Introduction to Large Language Models

Changlong Wu & Wojciech Szpankowski

Center for Science of Information
Purdue University

October 25, 2024

Overview

- **The Next-Token Prediction Paradigm**
  - Stochastic modeling of languages
  - Conditional distribution estimation
- **The Transformer Architecture**
  - Attention mechanism: Self-attention and multi-head attention
  - Positional encoding and why it's needed
  - Layer structure: Encoder vs. decoder
- **Generative Pre-trained Transformer (GPT) Models**
  - Pretraining: Learning from large, diverse datasets
  - Fine-tuning: Specializing GPT for specific tasks
  - GPT-3 and beyond: Scaling and capabilities

Shannon's Stochastic Approximation of English

Claude E. Shannon, "A Mathematical Theory of Communication", 1948

3. THE SERIES OF APPROXIMATIONS TO ENGLISH

To give a visual idea of how this series of processes approaches a language, typical sequences in the approximations to English have been constructed and are given below. In all cases we have assumed a 27-symbol "alphabet," the 26 letters and a space.

1. Zero-order approximation (symbols independent and equiprobable).

    XFOML RXKHRJFFJUJ ZLPWCFWKCYJ FFJEYVKCQSGHYD QPAAMKBZAACIBZL-HJQD.

- "... a discrete source as generating the message, symbol by symbol. It will choose successive symbols according to certain probabilities depending..."
- Shannon also introduces the "n-gram" model, where one models the conditional probability of next word depending on its previous *n* words.

What is a Language Model?

Let $\mathcal{Y}$ be a set of tokens (which may include symbols, words, subwords...), which is the basic building block of language models.

What is a Language Model?

Let $\mathcal{Y}$ be a set of tokens (which may include symbols, words, subwords...), which is the basic building block of language models.

A language model P is a probability distribution over $\mathcal{Y}^*$.

What is a Language Model?

Let $\mathcal{Y}$ be a set of tokens (which may include symbols, words, subwords...), which is the basic building block of language models.

A language model P is a probability distribution over $\mathcal{Y}^*$.

It is typically represented by specifying the conditional distribution:

$$P(y_t \mid y^{t-1}), \text{ for } y^{t-1} \in \mathcal{Y}^*.$$

What is a Language Model?

Let $\mathcal{Y}$ be a set of tokens (which may include symbols, words, subwords...), which is the basic building block of language models.

A language model P is a probability distribution over $\mathcal{Y}^*$.

It is typically represented by specifying the conditional distribution:

$$P(y_t \mid y^{t-1}), \text{ for } y^{t-1} \in \mathcal{Y}^*.$$

The probability assigned to a sentence $y^T \in \mathcal{Y}^*$ is given by

$$P(y^T) = \prod_{t=1}^{T} P(y_t \mid y^{t-1}).$$

What is a Language Model?

Let $\mathcal{Y}$ be a set of tokens (which may include symbols, words, subwords...), which is the basic building block of language models.

A language model P is a probability distribution over $\mathcal{Y}^*$.

It is typically represented by specifying the conditional distribution:

$$P(y_t \mid y^{t-1}), \text{ for } y^{t-1} \in \mathcal{Y}^*.$$

The probability assigned to a sentence $y^T \in \mathcal{Y}^*$ is given by

$$P(y^T) = \prod_{t=1}^{T} P(y_t \mid y^{t-1}).$$

Goal: Find an algorithm for computing the conditional distribution $P(\cdot \mid y^t)$.

Learning Language Models

Let $\mathcal{H} := \{h_\theta : \theta \in \Theta\} \subset \Delta(\mathcal{Y})^{\mathcal{Y}^*}$ be a hypothesis class of language models.

Learning Language Models

Let $\mathcal{H} := \{h_\theta : \theta \in \Theta\} \subset \Delta(\mathcal{Y})^{\mathcal{Y}^*}$ be a hypothesis class of language models.

We interpret $h_\theta(y^{t-1}) \in \Delta(\mathcal{Y})$ as the conditional distribution on the past y^{t-1}.

Learning Language Models

Let $\mathcal{H} := \{h_\theta : \theta \in \Theta\} \subset \Delta(\mathcal{Y})^{\mathcal{Y}^*}$ be a hypothesis class of language models.

We interpret $h_\theta(y^{t-1}) \in \Delta(\mathcal{Y})$ as the conditional distribution on the past y^{t-1}.

