

Generative Models for NLP

Attention Mechanisms and Transformer Architectures

Nadi Tomeh

7/2/25

Notes

Outline

- Recap and Motivation
- Expanding RNN Memory Beyond a Single Hidden State
- Attention Mechanisms
- Transformer Architecture for Language Modeling
- Training Transformer Models
- Pretraining and Fine-Tuning Transformers
- Transformer Setup Variants: GPT, Full Transformer, and BERT

Notes

Outline

- **Recap and Motivation**
- Expanding RNN Memory Beyond a Single Hidden State
- Attention Mechanisms
- Transformer Architecture for Language Modeling
- Training Transformer Models
- Pretraining and Fine-Tuning Transformers
- Transformer Setup Variants: GPT, Full Transformer, and BERT

Notes

Recap of RNN-Based Language Models

Quick Review

- **RNN/GRU/LSTM Architectures:** These architectures process sequences token by token, updating a hidden state $\mathbf{h}_t$ at each step t .
 - $\mathbf{h}_t$ captures the information from all previously seen tokens.
 - GRUs and LSTMs introduce gating mechanisms to mitigate vanishing or exploding gradients.

- **Hidden State $\mathbf{h}_t$:**
 - Serves as a summary (or *memory*) of the sequence up to position t .
 - Used for predicting the next token in a language modeling setup:

$$p(w_{t+1} \mid w_1 \dots w_t) \approx g_{\theta}(\mathbf{h}_t),$$

where $\mathbf{h}_t$ evolves from the previous hidden state and the current input token embedding.

Limitations of RNN-Based Models

- **Sequential Dependence:**
 - For a sequence of length n , RNNs require $O(n)$ steps of recurrent updates. Can be slow and hard to parallelize.
- **Difficulty Capturing Long-Range Context:**
 - Even LSTMs/GRUs can struggle with extremely distant dependencies, as gradients still degrade over many timesteps.

Notes

Why RNNs Cannot Be Parallelized Across Time

Core Recurrence Equation

In a n RNN, the hidden state $\mathbf{h}_t \in \mathbb{R}^m$ at time t is defined by a recurrence of the form:

$$\mathbf{h}_t = f_{\theta}(\mathbf{h}_{t-1}, \mathbf{w}_t),$$

Forward Pass Constraint

Because $\mathbf{h}_t$ depends on $\mathbf{h}_{t-1}$, each state must be computed *in sequence*:

$$\mathbf{h}_1 \rightarrow \mathbf{h}_2 \rightarrow \dots \rightarrow \mathbf{h}_t.$$

We cannot compute $\mathbf{h}_t$ until we have $\mathbf{h}_{t-1}$. This prohibit parallelizing over time steps in the forward pass.

BPTT Perspective

The gradient w.r.t. $\mathbf{h}_{t-1}$ involves a Jacobian term:

$$\frac{\partial \ell}{\partial \mathbf{h}_{t-1}} = \frac{\partial \ell}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} + \dots$$

where $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$ must be known before updating $\mathbf{h}_{t-1}$: gradients also have to be propagated *step-by-step*.

Notes

$\mathbf{h}_t$ as Memory: Markovian Perspective and Short Memory

Hidden State $\mathbf{h}_t$ as a Memory of the Past

- In an RNN, the hidden state $\mathbf{h}_t \in \mathbb{R}^m$ evolves via:

$$\mathbf{h}_t = f_{\theta}(\mathbf{h}_{t-1}, \mathbf{x}_t),$$

capturing *all* past inputs $\{\mathbf{x}_1, \dots, \mathbf{x}_t\}$ through a single vector.

- Intuitively, $\mathbf{h}_t$ serves as the network's *internal memory*, summarizing prior context relevant for predicting future tokens.

Markovian and Geometric Ergodicity

- $\mathbf{h}_t$ forms a **Markov chain** in the hidden-space:

$$p(\mathbf{h}_t \mid \mathbf{h}_{t-1}, \mathbf{h}_{t-2}, \dots) = p(\mathbf{h}_t \mid \mathbf{h}_{t-1}).$$

- Under mild contractive conditions on f_{θ} (e.g., Lipschitz constant < 1 in a bounded region), the Markov chain is *geometrically ergodic*: For any two initial states $\mathbf{h}_0$ and $\mathbf{h}'_0$, we have

$$\|\mathbf{h}_t - \mathbf{h}'_t\| \leq \lambda^t \|\mathbf{h}_0 - \mathbf{h}'_0\|, \quad \text{for some } 0 < \lambda < 1.$$

- The result is **Exponential Forgetting**: The influence of initial states $\mathbf{h}_0$ vanishes at a rate λ^t .

Notes

Outline

- Recap and Motivation
- Expanding RNN Memory Beyond a Single Hidden State
- Attention Mechanisms
- Transformer Architecture for Language Modeling
- Training Transformer Models
- Pretraining and Fine-Tuning Transformers
- Transformer Setup Variants: GPT, Full Transformer, and BERT

Notes

Bottleneck of a Single Hidden State $\mathbf{h}_t$

Recurrence in a Standard Elman RNN LM

- Hidden state update:

$$\mathbf{h}_t = \tanh(\mathbf{W}_{xh} \mathbf{x}_t + \mathbf{W}_{hh} \mathbf{h}_{t-1} + \mathbf{b}_h).$$

- Prediction logits:

$$\mathbf{z}_t = \mathbf{W}_{hy} \mathbf{h}_t + \mathbf{b}_y, \quad \mathbf{p}_t = \text{softmax}(\mathbf{z}_t), \quad p_{\theta}(w_{t+1} \mid w_{1:t}) = \mathbf{p}_{t, w_{t+1}}.$$

- $\mathbf{h}_t \in \mathbb{R}^m$, $\mathbf{x}_t \in \mathbb{R}^d$, $\mathbf{W}_{xh} \in \mathbb{R}^{m \times d}$, $\mathbf{W}_{hh} \in \mathbb{R}^{m \times m}$, $\mathbf{W}_{hy} \in \mathbb{R}^{|\mathcal{V}| \times m}$, etc.

