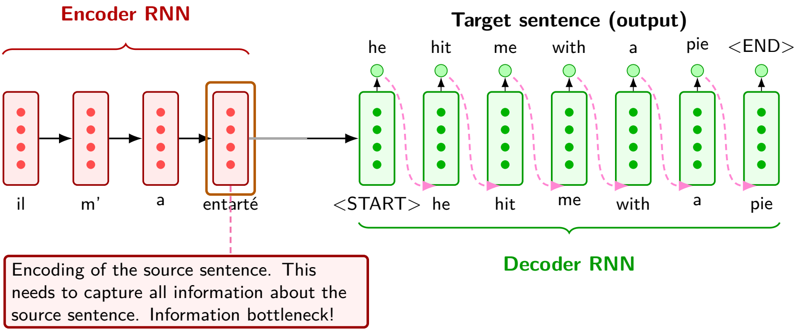
# The evolution of sequence models

- **RNNs and Long Short-Term Memory networks (LSTMs):** process tokens left-to-right; hidden state carries context.
- Strengths:
  - Naturally handle variable-length sequences.
  - Good inductive bias for local, temporal dependencies.
- Limitations:
  - The last block captures all the information about the source.
  - Long-range dependencies degrade (vanishing/exploding gradients).
  - *Sequential* dependency $\Rightarrow$ poor parallelism on modern hardware.




Hidden state must flow sequentially $\Rightarrow$ limited parallelism.

# The information bottleneck problem in RNNs

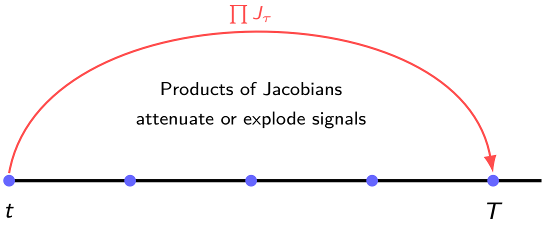


# Why long-range fades in RNNs?

- Backprop through time multiplies many Jacobians:

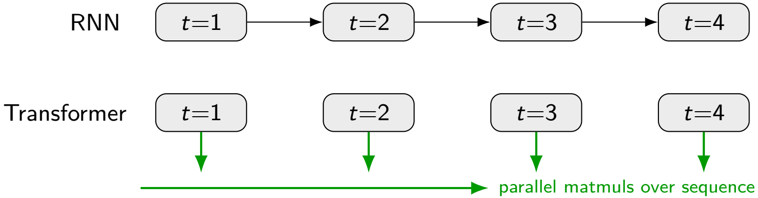
$$\frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_t} = \prod_{\tau=t}^{T-1} \frac{\partial \boldsymbol{h}_{\tau+1}}{\partial \boldsymbol{h}_\tau}.$$

- Eigenvalues $< 1 \Rightarrow$ *vanishing*; $> 1 \Rightarrow$ *exploding*.



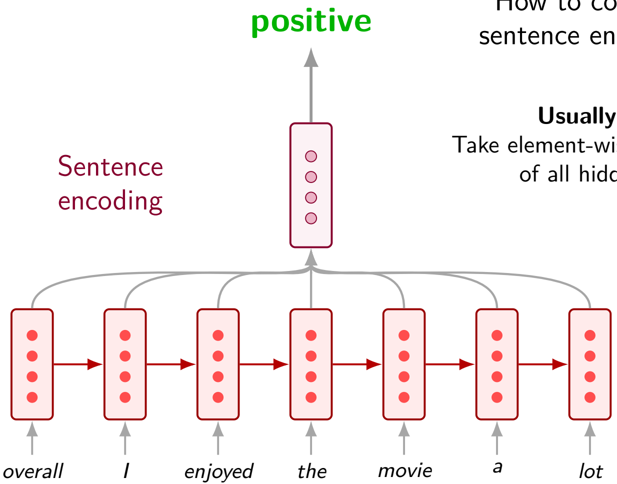
# Parallelism bottleneck in RNNs

- Token $t$ must wait for $t-1$ to finish; no full-sequence parallelism.
- Modern computer accelerators favor wide, batched matrix operations; serial chains underutilize compute.




