Generative Models for NLP

Attention Mechanisms and Transformer Architectures

Nadi Tomeh

7/2/25

Notes

Outline

Notes

Outline

Notes

Recap of RNN-Based Language Models

Quick Review

- **RNN/GRU/LSTM Architectures**: These architectures process sequences token by token, updating a hidden state $\boldsymbol{h}_t$ at each step t.
  - $\boldsymbol{h}_t$ captures the information from all previously seen tokens.
  - GRUs and LSTMs introduce gating mechanisms to mitigate vanishing or exploding gradients.
- **Hidden State $\boldsymbol{h}_t$**:
  - Serves as a summary (or *memory*) of the sequence up to position t.
  - Used for predicting the next token in a language modeling setup:

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

  where $\boldsymbol{h}_t$ 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 $\boldsymbol{h}_t \in \mathbb{R}^m$ at time t is defined by a recurrence of the form:

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

Forward Pass Constraint

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

$$\boldsymbol{h}_1 \to \boldsymbol{h}_2 \to \cdots \to \boldsymbol{h}_t.$$

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

BPTT Perspective

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

$$\frac{\partial \ell}{\partial \boldsymbol{h}_{t-1}} \;=\; \frac{\partial \ell}{\partial \boldsymbol{h}_t}\,\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}} \;+\; \cdots$$

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

Notes

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

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

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

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

  capturing *all* past inputs $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t\}$ through a single vector.
- Intuitively, $\boldsymbol{h}_t$ serves as the network's *internal memory*, summarizing prior context relevant for predicting future tokens.

Markovian and Geometric Ergodicity

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

$$p\big(\boldsymbol{h}_t \mid \boldsymbol{h}_{t-1}, \boldsymbol{h}_{t-2}, \ldots\big) \;=\; p\big(\boldsymbol{h}_t \mid \boldsymbol{h}_{t-1}\big).$$

- 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 $\boldsymbol{h}_0$ and $\boldsymbol{h}'_0$, we have

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

- The result is **Exponential Forgetting**: The influence of initial states $\boldsymbol{h}_0$ vanishes at a rate λ^t.
- **Short Memory**: As $t \to \infty$, $\boldsymbol{h}_t$ converges to a stationary regime, losing detailed information about very distant past.

Notes

Outline

Notes

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

Recurrence in a Standard Elman RNN LM

- Hidden state update:

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

- Prediction logits:

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

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

The Bottleneck

- $\boldsymbol{h}_t$ must encode *all* relevant history in a single vector of size m.
- As t grows, $\boldsymbol{h}_t$ struggles to maintain detailed information about very distant tokens.
- This can degrade the accuracy of $\boldsymbol{z}_t$ (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 $\boldsymbol{h}_t$, we keep each past hidden state:

$$\mathcal{M}_t = \{\boldsymbol{h}_1, \boldsymbol{h}_2, \ldots, \boldsymbol{h}_t\}.$$

- Each $\boldsymbol{h}_\tau \in \mathbb{R}^m$ 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 $\boldsymbol{c}_t$

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

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

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

Notes

Combining the Memory Vectors $\mathcal{M}_t = \{\boldsymbol{h}_1, \ldots, \boldsymbol{h}_t\}$ into a Context Vector I

Naive Summation or Averaging

- **Sum or average:**

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

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

Hard Selection (One-Hot or Multi-Hot)

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

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

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

Notes

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

Differentiable Soft Selection

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

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

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

Notes

Outline

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 $\boldsymbol{h}_\tau$ (for $\tau = 1, \ldots, t-1$) contributes to the current context.
- Define a **score function** $e_{t,\tau}$ that measures the similarity between $\boldsymbol{h}_t$ (the **"query"**) and $\boldsymbol{h}_\tau$ (the **"key"**).

$$s_{t,\tau} \;=\; \mathrm{sim}\big(\boldsymbol{h}_t,\; \boldsymbol{h}_\tau\big).$$

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

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

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

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

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

Notes

Four Common Similarity (Score) Functions I

1. Dot Product

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

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

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

2. Scaled Dot Product (Vaswani)

$$s_{t,\tau} \;=\; \frac{\boldsymbol{h}_t^\top\,\boldsymbol{h}_\tau}{\sqrt{m}},$$