The Bottleneck

- $\mathbf{h}_t$ must encode *all* relevant history in a single vector of size m .
- As t grows, $\mathbf{h}_t$ struggles to maintain detailed information about very distant tokens.
- This can degrade the accuracy of $\mathbf{z}_t$ (the logits) and thus the next-word distribution.

Notes

Storing All Previous Hidden States in $\mathcal{M}_t$

Expandable Memory

- Instead of relying purely on $\mathbf{h}_t$, we keep each past hidden state:

$$\mathcal{M}_t = \{\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_t\}.$$

- Each $\mathbf{h}_\tau \in \mathbb{R}^m$ can be viewed as an encoding of the input at time τ .
- $\mathcal{M}_t$ *expands* over time, forming a *dynamically growing repository* of contextual vectors.

Context Vector $\mathbf{c}_t$

- We combine the memory vectors in $\mathcal{M}_t$ into a single $\mathbf{c}_t \in \mathbb{R}^m$, representing the *relevant* information from $\{\mathbf{h}_1, \dots, \mathbf{h}_t\}$.
- Next, we incorporate $\mathbf{c}_t$ along with $\mathbf{h}_t$ to predict:

$$\mathbf{z}_t = \mathbf{W}_{cy} [\mathbf{h}_t; \mathbf{c}_t] + \mathbf{b}_y, \quad \mathbf{p}_t = \text{softmax}(\mathbf{z}_t).$$

- Here, $[\mathbf{h}_t; \mathbf{c}_t] \in \mathbb{R}^{2m}$ is the concatenation; $\mathbf{W}_{cy} \in \mathbb{R}^{|\mathcal{V}| \times 2m}$.

Navigation icons: back, forward, search, etc.

Notes

Combining the Memory Vectors $\mathcal{M}_t = \{\mathbf{h}_1, \dots, \mathbf{h}_t\}$ into a Context Vector

Naive Summation or Averaging

- **Sum or average:**

$$\mathbf{c}_t = \sum_{\tau=1}^t \mathbf{h}_\tau \quad \text{or} \quad \mathbf{c}_t = \frac{1}{t} \sum_{\tau=1}^t \mathbf{h}_\tau.$$

- *All vectors contribute equally*, often losing important distinctions among tokens (no weighting).

Hard Selection (One-Hot or Multi-Hot)

- Define a discrete vector $\alpha_t \in \{0, 1\}^t$, then:

$$\mathbf{c}_t = \frac{1}{\sum_{\tau=1}^t \alpha_{t,\tau}} \sum_{\tau=1}^t \alpha_{t,\tau} \mathbf{h}_\tau.$$

- **One-hot:** exactly one entry $\alpha_{t,\tau^*} = 1$, rest 0.
- **Multi-hot:** could select multiple $\mathbf{h}_\tau$ simultaneously.
- *Non-differentiable* w.r.t. $\alpha_{t,\tau}$, complicates gradient-based learning.

Navigation icons: back, forward, search, etc.

Notes

Combining the Memory Vectors $\mathcal{M}_t = \{\mathbf{h}_1, \dots, \mathbf{h}_t\}$ into a Context Vector II

Differentiable Soft Selection

- Let $\alpha_{t,\tau} \in [0, 1]$ with $\sum_{\tau=1}^t \alpha_{t,\tau} = 1$.

$$\mathbf{c}_t = \boldsymbol{\alpha}_t^\top \cdot \mathbf{h}_t = \sum_{\tau=1}^t \alpha_{t,\tau} \mathbf{h}_\tau.$$

- $\mathbf{c}_t$ is a *weighted combination* of memory vectors, focusing on relevant ones.
- $\{\alpha_{t,\tau}\}$ can be learned end-to-end with backprop.

Notes

Outline

- Recap and Motivation
- Expanding RNN Memory Beyond a Single Hidden State
- **Attention Mechanisms**
- Transformer Architecture for Language Modeling
- Training Transformer Models
- Pretraining and Fine-Tuning Transformers
- Transformer Setup Variants: GPT, Full Transformer, and BERT

Notes

Learning the Weights $\alpha_{t,\tau}$ with Attention

Attention as a Similarity-Based Weighting

- We want to find how much each past hidden state $\mathbf{h}_\tau$ (for $\tau = 1, \dots, t-1$) contributes to the current context.
- Define a **score function** $e_{t,\tau}$ that measures the similarity between $\mathbf{h}_t$ (the “query”) and $\mathbf{h}_\tau$ (the “key”).

$$\mathbf{s}_{t,\tau} = \text{sim}(\mathbf{h}_t, \mathbf{h}_\tau).$$

- We then convert these raw scores $\{e_{t,\tau}\}$ into attention weights via a softmax:

$$\alpha_{t,\tau} = \frac{\exp(s_{t,\tau})}{\sum_{k=1}^{t-1} \exp(s_{t,k})}, \quad \text{for } \tau = 1, \dots, t-1 \quad \text{This ensures } \sum_{\tau=1}^{t-1} \alpha_{t,\tau} = 1.$$

Context Vector $\mathbf{c}_t$ Using Learned Weights

$$\mathbf{c}_t = \sum_{\tau=1}^{t-1} \alpha_{t,\tau} \mathbf{h}_\tau.$$

- The $\alpha_{t,\tau}$ are learned *dynamically* based on the similarity of $\mathbf{h}_t$ and $\mathbf{h}_\tau$, focuses on relevant past states.