Let $y^T \in \mathcal{Y}^*$ be the training data, the Maximum Likelihood Estimation (MLE) is given by:

$$\hat{\theta} := \arg\max_{\theta \in \Theta} \prod_{t=1}^{T} h_\theta(y^{t-1})[y_t]$$

Learning Language Models

Let $\mathcal{H} := \{h_\theta : \theta \in \Theta\} \subset \Delta(\mathcal{Y})^{\mathcal{Y}^*}$ be a hypothesis class of language models.

We interpret $h_\theta(y^{t-1}) \in \Delta(\mathcal{Y})$ as the conditional distribution on the past y^{t-1}.

Let $y^T \in \mathcal{Y}^*$ be the training data, the Maximum Likelihood Estimation (MLE) is given by:

$$\hat{\theta} := \arg\max_{\theta \in \Theta} \prod_{t=1}^{T} h_\theta(y^{t-1})[y_t]$$

Equivalently, we can express this as:

$$\begin{aligned}\hat{\theta} &= \arg\min_{\theta \in \Theta} \sum_{t=1}^{T} -\log h_\theta(y^{t-1})[y_t] \\ &= \arg\min_{\theta \in \Theta} \sum_{t=1}^{T} \ell^{\log}(h_\theta(y^{t-1}), y_t).\end{aligned}$$

Learning Language Models

Let $\mathcal{H} := \{h_\theta : \theta \in \Theta\} \subset \Delta(\mathcal{Y})^{\mathcal{Y}^*}$ be a hypothesis class of language models.

We interpret $h_\theta(y^{t-1}) \in \Delta(\mathcal{Y})$ as the conditional distribution on the past y^{t-1}.

Let $y^T \in \mathcal{Y}^*$ be the training data, the Maximum Likelihood Estimation (MLE) is given by:

$$\hat{\theta} := \arg\max_{\theta \in \Theta} \prod_{t=1}^{T} h_\theta(y^{t-1})[y_t]$$

Equivalently, we can express this as:

$$\begin{aligned}\hat{\theta} &= \arg\min_{\theta \in \Theta} \sum_{t=1}^{T} -\log h_\theta(y^{t-1})[y_t] \\ &= \arg\min_{\theta \in \Theta} \sum_{t=1}^{T} \ell^{\log}(h_\theta(y^{t-1}), y_t).\end{aligned}$$

Here, $\ell^{\log}(p, y)$ refers to the logarithmic loss (see **Lecture 4**).

Learning Language Models

Let $\mathcal{H} := \{h_\theta : \theta \in \Theta\} \subset \Delta(\mathcal{Y})^{\mathcal{Y}^*}$ be a hypothesis class of language models.

We interpret $h_\theta(y^{t-1}) \in \Delta(\mathcal{Y})$ as the conditional distribution on the past y^{t-1}.

Let $y^T \in \mathcal{Y}^*$ be the training data, the Maximum Likelihood Estimation (MLE) is given by:

$$\hat{\theta} := \arg\max_{\theta \in \Theta} \prod_{t=1}^{T} h_\theta(y^{t-1})[y_t]$$

Equivalently, we can express this as:

$$\begin{aligned} \hat{\theta} &= \arg\min_{\theta \in \Theta} \sum_{t=1}^{T} -\log h_\theta(y^{t-1})[y_t] \\ &= \arg\min_{\theta \in \Theta} \sum_{t=1}^{T} \ell^{\log}(h_\theta(y^{t-1}), y_t). \end{aligned}$$

Here, $\ell^{\log}(p, y)$ refers to the logarithmic loss (see **Lecture 4**).

The MLE $\hat{\theta}$ is typically computed via gradient based algorithms (e.g., SGD).

Choosing the Hypothesis Classes

The key to learning a language model is choosing an appropriate architecture for the hypothesis class $\mathcal{H}$.

Choosing the Hypothesis Classes

The key to learning a language model is choosing an appropriate architecture for the hypothesis class $\mathcal{H}$.

Major Architectures Include:

- **n-gram Models**: Use fixed-length context of $n - 1$ words.
- **Recurrent Neural Networks (RNNs)**: Process sequences word-by-word, capturing temporal dependencies.
- **Long Short-Term Memory (LSTM)**: A type of RNN designed to capture long-range dependencies.
- **Transformer Models**: Use self-attention to process sequences in parallel (e.g., GPT, BERT).

Choosing the Hypothesis Classes

The key to learning a language model is choosing an appropriate architecture for the hypothesis class $\mathcal{H}$.