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

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

Notes

Four Common Similarity (Score) Functions II

3. Bilinear (Luong Attention)

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

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

4. MLP (Additive / Bahdanau Attention)

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

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

- Uses a small neural network for scoring each pair $(\boldsymbol{h}_t, \boldsymbol{h}_\tau)$.
- 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)**: $\big(w_1^{\mathrm{src}}, \ldots, w_n^{\mathrm{src}}\big)$.
- **Output (target sequence)**: $\big(w_1^{\mathrm{trg}}, \ldots, w_m^{\mathrm{trg}}\big)$.
- We want to learn $p_\theta(w_1^{\mathrm{trg}}, \ldots, w_m^{\mathrm{trg}} \mid w_1^{\mathrm{src}}, \ldots, w_n^{\mathrm{src}})$, a *conditional* generative model.

Encoder–Decoder Architecture

- **Encoder (RNN)**: Processes the source tokens into hidden states $\{\boldsymbol{h}_1^{\mathrm{enc}}, \ldots, \boldsymbol{h}_n^{\mathrm{enc}}\}$.
- **Decoder (RNN)**: Generates target tokens $\big(w_t^{\mathrm{trg}}\big)$ 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:

$$\boldsymbol{h}_{\mathrm{context}} = \boldsymbol{h}_n^{\mathrm{enc}}.$$

- **Limitation**: A single fixed-size vector $\boldsymbol{h}_n^{\mathrm{enc}}$ 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 $\boldsymbol{h}_n^{\mathrm{enc}}$ alone, maintain a memory of $\{\boldsymbol{h}_1^{\mathrm{enc}}, \ldots, \boldsymbol{h}_n^{\mathrm{enc}}\}$.
- At each decoder timestep t:

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

  where $\boldsymbol{h}_t^{\mathrm{dec}} \in \mathbb{R}^m$ is the current decoder hidden state.
- The $\alpha_{t,\tau}$ measure how relevant the encoder's state $\boldsymbol{h}_\tau^{\mathrm{enc}}$ is at decoding step t.

Context-Augmented Decoder State

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

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

- **Result**: A *trainable* **alignment matrix** $\boldsymbol{\alpha}$ via attention, letting the model focus on relevant source positions for each target word.

Notes

Outline

Notes

Overview and Motivation

Goal: Autoregressive Language Modeling

- We want a next-token distribution:
$$p_\theta(w_{t+1} \mid w_1, \ldots, w_t),$$
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 $\boldsymbol{x}_i \in \mathbb{R}^d$, for $i = 1, \ldots, n$.

- **Per-token** query, key, value:
$$\boldsymbol{q}_i = \boldsymbol{W}^Q \boldsymbol{x}_i \in \mathbb{R}^{d_k}, \quad \boldsymbol{k}_j = \boldsymbol{W}^K \boldsymbol{x}_j \in \mathbb{R}^{d_k}, \quad \boldsymbol{v}_j = \boldsymbol{W}^V \boldsymbol{x}_j \in \mathbb{R}^{d_v},$$
where $\boldsymbol{W}^Q, \boldsymbol{W}^K \in \mathbb{R}^{d \times d_k}$, $\boldsymbol{W}^V \in \mathbb{R}^{d \times d_v}$.
- **Scores and softmax:**
$$s_{i,j} = \frac{\boldsymbol{q}_i^\top \boldsymbol{k}_j}{\sqrt{d_k}}, \quad \alpha_{i,j} = \frac{\exp(s_{i,j})}{\sum_{m=1}^n \exp(s_{i,m})}.$$
- **Attention output for token i:**
$$\boldsymbol{y}_i = \sum_{j=1}^n \alpha_{i,j} \boldsymbol{v}_j.$$

Notes

Masked Multi-Head Self-Attention II

Matrix Form Equations (All Tokens in Parallel)

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

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

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

Each row of the result is $\boldsymbol{y}_i^\top$, matching the per-token outputs $\boldsymbol{y}_i$ 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 $\boldsymbol{M} \in \mathbb{R}^{n\times n}$:

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

- Then:

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

- 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 W_i^Q, W_i^K, W_i^V for $i = 1, \ldots, h$.

$$\text{head}_i = \mathrm{Attention}\Big(X\,W_i^Q,\; X\,W_i^K,\; X\,W_i^V\Big) \;\in \mathbb{R}^{n \times d_v}.$$

- **Concatenate** heads:

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

- **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 X just permutes the rows of Q, K, 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 $\mathrm{PE}(i) \in \mathbb{R}^d$.
- Then we **add** $\mathrm{PE}(i)$ to the original token embedding x_i:

$$x_i' = x_i + \mathrm{PE}(i).$$

- The model's self-attention layers now see x_i', which encodes both the token's identity *and* its position.