Notes

[illegible]

Four Common Similarity (Score) Functions I

1. Dot Product

$$s_{t,\tau} = \mathbf{h}_t^\top \mathbf{h}_\tau,$$

where $\mathbf{h}_t, \mathbf{h}_\tau \in \mathbb{R}^m$.

- Simple and fast; purely linear similarity.
- Works well if the norms $\|\mathbf{h}_t\|$ and $\|\mathbf{h}_\tau\|$ are not too large.

2. Scaled Dot Product (Vaswani)

$$S_{t,\tau} = \frac{\mathbf{h}_t^\top \mathbf{h}_\tau}{\sqrt{m}},$$

where again $\mathbf{h}_t, \mathbf{h}_\tau \in \mathbb{R}^m$.

- Dividing by $\sqrt{m}$ (the dimension of $\mathbf{h}$) prevents large dot-product values.
- Popular in Transformer architectures.

Notes

[illegible]

Four Common Similarity (Score) Functions II

3. Bilinear (Luong Attention)

$$s_{t,\tau} = \mathbf{h}_t^\top \mathbf{W}_{\text{attn}} \mathbf{h}_\tau, \quad \mathbf{W}_{\text{attn}} \in \mathbb{R}^{m \times m}.$$

- Learns a transformation of $\mathbf{h}_\tau$ *before* comparing to $\mathbf{h}_t$.
- More expressive than a raw dot product, but adds $\mathcal{O}(m^2)$ parameters.

4. MLP (Additive / Bahdanau Attention)

$$s_{t,\tau} = \mathbf{v}_a^\top \tanh(\mathbf{W}_a \mathbf{h}_t + \mathbf{U}_a \mathbf{h}_\tau),$$

where $\mathbf{W}_a, \mathbf{U}_a \in \mathbb{R}^{m \times m}$, $\mathbf{v}_a \in \mathbb{R}^m$.

- Uses a small neural network for scoring each pair $(\mathbf{h}_t, \mathbf{h}_\tau)$.
- Potentially more flexible than dot-based approaches, but computationally heavier.

Notes

Extending RNNs to Sequence-to-Sequence Models

Goal: Machine Translation (MT) Example

- **Input (source sequence):** $(w_1^{\text{src}}, \dots, w_n^{\text{src}})$.
- **Output (target sequence):** $(w_1^{\text{trg}}, \dots, w_m^{\text{trg}})$.
- We want to learn $p_\theta(w_1^{\text{trg}}, \dots, w_m^{\text{trg}} \mid w_1^{\text{src}}, \dots, w_n^{\text{src}})$, a *conditional* generative model.

Encoder–Decoder Architecture

- **Encoder (RNN):** Processes the source tokens into hidden states $\{\mathbf{h}_1^{\text{enc}}, \dots, \mathbf{h}_n^{\text{enc}}\}$.
- **Decoder (RNN):** Generates target tokens (w_t^{trg}) one by one, conditioning on the encoder outputs.
- Without attention, the decoder uses only the *final* encoder hidden state (a single vector) as a context:

$$\mathbf{h}_{\text{context}} = \mathbf{h}_n^{\text{enc}}.$$

- **Limitation:** A single fixed-size vector $\mathbf{h}_n^{\text{enc}}$ must encode the entire source sentence: bottleneck, again.

Notes

Integrating Attention in Seq2Seq (Bahdanau et al.)

Attention Over Encoder Hidden States

- Instead of relying on $\mathbf{h}_n^{\text{enc}}$ alone, maintain a memory of $\{\mathbf{h}_1^{\text{enc}}, \dots, \mathbf{h}_n^{\text{enc}}\}$.
- At each decoder timestep t :

$$\mathbf{c}_t^{\text{enc}} = \sum_{\tau=1}^n \alpha_{t,\tau} \mathbf{h}_{\tau}^{\text{enc}}, \quad \alpha_{t,\tau} = \frac{\exp(s(\mathbf{h}_t^{\text{dec}}, \mathbf{h}_{\tau}^{\text{enc}}))}{\sum_{k=1}^n \exp(s(\mathbf{h}_t^{\text{dec}}, \mathbf{h}_k^{\text{enc}}))},$$

where $\mathbf{h}_t^{\text{dec}} \in \mathbb{R}^m$ is the current decoder hidden state.

- The $\alpha_{t,\tau}$ measure how relevant the encoder's state $\mathbf{h}_{\tau}^{\text{enc}}$ is at decoding step t .