Transformers replace serial recurrence with *parallel* attention.

# The starting point: mean-pooling for RNNs




How to compute sentence encoding?

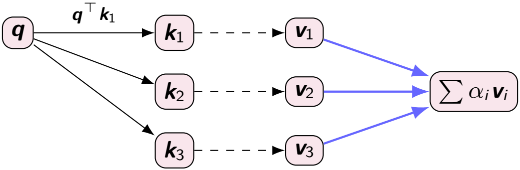
**Usually better:**
Take element-wise **max** or **mean** of all hidden states.

# Attention as weighted averaging with “learned" weights

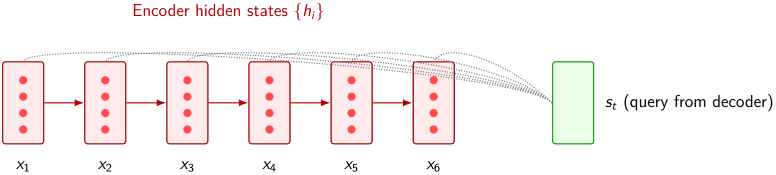
- Each query $\boldsymbol{q}$ *looks up* relevant information in (key, value) $\{(\boldsymbol{k}_i, \boldsymbol{v}_i)\}$.
- Relevance via similarity $s_i = \boldsymbol{q}^\top \boldsymbol{k}_i$; aggregate values by soft weights.

$$\alpha_i \;=\; \mathrm{softmax}\left(\frac{\boldsymbol{q}^\top \boldsymbol{k}_i}{\sqrt{d_k}}\right), \qquad \mathrm{Attn}(\boldsymbol{q}, K, V) \;=\; \sum\nolimits_i \alpha_i \, \boldsymbol{v}_i.$$

Scale $1/\sqrt{d_k}$ stabilizes dot products as $d_k$ grows.

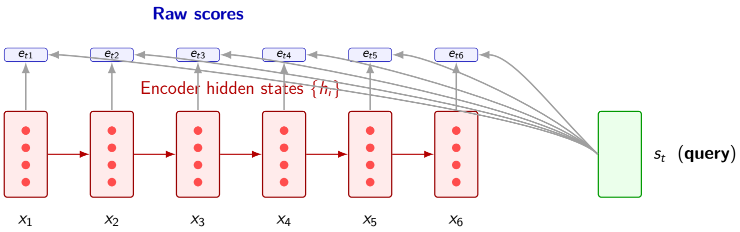


# Sequence-to-sequence with attention



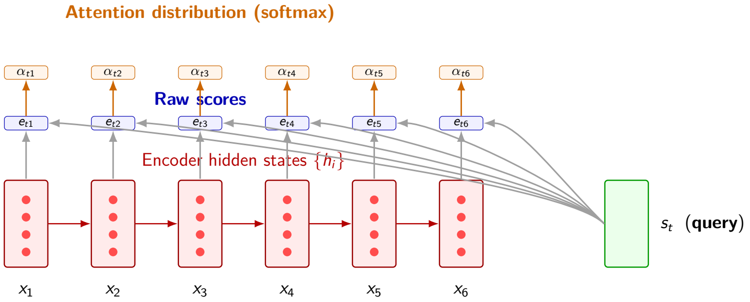

We have encoder vectors $\{h_i\}$ and the current decoder state $s_t$ (**query**).

# Computing attention scores




$e_{ti} = \mathrm{score}(s_t, h_i)$ (dot, general, or additive/Bahdanau)

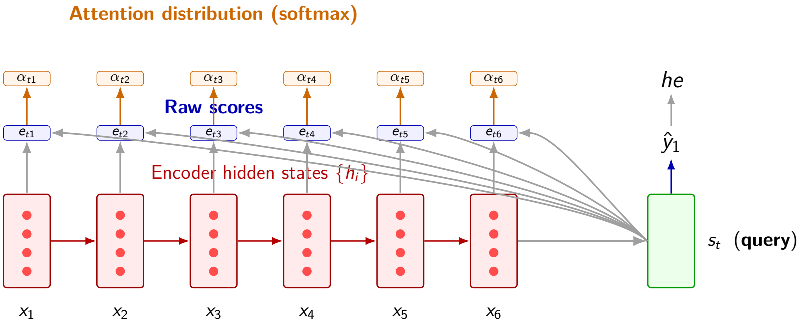
# From raw scores to attention distribution




$$e_{ti} = \mathrm{score}(s_t, h_i), \quad \alpha_{ti} = \mathrm{softmax}(e_{ti}) = \frac{\exp(e_{ti})}{\sum_j \exp(e_{tj})}$$