Learned vs. Sinusoidal

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

Notes

Sinusoidal Positional Encodings I

Formula

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

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

Why Sinusoids?

- **Relative Offsets**: $\mathrm{PE}(pos_2) - \mathrm{PE}(pos_1)$ 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:

$$\mathrm{PE}(pos_2) - \mathrm{PE}(pos_1) = 2\cos\left(\frac{pos_1 + pos_2}{2 \cdot 10000^{2j/d}}\right)\sin\left(\frac{pos_2 - pos_1}{2 \cdot 10000^{2j/d}}\right).$$

- 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:

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

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

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

- **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

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

- $\mu(\boldsymbol{x}) = \frac{1}{d}\sum_{i=1}^{d} x_i$ is the **mean** of $\boldsymbol{x}$.
- $\sigma(\boldsymbol{x}) = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu(\boldsymbol{x}))^2}$ is the **standard deviation** of $\boldsymbol{x}$.
- $\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^d$ 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:

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

- 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.

Notes

Position-Wise Feed-Forward Network (FFN)

Position-Wise MLP

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

$$\mathrm{FFN}(\boldsymbol{x}) = \max\big(0,\ \boldsymbol{x}\,\boldsymbol{W}_1 + \boldsymbol{b}_1\big)\ \boldsymbol{W}_2 + \boldsymbol{b}_2,$$

where:

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

Shape & Per-Token Independence

- Input to FFN layer: $\boldsymbol{H} \in \mathbb{R}^{n \times d}$, where n is the number of tokens.
- We apply FFN *row by row*, i.e. each $\boldsymbol{h}_i \in \mathbb{R}^d$ (the i-th token's vector) is mapped to another $\boldsymbol{h}'_i \in \mathbb{R}^d$.
- $\max(0, \cdot)$ 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.

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
$$\boldsymbol{X}^{(0)} = \boldsymbol{X} + \mathrm{PE} \quad \in \mathbb{R}^{n \times d},$$
where $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ are token embeddings and $\mathrm{PE} \in \mathbb{R}^{n \times d}$ are positional encodings.
- Each decoder layer l (for $l = 1, \ldots, L$) is a composition of sublayers:
$$\boldsymbol{X}^{(l)} = \mathrm{LayerNorm}\Big(\boldsymbol{X}^{(l-1)} + \mathrm{FFN}\Big(\mathrm{LayerNorm}\big(\boldsymbol{X}^{(l-1)} + \mathrm{MaskedMHA}(\boldsymbol{X}^{(l-1)})\big)\Big)\Big).$$

Notes

Putting It All Together: Composing Transformer Sublayers for LM II

Output Projection to Vocabulary Distribution

- After L layers, we obtain final representations:
$$\boldsymbol{Y} = \boldsymbol{X}^{(L)} \in \mathbb{R}^{n \times d}.$$
- For each token (row) $\boldsymbol{y}_i \in \mathbb{R}^d$ in $\boldsymbol{Y}$, compute logits:
$$\boldsymbol{z}_i = \boldsymbol{y}_i \boldsymbol{W}_{\mathrm{out}} + \boldsymbol{b}_{\mathrm{out}}, \quad \boldsymbol{W}_{\mathrm{out}} \in \mathbb{R}^{d \times |\mathcal{V}|}, \quad \boldsymbol{b}_{\mathrm{out}} \in \mathbb{R}^{|\mathcal{V}|}.$$
- Apply softmax to obtain the next-token probability distribution:
$$p_{\boldsymbol{\theta}}\big(w_i \mid w_1, \ldots, w_{i-1}\big) = \boldsymbol{p}_i = \mathrm{softmax}(\boldsymbol{z}_i) \in \mathbb{R}^{|\mathcal{V}|}.$$