Context-Augmented Decoder State

- The decoder RNN update can then incorporate $\mathbf{c}_t^{\text{enc}}$ (plus $\mathbf{h}_{t-1}^{\text{dec}}$, the previous decoder state) to predict:

$$\mathbf{h}_t^{\text{dec}} = f_{\theta}(\mathbf{h}_{t-1}^{\text{dec}}, \mathbf{w}_t^{\text{trg}}, \mathbf{c}_t^{\text{enc}}) \quad \text{and} \quad p_{\theta}(\mathbf{w}_t^{\text{trg}}) = \text{softmax}(\mathbf{W}_{hy} \mathbf{h}_t^{\text{dec}} + \mathbf{b}_y).$$

- **Result:** A *trainable alignment matrix* α via attention, letting the model focus on relevant source positions for each target word.

Navigation icons: back, forward, search, etc.

Notes

Outline

- Recap and Motivation
- Expanding RNN Memory Beyond a Single Hidden State
- Attention Mechanisms
- Transformer Architecture for Language Modeling
- Training Transformer Models
- Pretraining and Fine-Tuning Transformers
- Transformer Setup Variants: GPT, Full Transformer, and BERT

Navigation icons: back, forward, search, etc.

Notes

Overview and Motivation

Goal: Autoregressive Language Modeling

- We want a next-token distribution:

$$p_{\theta}(w_{t+1} \mid w_1, \dots, w_t),$$

but **without** RNN recurrence.

- **Decoder-Only Transformer**: Each token attends to all prior tokens *in parallel*, using a **mask** to maintain causal order.

High-Level Steps (One Layer)

1. **Multi-Head Self-Attention** (*masked*).
2. **Positional Encodings** to inject sequence ordering.
3. **Residual + LayerNorm**.
4. **Feed-Forward Network (FFN)** (applied to each token).
5. **Another Residual + LayerNorm**.

Stacked for L layers, then project to output logits.

Notes

Masked Multi-Head Self-Attention I

Token-by-Token Equations

Setup: We have n tokens, each with embedding $\mathbf{x}_i \in \mathbb{R}^d$, for $i = 1, \dots, n$.

- **Per-token** query, key, value:

$$\mathbf{q}_i = \mathbf{W}^Q \mathbf{x}_i \in \mathbb{R}^{d_k}, \quad \mathbf{k}_j = \mathbf{W}^K \mathbf{x}_j \in \mathbb{R}^{d_k}, \quad \mathbf{v}_j = \mathbf{W}^V \mathbf{x}_j \in \mathbb{R}^{d_v},$$

where $\mathbf{W}^Q, \mathbf{W}^K \in \mathbb{R}^{d \times d_k}$, $\mathbf{W}^V \in \mathbb{R}^{d \times d_v}$.

- **Scores and softmax:**

$$s_{i,j} = \frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_k}}, \quad \alpha_{i,j} = \frac{\exp(s_{i,j})}{\sum_{m=1}^n \exp(s_{i,m})}.$$

- **Attention output for token i :**

$$\mathbf{y}_i = \sum_{j=1}^n \alpha_{i,j} \mathbf{v}_j.$$

Notes

Masked Multi-Head Self-Attention II

Matrix Form Equations (All Tokens in Parallel)

Collect token embeddings $\mathbf{x}_i$ into matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$: $\mathbf{X} = [\mathbf{x}_1^\top; \dots; \mathbf{x}_1^\top]$.

$$\mathbf{Q} = \mathbf{X} \mathbf{W}^Q \in \mathbb{R}^{n \times d_k}, \quad \mathbf{K} = \mathbf{X} \mathbf{W}^K \in \mathbb{R}^{n \times d_k}, \quad \mathbf{V} = \mathbf{X} \mathbf{W}^V \in \mathbb{R}^{n \times d_v}.$$

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q} \mathbf{K}^\top}{\sqrt{d_k}}\right) \mathbf{V}, \quad \in \mathbb{R}^{n \times d_v}.$$

Each row of the result is $\mathbf{y}_i^\top$, matching the per-token outputs $\mathbf{y}_i$ from the single-element view.

Notes

Masked Multi-Head Self-Attention III

Masked Self-Attention (Causal LM)

- For **autoregressive** language modeling, token i must *not* attend to tokens $j > i$.
- We add a $\mathbf{M} \in \mathbb{R}^{n \times n}$:

$$\mathbf{M}[i, j] = \begin{cases} 0, & \text{if } j \leq i, \\ -\infty, & \text{if } j > i. \end{cases}$$

- Then:

$$\mathbf{Q} \mathbf{K}^\top + \mathbf{M} \xrightarrow{\text{softmax}} \in \mathbb{R}^{n \times n} \text{ ensures each row } i \text{ ignores columns } j > i.$$

- Output shape still $\mathbb{R}^{n \times d_v}$, but strictly *left-to-right* in coverage.

Example: $n = 4$ Tokens

$$\mathbf{M} = \begin{pmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Token visibility:

- Token #1 sees no preceding tokens (only itself).
- Token #2 sees #1 and itself, but not #3 or #4.
- Etc.

Notes

Masked Multi-Head Self-Attention IV

Multi-Head Extension

- **Multiple sets** of $\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V$ for $i = 1, \dots, h$.

$$\text{head}_i = \text{Attention}\left(\mathbf{X} \mathbf{W}_i^Q, \mathbf{X} \mathbf{W}_i^K, \mathbf{X} \mathbf{W}_i^V\right) \in \mathbb{R}^{n \times d_v}.$$

- **Concatenate** heads:

$$\text{MultiHead}(\mathbf{X}) = \text{Concat}(\text{head}_1, \dots, \text{head}_h) \mathbf{W}^O, \quad \mathbf{W}^O \in \mathbb{R}^{(h \cdot d_v) \times d}.$$

- **Why multi-head?**
Different heads can specialize: e.g., local vs. distant context, syntactic vs. semantic cues, etc.