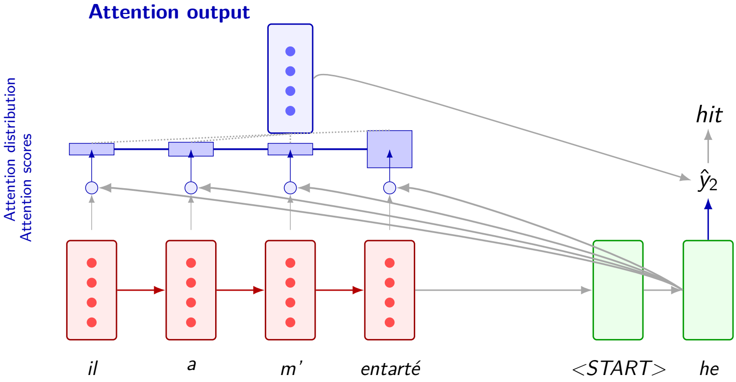
- Use the attention distribution to weight the encoder hidden states.
- The attention output mostly contains information from the hidden states that received high attention.

# From attention distribution to next-token prediction




Concatenate attention output with decoder hidden state, then use to compute $\hat{y}_1$ as before.

# From attention distribution to next-token prediction




Sometimes we take the attention output from the previous step, and also feed it into the decoder (along with the usual decoder input).

# Computing attention step-by-step

- We have encoder hidden states $h_1, \ldots, h_N \in \mathbb{R}^h$
- On timestep $t$, we have decoder hidden state $s_t \in \mathbb{R}^h$
- We get the attention scores $e^t$ for this step:

$$e^t = [\, s_t^\top h_1, \ldots, s_t^\top h_N \,] \in \mathbb{R}^N$$

There are multiple ways to do this

- We take softmax to get the attention distribution $\alpha^t$ for this step (this is a probability distribution and sums to 1)

$$\alpha^t = \mathrm{softmax}(e^t) \in \mathbb{R}^N$$

- Use $\alpha^t$ to get the attention output: $a_t = \sum_{i=1}^N \alpha_i^t h_i \in \mathbb{R}^h$
- Concatenate the attention and the decoder hidden state to proceed:

$$[\, a_t ; s_t \,] \in \mathbb{R}^{2h}$$

# There are several attention variants

- **Dot-product attention:** (assume $d_1 = d_2$ - the version we saw earlier)

$$e_i = s^\top h_i \in \mathbb{R}$$

- **Multiplicative attention:** $W \in \mathbb{R}^{d_2 \times d_1}$ is a weight matrix.

$$e_i = s^\top W h_i \in \mathbb{R}$$

- **Reduced-rank attention:** $U \in \mathbb{R}^{k \times d_2}$, $V \in \mathbb{R}^{k \times d_1}$, $k \ll d_1, d_2$

$$e_i = s^\top (U^\top V) h_i = (Us)^\top (V h_i)$$

- **Additive attention:** [Bahdanau, Cho, and Bengio 2014]

$$e_i = v^\top \tanh(W_1 h_i + W_2 s) \in \mathbb{R}$$

  - $W_1 \in \mathbb{R}^{d_3 \times d_1}$, $W_2 \in \mathbb{R}^{d_3 \times d_2}$ weight matrices; $v \in \mathbb{R}^{d_3}$ weight vector.
  - $d_3$ (the attention dimensionality) is a hyperparameter.
  - “Additive" is a weird/bad name - using a FNN layer.