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:

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

where $p_{\boldsymbol{\theta}}(w_{i+1} \mid w_1, \ldots, w_i)$ 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:

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

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 $\boldsymbol{z} \in \mathbb{R}^d$, dropout applies a mask $\boldsymbol{m} \in \{0,1\}^d$ with

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

and outputs

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

- 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:

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

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 $\boldsymbol{X} \in \mathbb{R}^{B\times n\times d}$.
- The Transformer computes outputs $\boldsymbol{Y} \in \mathbb{R}^{B\times n\times d}$ in parallel for every position.
- The predicted logits $\boldsymbol{Z} \in \mathbb{R}^{B\times n\times |\mathcal{V}|}$ are computed via:

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

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

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: $\boldsymbol{X} \in \mathbb{R}^{B\times n\times d}$.
- **Self-Attention Computations:**
  - For each layer, self-attention requires computing the matrix product $\boldsymbol{Q}\,\boldsymbol{K}^\top$ with cost:

$$\mathcal{O}\Big(B \cdot n^2 \cdot d_k\Big),$$

    where $\boldsymbol{Q}, \boldsymbol{K} \in \mathbb{R}^{B\times n\times d_k}$.
  - Additional matrix multiplications (e.g., with $\boldsymbol{V}$) 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.

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: $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$ with recurrence.
- **Per-Step Cost:**
  - Each time step involves computing:
  $$\boldsymbol{h}_t = f_{\boldsymbol{\theta}}(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t),$$
  with cost $\mathcal{O}(f(d))$ per step.
- **Overall Training:**
  - Total cost per sequence: $\mathcal{O}\big(n \cdot f(d)\big)$.
  - 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

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):**

$$\mathcal{L}_{\mathsf{LM}}(\boldsymbol{\theta}) = -\sum_{i=1}^{n} \log p_{\boldsymbol{\theta}}\Big(w_{i+1} \,\Big|\, w_1, \ldots, w_i\Big),$$

    where $p_{\boldsymbol{\theta}}(w_{i+1} \mid w_1, \ldots, w_i)$ is computed via softmax over vocabulary logits.
  - **Masked Language Modeling (e.g., BERT-style):**

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

    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 $\boldsymbol{\theta}$ shared across all layers.

Notes

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 $\boldsymbol{W}_{\mathsf{cls}} \in \mathbb{R}^{d \times C}$ and bias $\boldsymbol{b}_{\mathsf{cls}} \in \mathbb{R}^{C}$, where C is the number of classes.

$$\hat{\boldsymbol{y}} = \mathrm{softmax}\Big(\boldsymbol{y}\, \boldsymbol{W}_{\mathsf{cls}} + \boldsymbol{b}_{\mathsf{cls}}\Big),$$

    with $\boldsymbol{y}$ 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:

$$\mathcal{L}_{\mathsf{FT}}(\boldsymbol{\theta}, \boldsymbol{\theta}_{\mathsf{head}}) = -\sum_{i} \log p_{\boldsymbol{\theta}, \boldsymbol{\theta}_{\mathsf{head}}}\Big(y_i \,\Big|\, x_i\Big),$$

  where x_i is the input and y_i is the target label.
- Parameters $\boldsymbol{\theta}$ are initialized from the pretrained model.
- The task-specific head parameters $\boldsymbol{\theta}_{\mathsf{head}}$ are initialized randomly.

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

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 $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ (with positional encodings added).
  - **Mask:** Enforces causal (left-to-right) attention:

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

  where $\boldsymbol{M}[i,j] = 0$ for $j \leq i$ and $-\infty$ for $j > i$.
- **Objective:** Autoregressive language modeling.

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

- **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 $\boldsymbol{X}^{\text{src}} \in \mathbb{R}^{n \times d}$ with self-attention (unmasked).
  - **Decoder:** Processes target sequence $\boldsymbol{X}^{\text{trg}} \in \mathbb{R}^{m \times d}$ 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):

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

  where $\boldsymbol{H}^{\text{enc}}$ 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 $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ 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.

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

    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