Notes

The Problem of Order and the Role of Positional Encodings

Permutation-Invariant Attention

- So far, **self-attention** alone (without masks or positional info) is inherently *permutation-invariant*:
 - Swapping token i with token j in $\mathbf{X}$ just permutes the rows of $\mathbf{Q}, \mathbf{K}, \mathbf{V}$, and thus permutes the output as well.
- **Why is this a problem?**
 - A sentence like Cat chases dog conveys a different meaning if tokens are rearranged to Dog chases cat.
 - Pure attention sees token embeddings as a set with no inherent notion of "first token," "second token," etc.

Positional Encodings: Injecting Order

- We assign each position i a vector $\text{PE}(i) \in \mathbb{R}^d$.
- Then we **add** $\text{PE}(i)$ to the original token embedding $\mathbf{x}_i$:

$$\mathbf{x}'_i = \mathbf{x}_i + \text{PE}(i).$$

- The model's self-attention layers now see $\mathbf{x}'_i$, which encodes both the token's identity *and* its position.

Learned vs. Sinusoidal

- **Learned** position embeddings: we maintain a parameter table $\mathbf{P} \in \mathbb{R}^{n_{\max} \times d}$, so $\text{PE}(i)$ is just $\mathbf{P}[i, :]$.
- **Sinusoidal**: uses sines and cosines at different frequencies to represent positions.

Notes

Sinusoidal Positional Encodings I

Formula

$$\text{PE}(\text{pos}, 2j) = \sin\left(\frac{\text{pos}}{10000^{2j/d}}\right), \quad \text{PE}(\text{pos}, 2j + 1) = \cos\left(\frac{\text{pos}}{10000^{2j/d}}\right),$$

where pos = position index $\in \{0, 1, \dots\}$, and $2j, 2j + 1$ index the even/odd dimensions in $\mathbb{R}^d$.

Why Sinusoids?

- **Relative Offsets:** $\text{PE}(\text{pos}_2) - \text{PE}(\text{pos}_1)$ can be learned by the model to represent “distance” between positions.
- **Periodicity:** The model can exploit trigonometric functions to detect repeating patterns (e.g., rhythmic or periodic structure).
- **No Extra Parameters:** These are fixed functions, so no large parameter table is needed.

Notes

Sinusoidal Positional Encodings II

Relative Offsets: Encoding Distance Between Tokens

- The difference between two positional encodings encodes relative position information:

$$\text{PE}(\text{pos}_2) - \text{PE}(\text{pos}_1) = 2 \cos\left(\frac{\text{pos}_1 + \text{pos}_2}{2 \cdot 10000^{2j/d}}\right) \sin\left(\frac{\text{pos}_2 - \text{pos}_1}{2 \cdot 10000^{2j/d}}\right).$$

- The model can learn to use this difference to infer **how far apart two tokens are**, rather than relying on absolute positions.
- This helps generalize to **longer sequences** beyond those seen in training.

Periodicity: Capturing Repeating Patterns

- The sinusoidal function is **periodic**, meaning:

$$\sin(x) = \sin(x + 2\pi k), \quad \forall k \in \mathbb{Z}.$$

- Different frequency components allow the model to capture:
 - **Short-range dependencies** (small denominator: high frequency).
 - **Long-range dependencies** (large denominator: low frequency).

Notes

Sinusoidal Positional Encodings III

Example: $d = 6, pos = 0 \dots 3$

- Suppose $d = 6$. Then half of those (3 dims) are sines (even indices: 0,2,4), half (odd indices: 1,3,5) are cosines.
- For positions $pos = 0, 1, 2, 3$, you might get:

$$PE(0) = \begin{pmatrix} \sin(0) \\ \cos(0) \\ \sin(0/10000^{1/3}) \\ \cos(0/10000^{1/3}) \\ \sin(0/10000^{2/3}) \\ \cos(0/10000^{2/3}) \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}.$$

- For $pos = 1$, these become small angles in some coordinates; for $pos = 2, 3$, the angles grow accordingly.

Notes

Residual + Layer Normalization I

Residual Connections

- Let $\mathbf{Z}$ = (Multi-Head Attn or FFN output) $\in \mathbb{R}^{n \times d}$, then the output becomes:

$$\mathbf{X}' = \mathbf{X} + \mathbf{Z}.$$

- Better gradient flow:** The gradient can “skip” sub-layers if needed, preventing severe vanishing/exploding issues in deep networks.
- Stabilizes training:** Each sub-layer only needs to learn a “residual” function around the identity. In practice, deeper models converge faster and more reliably.

Notes

Residual + Layer Normalization II

LayerNorm

$$\text{LayerNorm}(\mathbf{x}) = \frac{\mathbf{x} - \mu(\mathbf{x})}{\sigma(\mathbf{x})} \odot \boldsymbol{\gamma} + \boldsymbol{\beta}, \quad \mathbf{x} \in \mathbb{R}^d.$$

- $\mu(\mathbf{x}) = \frac{1}{d} \sum_{i=1}^d x_i$ is the **mean** of $\mathbf{x}$.
- $\sigma(\mathbf{x}) = \sqrt{\frac{1}{d} \sum_{i=1}^d (x_i - \mu(\mathbf{x}))^2}$ is the **standard deviation** of $\mathbf{x}$.
- $\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^d$ are learned **scale** and **shift** parameters.
- Helps maintain stable activations across tokens and layers.