Major Architectures Include:

- **n-gram Models**: Use fixed-length context of $n - 1$ words.
- **Recurrent Neural Networks (RNNs)**: Process sequences word-by-word, capturing temporal dependencies.
- **Long Short-Term Memory (LSTM)**: A type of RNN designed to capture long-range dependencies.
- **Transformer Models**: Use self-attention to process sequences in parallel (e.g., GPT, BERT).

This lecture will focus entirely on the transformer-based models.

Overview

- **The Next-Token Prediction Paradigm**
  - Stochastic modeling of languages
  - Conditional distribution estimation
- **The Transformer Architecture**
  - Vector embedding
  - Attention mechanism: Self-attention and multi-head attention
  - Positional encoding and why it's needed
- **Generative Pre-trained Transformer (GPT) Models**
  - Pretraining: Learning from large, diverse datasets
  - Fine-tuning: Specializing GPT for specific tasks
  - GPT-3 and beyond: Scaling and capabilities

The Transformer Architecture

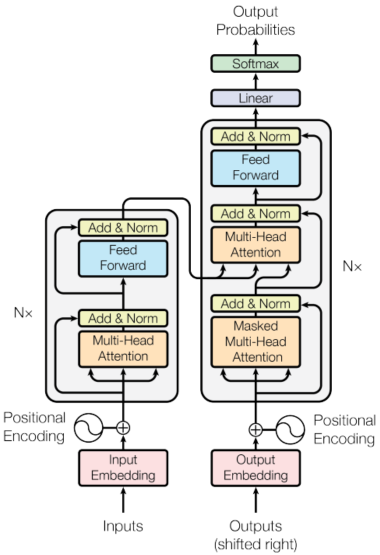


Figure 1: The Transformer - model architecture.

1. The input to the transformer is a sequence of tokens.
   - These tokens are obtained through a non-learnable tokenization process.
2. The tokens are mapped to vectors via a learnable embedding layer.
3. These vectors are combined with non-learnable positional encoding.
4. The processed vectors are passed through N repeated layers.
   - Each layer includes multi-head attention and a feed-forward MLP.
   - Each layer transforms the vectors into another of the same shape.
5. The output is a probability distribution over all possible tokens.

The Vector Embedding

Let $\mathcal{Y}$ be the set of all tokens. For example, in GPT-3, we have $|\mathcal{Y}| \sim 50,000$.

The Vector Embedding

Let $\mathcal{Y}$ be the set of all tokens. For example, in GPT-3, we have $|\mathcal{Y}| \sim 50,000$.

The embedding layer is represented by a large matrix, $M \in \mathbb{R}^{d \times |\mathcal{Y}|}$.

The Vector Embedding

Let $\mathcal{Y}$ be the set of all tokens. For example, in GPT-3, we have $|\mathcal{Y}| \sim 50,000$.

The embedding layer is represented by a large matrix, $M \in \mathbb{R}^{d \times |\mathcal{Y}|}$.

Here, d is the embedding dimension (e.g., for GPT-3, $d = 12,288$).

The Vector Embedding

Let $\mathcal{Y}$ be the set of all tokens. For example, in GPT-3, we have $|\mathcal{Y}| \sim 50,000$.

The embedding layer is represented by a large matrix, $M \in \mathbb{R}^{d \times |\mathcal{Y}|}$.

Here, d is the embedding dimension (e.g., for GPT-3, $d = 12,288$).

For any token $y \in \mathcal{Y}$, let $e_y \in \mathbb{R}^{\mathcal{Y}}$ be the standard basis vector, where:

$$e_y[y'] = \begin{cases} 1, & \text{if } y = y' \\ 0, & \text{otherwise} \end{cases}.$$

The Vector Embedding

Let $\mathcal{Y}$ be the set of all tokens. For example, in GPT-3, we have $|\mathcal{Y}| \sim 50,000$.

The embedding layer is represented by a large matrix, $M \in \mathbb{R}^{d \times |\mathcal{Y}|}$.

Here, d is the embedding dimension (e.g., for GPT-3, $d = 12,288$).

For any token $y \in \mathcal{Y}$, let $e_y \in \mathbb{R}^{\mathcal{Y}}$ be the standard basis vector, where:

$$e_y[y'] = \begin{cases} 1, & \text{if } y = y' \\ 0, & \text{otherwise} \end{cases}.$$

The embedding layer maps each token $y \in \mathcal{Y}$ to a vector $\mathbf{x}_y$ (viewed vertically) as follows:

$$\mathbf{x}_y := Me_y \in \mathbb{R}^d.$$

The Vector Embedding

Let $\mathcal{Y}$ be the set of all tokens. For example, in GPT-3, we have $|\mathcal{Y}| \sim 50{,}000$.