# Attention is a general deep learning technique

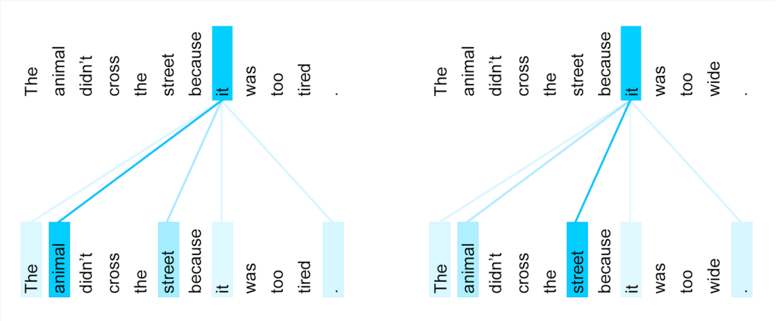
- **More general definition of attention:**

  Given a set of *values*, and a vector *query*, **attention** is a technique to compute a weighted sum of the values, dependent on the query.

- We sometimes say that the query attends to the values.
- For example, in the seq2seq + attention model, each decoder hidden state (query) attends to all the encoder hidden states (values).

# Table of Contents

# Can we get rid of recurrence entirely?



# Self-attention: keys, queries, values from same sequence

Let $\mathbf{w}_{1:n}$ be a sequence of words in vocabulary $V$.

For each $\mathbf{w}_i$, let $\mathbf{x}_i = E\mathbf{w}_i$, where $E \in \mathbb{R}^{d \times |V|}$ is an embedding matrix.

- Transform each word embedding with trainable weight matrices $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d \times d}$:

$$\mathbf{q}_i = \mathbf{W}_Q \mathbf{x}_i \text{ (queries)} \quad \mathbf{k}_i = \mathbf{W}_K \mathbf{x}_i \text{ (keys)} \quad \mathbf{v}_i = \mathbf{W}_V \mathbf{x}_i \text{ (values)}$$

- Compute pairwise similarities between keys and queries; normalize with softmax:

$$e_{ij} = \mathbf{q}_i^\top \mathbf{k}_j, \quad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{j'} \exp(e_{ij'})}$$

- Compute output for each word as weighted sum of values:

$$\mathbf{o}_i = \sum_j \alpha_{ij} \mathbf{v}_i$$

# Scaled dot-product attention

- Matrix form (multiple tokens): $\mathbf{Q} \in \mathbb{R}^{n \times d_k}, \mathbf{K} \in \mathbb{R}^{n \times d_k}, \mathbf{V} \in \mathbb{R}^{n \times d_v}$

$$\mathrm{Attn}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \;=\; \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right) \mathbf{V}.$$

- **Why the scale** $1/\sqrt{d_k}$? If entries are i.i.d. with variance $\sigma^2$, then $\mathrm{Var}(\boldsymbol{q}^\top \boldsymbol{k}) \propto d_k \sigma^4$; scaling keeps logits in a stable range for softmax.

- **Benefit:** enables *parallel* matmuls ($\mathbf{Q}\mathbf{K}^\top$ and with $\mathbf{V}$) over the entire sequence.

All tokens attend to all tokens in one or a few large matrix multiplies.

# Barriers for Self-Attention as a building block

**Barriers**

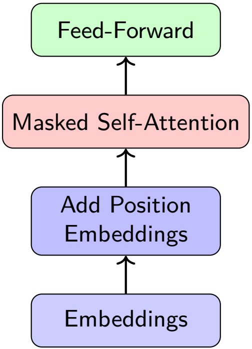
- Doesn't have an inherent notion of order!
- No nonlinearities for deep learning magic!
  It's all just weighted averages
- Need to ensure we don't "look at the future" when predicting a sequence
  - Like in machine translation
  - Or language modeling

**Solutions**

- Add position representations to the inputs
- Easy fix: apply the same feedforward network to each self-attention output
- Mask out the future by artificially setting attention weights to 0!