Application within a Block

- Each Transformer sub-layer (Multi-Head Attention or FFN) is wrapped with:

$$\mathbf{X} \leftarrow \text{LayerNorm}(\mathbf{X} + \text{subLayer}(\mathbf{X})).$$

- Residual connections allow deeper networks by letting gradients bypass sub-layers if needed.
- LayerNorm ensures each token's feature dimension remains stable in mean and variance.

Navigation icons: back, forward, search, etc.

Notes

Position-Wise Feed-Forward Network (FFN)

Position-Wise MLP

Definition: For each token embedding $\mathbf{x} \in \mathbb{R}^d$, we apply a 2-layer feed-forward transformation:

$$\text{FFN}(\mathbf{x}) = \max(0, \mathbf{x} \mathbf{W}_1 + \mathbf{b}_1) \mathbf{W}_2 + \mathbf{b}_2,$$

where:

- $\mathbf{W}_1 \in \mathbb{R}^{d \times d_{\text{ff}}}$, $\mathbf{W}_2 \in \mathbb{R}^{d_{\text{ff}} \times d}$.
- $\mathbf{b}_1 \in \mathbb{R}^{d_{\text{ff}}}$, $\mathbf{b}_2 \in \mathbb{R}^d$.
- Typically $d_{\text{ff}} > d$; e.g. $d_{\text{ff}} = 2048$ and $d = 512$, called **bottleneck** $\rightarrow$ **expansion** $\rightarrow$ **projection** structure.

Shape & Per-Token Independence

- Input to FFN layer: $\mathbf{H} \in \mathbb{R}^{n \times d}$, where n is the number of tokens.
- We apply FFN *row by row*, i.e. each $\mathbf{h}_i \in \mathbb{R}^d$ (the i -th token's vector) is mapped to another $\mathbf{h}'_i \in \mathbb{R}^d$.
- $\max(0, \cdot)$ is the ReLU nonlinearity.
- This is called **position-wise** because each token's position is processed *independently*, ignoring any cross-token interaction in this sub-layer.

Navigation icons: back, forward, search, etc.

Notes

Putting It All Together: Composing Transformer Sublayers for LM I

Layer Composition in a Decoder Block

- Let the input to the first layer be

$$\mathbf{X}^{(0)} = \mathbf{X} + \text{PE} \in \mathbb{R}^{n \times d},$$

where $\mathbf{X} \in \mathbb{R}^{n \times d}$ are token embeddings and $\text{PE} \in \mathbb{R}^{n \times d}$ are positional encodings.

- Each decoder layer l (for $l = 1, \dots, L$) is a composition of sublayers:

$$\mathbf{X}^{(l)} = \text{LayerNorm}\left(\mathbf{X}^{(l-1)} + \text{FFN}\left(\text{LayerNorm}\left(\mathbf{X}^{(l-1)} + \text{MaskedMHA}(\mathbf{X}^{(l-1)})\right)\right)\right).$$

Notes

Putting It All Together: Composing Transformer Sublayers for LM II

Output Projection to Vocabulary Distribution

- After L layers, we obtain final representations:

$$\mathbf{Y} = \mathbf{X}^{(L)} \in \mathbb{R}^{n \times d}.$$

- For each token (row) $\mathbf{y}_i \in \mathbb{R}^d$ in $\mathbf{Y}$, compute logits:

$$\mathbf{z}_i = \mathbf{y}_i \mathbf{W}_{\text{out}} + \mathbf{b}_{\text{out}}, \quad \mathbf{W}_{\text{out}} \in \mathbb{R}^{d \times |\mathcal{V}|}, \quad \mathbf{b}_{\text{out}} \in \mathbb{R}^{|\mathcal{V}|}.$$

- Apply softmax to obtain the next-token probability distribution:

$$p_{\theta}(w_i \mid w_1, \dots, w_{i-1}) = \mathbf{p}_i = \text{softmax}(\mathbf{z}_i) \in \mathbb{R}^{|\mathcal{V}|}.$$

Notes

Outline

- Recap and Motivation
- Expanding RNN Memory Beyond a Single Hidden State
- Attention Mechanisms
- Transformer Architecture for Language Modeling
- **Training Transformer Models**
- Pretraining and Fine-Tuning Transformers
- Transformer Setup Variants: GPT, Full Transformer, and BERT

Notes

Training Transformers: Essential Points and Equations I

Training Objective

- For language modeling, minimize cross-entropy loss:

$$\mathcal{L}(\theta) = - \sum_{i=1}^n \log p_{\theta}(w_{i+1} \mid w_1, \dots, w_i),$$

where $p_{\theta}(w_{i+1} \mid w_1, \dots, w_i)$ is computed via a softmax over logits.

Optimizer and Learning Rate Schedule

- **Optimizer:** Adam/AdamW is used for adaptive moment estimation.
- **Learning Rate:** A warmup phase followed by inverse square-root decay:

$$\eta_t = d_{\text{model}}^{-0.5} \cdot \min(t^{-0.5}, t \tau^{-1.5}),$$

where τ is the warmup period (steps) and d_{model} is the model dimension.