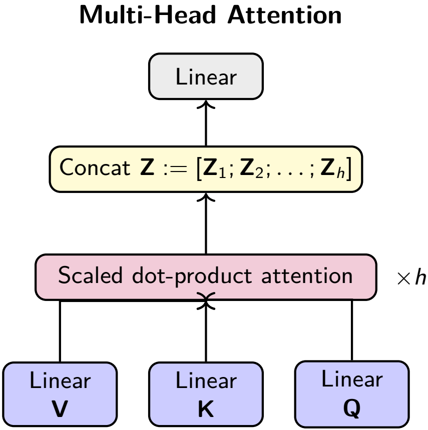
# Solutions for a self-attention building block:

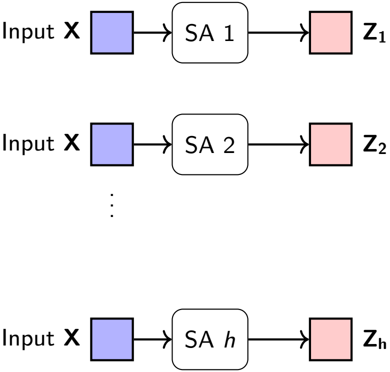
- **Self-attention:** The basis.
- **Position representations:**
  - Specify the sequence order, since self-attention is an unordered function of its inputs.
- **Nonlinearities:**
  - At the output of the self-attention.
  - Frequently implemented as a simple feed-forward network.




- **Masking:**
  - In order to parallelize operations while not looking at the future.
  - Keeps information about the future from “leaking" to the past.

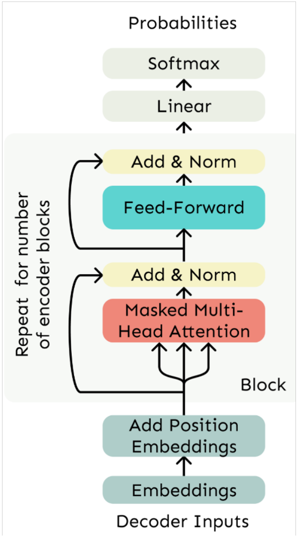
# Use multi-head attention to achieve better representation



# The transformer decoder

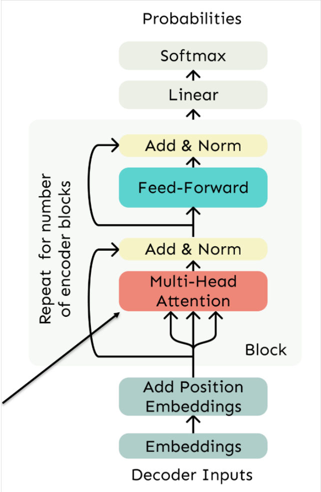
- The Transformer **Decoder** is a stack of Transformer **Decoder blocks**.
- Each block consists of:
  - Self-attention
  - Add & Norm
  - Feed-forward
  - Add & Norm
- That's it! We've gone through the Transformer Decoder.



# The transformer encoder

- The Transformer **decoder** constrains to **unidirectional context**, as for **language models**.
- What if we want **bidirectional context**, like in a bidirectional RNN?
- This is the **Transformer encoder**. The only difference is that we **remove the masking** in the self-attention.

**No Masking!**



# Self-Supervised Learning (SSL): the core idea

**Core Idea:** Train a model to solve a "puzzle" derived from the data itself, so it learns useful internal representations.

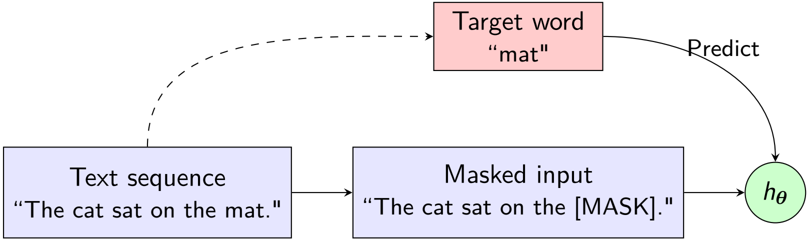



Figure: Learning by predicting missing words in text.

# SSL in action: masked word prediction

**Example: Masked Language Modeling (MLM)**

1. Start with a complete sentence: "The cat sat on the mat."
2. Mask one token $\rightarrow$ Input $\mathbf{x}' =$ "The cat sat on the [MASK]."
3. Pseudo-label $y_{\text{pretext}} =$ "mat."
4. Train the model to predict the masked token via cross-entropy loss

$$\min_{\boldsymbol{\theta}} \; J(\boldsymbol{\theta}) = -\sum_{i} \log P(y_{\text{pretext}}^{(i)} \mid \mathbf{x}'^{(i)}; \boldsymbol{\theta})$$

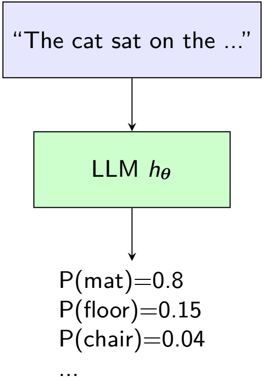
## Goal

Learn language representations by predicting what's missing - without human labels.

# LLM pre-training: self-supervision at scale

**Pretext Task:** Predict the next token given the previous ones.

$$\min_{\boldsymbol{\theta}} J_{\text{pre}}(\boldsymbol{\theta}) = -\sum_{i=1}^{m} \log P(w_i \mid w_{<i}; \boldsymbol{\theta})$$

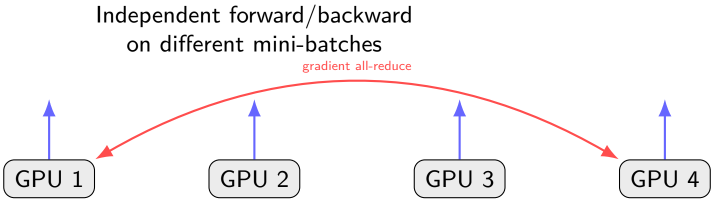



**Learning objective:**

- Train on massive text corpora without human labels.
- Use categorical cross-entropy loss.
- Model learns to represent syntax, semantics, and context.

# From attention parallelism to distributed optimization

- **RNNs:** sequential dependency ⇒ limited overlap of forward/backward passes.
- **Transformers:** attention-based layers ⇒ fully batched computation.
- Enables efficient **data, model, and pipeline parallelism**.




Parallel architecture ⇒ distributed optimization using SGD + AllReduce.

# Quadratic computation as a function of sequence length

- One of the benefits of self-attention over recurrence is that it's highly **parallelizable**.
- However, its total number of operations grows as $O(n^2 d)$, where $n$ is the sequence length and $d$ is the dimensionality.

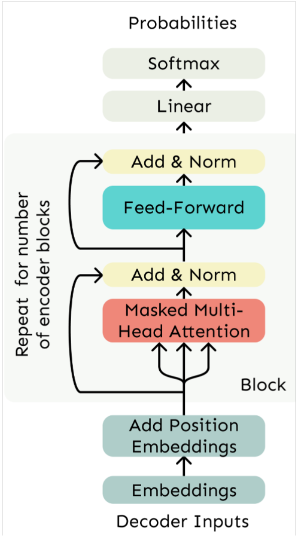
$$n \underbrace{\boxed{\;Q\;}}_{d} \times d \underbrace{\boxed{\;K^\top\;}}_{n} = \boxed{QK^\top}$$

- Think of $d$ as around **1,000** (though for LLMs it's much larger!).
- So, for a single (shortish) sentence, $n \leq 30$; thus $n^2 \leq 900$.
- In practice, we often set a bound like $n = 512$.
- **But what if we'd like $n \geq 50{,}000$?** For example, long documents?

# Table of Contents

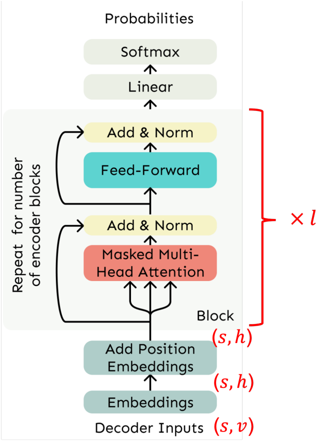
# GPT: Generative pre-trained transformer

- A generative pre-trained transformer (GPT) is a type of LLMs.
- GPT is based on the decoder-only transformer.
- Each block consists of:
  - Self-attention
  - Add & Norm
  - Feed-forward
  - Add & Norm
- We will analyze the parameters, memories, and computation costs for decoder-only transformer.