Notes

Training Transformers: Essential Points and Equations II

Dropout

- For an activation vector $\mathbf{z} \in \mathbb{R}^d$, dropout applies a mask $\mathbf{m} \in \{0, 1\}^d$ with

$$m_i \sim \text{Bernoulli}(p),$$

and outputs

$$\tilde{\mathbf{z}} = \frac{\mathbf{z} \odot \mathbf{m}}{p}.$$

- Applied in attention, FFN, and residual connections to reduce overfitting.

Notes

Training Transformers: Essential Points and Equations III

Label Smoothing

- Instead of a one-hot target, assign a smoothed target distribution:

$$q(k) = \begin{cases} 1 - \epsilon, & \text{if } k = k^*, \\ \frac{\epsilon}{|\mathcal{V}| - 1}, & \text{if } k \neq k^*, \end{cases}$$

where k^* is the correct token, $|\mathcal{V}|$ is the vocabulary size, and ϵ is a small constant (e.g., 0.1).

- Helps prevent overconfidence and improves generalization.

Gradient Clipping

- To stabilize training, clip gradients:

$$\nabla_{\theta} \mathcal{L} \leftarrow \nabla_{\theta} \mathcal{L} \cdot \min\left(1, \frac{c}{\|\nabla_{\theta} \mathcal{L}\|}\right),$$

where c is the clipping threshold.

Notes

Why Transformers Train Efficiently on GPUs

Parallelizable Operations

- **Matrix Multiplications:** All sublayers (multi-head self-attention, feed-forward networks) involve large matrix multiplications that are highly optimized on GPUs.
- **Teacher Forcing in Training:** When training with teacher forcing, the target sequence is known. $\Rightarrow$ Losses for all positions are computed simultaneously by arranging tokens in tensors.
- **No Recurrence:** Unlike RNNs, Transformers do not require sequential updates over time steps. This allows all token positions to be processed in parallel.

Illustration: Parallel Loss Computation

- Suppose we have a batch of B sequences, each of length n . All token embeddings are stored in a tensor $\mathbf{X} \in \mathbb{R}^{B \times n \times d}$.
- The Transformer computes outputs $\mathbf{Y} \in \mathbb{R}^{B \times n \times d}$ in parallel for every position.
- The predicted logits $\mathbf{Z} \in \mathbb{R}^{B \times n \times |\mathcal{V}|}$ are computed via:

$$\mathbf{Z} = \mathbf{Y} \mathbf{W}_{\text{out}} + \mathbf{b}_{\text{out}},$$

and the cross-entropy loss is computed over all positions simultaneously.

Navigation icons: back, forward, search, etc.

Notes

Computational Complexity: Transformer vs. RNN I

Transformer Training Complexity

- **Parallel Processing:**
 - The entire input sequence of n tokens (batch size B) is processed simultaneously.
 - Token embeddings are arranged in a tensor: $\mathbf{X} \in \mathbb{R}^{B \times n \times d}$.
- **Self-Attention Computations:**
 - For each layer, self-attention requires computing the matrix product $\mathbf{Q} \mathbf{K}^\top$ with cost:
$$\mathcal{O}(B \cdot n^2 \cdot d_k),$$
where $\mathbf{Q}, \mathbf{K} \in \mathbb{R}^{B \times n \times d_k}$.
 - Additional matrix multiplications (e.g., with $\mathbf{V}$) also scale similarly.
- **Overall Training:**
 - Although self-attention has a quadratic dependency in n , modern GPUs/TPUs perform these large matrix multiplications in parallel.
 - Backpropagation is applied concurrently over all tokens, making training efficient even for long sequences.

Navigation icons: back, forward, search, etc.

Notes

Computational Complexity: Transformer vs. RNN II

RNN Training Complexity

- **Sequential Processing:**
 - An RNN processes tokens one by one, unrolling over n time steps.
 - The input is processed as a sequence: $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$ with recurrence.
- **Per-Step Cost:**
 - Each time step involves computing:

$$\mathbf{h}_t = f_{\theta}(\mathbf{h}_{t-1}, \mathbf{x}_t),$$

with cost $\mathcal{O}(f(d))$ per step.
- **Overall Training:**
 - Total cost per sequence: $\mathcal{O}(n \cdot f(d))$.
 - Gradients are propagated sequentially via BPTT, limiting parallelization.

Transformer Inference Complexity

- **Autoregressive Generation:** Inference is inherently sequential as each token depends on previously generated tokens: this requires $\mathcal{O}(t^2)$ operations (for a sequence of length t).
- **Caching Mechanism:** Previously computed key and value matrices are cached to avoid redundant computations.

Notes

Outline

- Recap and Motivation
- Expanding RNN Memory Beyond a Single Hidden State
- Attention Mechanisms
- Transformer Architecture for Language Modeling
- Training Transformer Models
- Pretraining and Fine-Tuning Transformers
- Transformer Setup Variants: GPT, Full Transformer, and BERT

Notes

Pretraining and Fine-Tuning: Technical Setup I

Pretraining Stage (Unsupervised)

- **Objective:** Learn general language representations from large-scale, unlabeled corpora.

- **Common Pretraining Objectives:**

- **Causal Language Modeling (e.g., GPT-style):**

$$\mathcal{L}_{\text{LM}}(\theta) = - \sum_{i=1}^n \log p_{\theta}(w_{i+1} \mid w_1, \dots, w_i),$$

where $p_{\theta}(w_{i+1} \mid w_1, \dots, w_i)$ is computed via softmax over vocabulary logits.

- Masked Language Modeling (e.g., BERT-style):

$$\mathcal{L}_{\text{MLM}}(\theta) = - \sum_{i \in \mathcal{M}} \log p_{\theta}(w_i | \tilde{w}),$$

where $\mathcal{M}$ is the set of masked token positions and $\tilde{w}$ denotes the input sequence with masks applied.

- **Input:** A large corpus of text.

- **Architecture:** A Transformer (decoder-only for GPT, encoder-only for BERT, or full encoder-decoder for models like T5) with parameters θ shared across all layers.

Notes

[illegible]

Pretraining and Fine-Tuning: Technical Setup II

Fine-Tuning Stage (Supervised)

- **Objective:** Adapt the pretrained Transformer to a downstream task (e.g., text classification, translation, question answering) using a labeled dataset.

- **Task-Specific Head:**

- For classification, add a linear layer with parameters $\mathbf{W}_{\text{cls}} \in \mathbb{R}^{d \times C}$ and bias $\mathbf{b}_{\text{cls}} \in \mathbb{R}^C$, where C is the number of classes.

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{y} \mathbf{W}_{\text{cls}} + \mathbf{b}_{\text{cls}}),$$

with y being the final hidden state (often corresponding to a special [CLS] token).

- For translation, the encoder-decoder architecture is used and cross-attention is added; the loss remains cross-entropy on the target sequence.

- **Fine-Tuning Loss:** Typically, a supervised cross-entropy loss is used:

$$\mathcal{L}_{\text{FT}}(\theta, \theta_{\text{head}}) = - \sum_i \log p_{\theta, \theta_{\text{head}}}(y_i | x_i),$$

where x_i is the input and y_i is the target label.

- Parameters θ are initialized from the pretrained model.

- The task-specific head parameters θ_{head} are initialized randomly.

Notes

[illegible]

Outline

- Recap and Motivation
- Expanding RNN Memory Beyond a Single Hidden State
- Attention Mechanisms
- Transformer Architecture for Language Modeling
- Training Transformer Models
- Pretraining and Fine-Tuning Transformers
- Transformer Setup Variants: GPT, Full Transformer, and BERT

Notes

Three Transformer Setups I

Decoder-Only Transformers: GPT Family

- **Architecture:**
 - Uses a *decoder-only* Transformer with masked self-attention.
 - Input: a sequence of token embeddings $\mathbf{X} \in \mathbb{R}^{n \times d}$ (with positional encodings added).
 - **Mask:** Enforces causal (left-to-right) attention:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top + \mathbf{M}}{\sqrt{d_k}}\right) \mathbf{V},$$

where $\mathbf{M}[i, j] = 0$ for $j \leq i$ and $-\infty$ for $j > i$.

- **Objective:** Autoregressive language modeling.

$$\mathcal{L}_{\text{LM}}(\boldsymbol{\theta}) = - \sum_{i=1}^n \log p_{\boldsymbol{\theta}}(w_{i+1} \mid w_1, \dots, w_i).$$

- **Key Points:**
 - All computations are parallelizable over sequence positions, except for the causal masking.
 - Suitable for large-scale pretraining and text generation.

Notes

Three Transformer Setups II

Full Transformer (Encoder–Decoder)

- **Architecture:**
 - Consists of an **Encoder** and a **Decoder**.
 - **Encoder:** Processes source sequence $\mathbf{X}^{\text{src}} \in \mathbb{R}^{n \times d}$ with self-attention (unmasked).
 - **Decoder:** Processes target sequence $\mathbf{X}^{\text{trg}} \in \mathbb{R}^{m \times d}$ with *masked* self-attention, and attends to encoder outputs via cross-attention.
 - Encoder and decoder stacks are each built from residual-connected layers of multi-head self-attention and FFN.
- **Objective:** Sequence-to-sequence learning (e.g., for translation):

$$\mathcal{L}_{\text{seq2seq}}(\theta) = - \sum_{i=1}^m \log p_{\theta} \left(w_i^{\text{trg}} \mid w_1^{\text{trg}}, \dots, w_{i-1}^{\text{trg}}, \mathbf{H}^{\text{enc}} \right).$$

where $\mathbf{H}^{\text{enc}}$ are the encoder outputs.

- Enables **contextualized encoding** of the source and dynamic alignment during decoding.
- Widely used for tasks like machine translation and summarization.

Notes

Three Transformer Setups III

Encoder-Only Transformers: BERT

- **Architecture:**
 - Uses only the **encoder** part of the Transformer.
 - Processes a full input sequence $\mathbf{X} \in \mathbb{R}^{n \times d}$ with self-attention (unmasked).
 - Positional encodings are added to maintain token order.
- **Pretraining Objectives:**
 - **Masked Language Modeling (MLM):** Randomly mask some tokens and predict them.

$$\mathcal{L}_{\text{MLM}}(\theta) = - \sum_{i \in \mathcal{M}} \log p_{\theta} \left(w_i \mid \tilde{w} \right),$$

where $\mathcal{M}$ is the set of masked token indices.

- **Fine-Tuning:** Adapt the pretrained encoder for downstream tasks (e.g., classification, question answering) by adding a task-specific head.

Notes