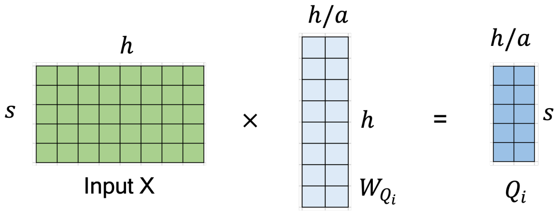
# Notations for GPT

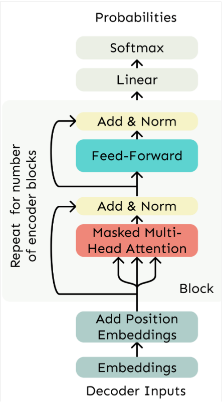
- Number of the transformer layers: $l$
- Sequence length: $s$
- Vocabulary size: $v$
- Embedding representation dims: $h$



# Multi-head self-attention computations

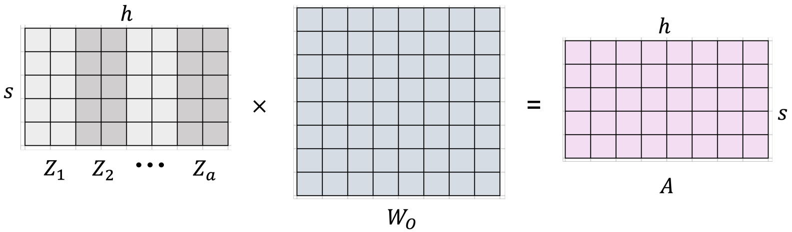
- Number of heads: $a$
- Dims of each $W_{Q_i}$, $W_{K_i}$ and $W_{V_i}$: $h \times \frac{h}{a}$



# Multi-head self-attention memory

- Number of heads: $a$
- Dims of each $W_O$: $h \times h$
- Dims of each $W_{Q_i}$, $W_{K_i}$ and $W_{V_i}$: $h \times \frac{h}{a}$



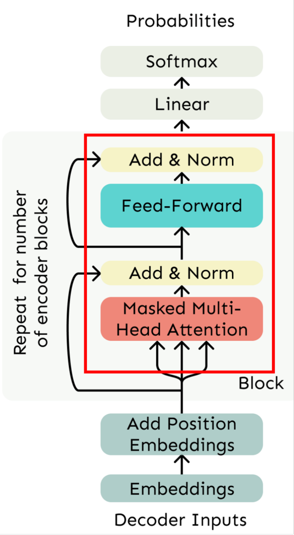

We need to store $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V$ and $\mathbf{W}_O$, which is in total $4h^2$

$$3 \times \frac{h^2}{\alpha} \times \alpha = 3h^2 \quad \text{and} \quad h^2$$

# Feed-forward layer memory

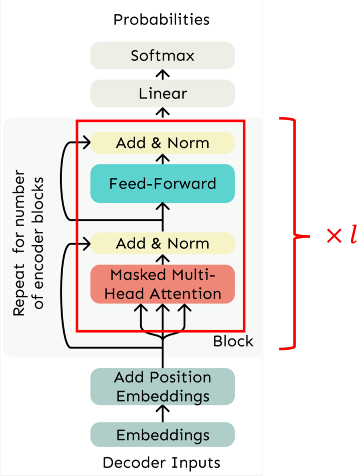
$$X' = \mathrm{ReLU}(A \cdot W_1 + b_1) \cdot W_2 + b_2$$

- Dims of $W_1$: $h \times 4h$
- Dims of each $W_2$: $4h \times h$
- We need to store $W_1$ and $W_2$ : $8h^2$
- The storage of $b_1$ and $b_2$ can be ignored



# Transformer block memory

- Multi-head attentions: $4h^2$
- Feed-forward layers : $8h^2$
- $l$ layers of attentions : $(4h^2 + 8h^2)\times l = 12lh^2$



# Recap and fine-tuning

- What we have talked about today?
  - $\Rightarrow$ How attention enables parallel and distributed computation?
  - $\Rightarrow$ How transformer-based models build upon attention blocks?
  - $\Rightarrow$ What is the memory complexity of decoder-based transformers?

Welcome anonymous survey!