The embedding layer is represented by a large matrix, $M \in \mathbb{R}^{d \times |\mathcal{Y}|}$.

Here, d is the embedding dimension (e.g., for GPT-3, $d = 12{,}288$).

For any token $y \in \mathcal{Y}$, let $e_y \in \mathbb{R}^{\mathcal{Y}}$ be the standard basis vector, where:

$$e_y[y'] = \begin{cases} 1, & \text{if } y = y' \\ 0, & \text{otherwise} \end{cases}.$$

The embedding layer maps each token $y \in \mathcal{Y}$ to a vector $\mathbf{x}_y$ (viewed vertically) as follows:

$$\mathbf{x}_y := M e_y \in \mathbb{R}^d.$$

The matrix M is part of the model's parameters in a Transformer architecture and will be updated during training.

Multi-Head Attention Layer

Let $y_1, \cdots, y_t$ be a set of input tokens, and let $\mathbf{x}_1, \cdots, \mathbf{x}_t \in \mathbb{R}^d$ be the corresponding embedding vectors (after position embedding).

Multi-Head Attention Layer

Let $y_1, \cdots, y_t$ be a set of input tokens, and let $\mathbf{x}_1, \cdots, \mathbf{x}_t \in \mathbb{R}^d$ be the corresponding embedding vectors (after position embedding).

Denote $X := [\mathbf{x}_1, \cdots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$.

Multi-Head Attention Layer

Let $y_1, \cdots, y_t$ be a set of input tokens, and let $\mathbf{x}_1, \cdots, \mathbf{x}_t \in \mathbb{R}^d$ be the corresponding embedding vectors (after position embedding).

Denote $X := [\mathbf{x}_1, \cdots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$.

The multi-head attention layer consists of h attention heads, where h divides d.

Multi-Head Attention Layer

Let $y_1, \cdots, y_t$ be a set of input tokens, and let $\mathbf{x}_1, \cdots, \mathbf{x}_t \in \mathbb{R}^d$ be the corresponding embedding vectors (after position embedding).

Denote $X := [\mathbf{x}_1, \cdots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$.

The multi-head attention layer consists of h attention heads, where h divides d.

Each attention head i is associated with three matrices $W_k^i, W_q^i, W_v^i \in \mathbb{R}^{d \times d'}$, where $d' = d/h$.

Multi-Head Attention Layer

Let $y_1, \cdots, y_t$ be a set of input tokens, and let $\mathbf{x}_1, \cdots, \mathbf{x}_t \in \mathbb{R}^d$ be the corresponding embedding vectors (after position embedding).

Denote $X := [\mathbf{x}_1, \cdots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$.

The multi-head attention layer consists of h attention heads, where h divides d.

Each attention head i is associated with three matrices $W_k^i, W_q^i, W_v^i \in \mathbb{R}^{d \times d'}$, where $d' = d/h$.

These matrices are then used to transform X into a matrix $X_i \in \mathbb{R}^{t \times d'}$ (to be explained in the **next slides**).

Multi-Head Attention Layer

Let $y_1, \cdots, y_t$ be a set of input tokens, and let $\mathbf{x}_1, \cdots, \mathbf{x}_t \in \mathbb{R}^d$ be the corresponding embedding vectors (after position embedding).

Denote $X := [\mathbf{x}_1, \cdots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$.

The multi-head attention layer consists of h attention heads, where h divides d.

Each attention head i is associated with three matrices $W_k^i, W_q^i, W_v^i \in \mathbb{R}^{d \times d'}$, where $d' = d/h$.

These matrices are then used to transform X into a matrix $X_i \in \mathbb{R}^{t \times d'}$ (to be explained in the **next slides**).

The outputs matrices are concatenated to form a matrix $[X_1, \cdots, X_h] \in \mathbb{R}^{t \times d}$.

Multi-Head Attention Layer

Let $y_1, \cdots, y_t$ be a set of input tokens, and let $\mathbf{x}_1, \cdots, \mathbf{x}_t \in \mathbb{R}^d$ be the corresponding embedding vectors (after position embedding).

Denote $X := [\mathbf{x}_1, \cdots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$.

The multi-head attention layer consists of h attention heads, where h divides d.

Each attention head i is associated with three matrices $W_k^i, W_q^i, W_v^i \in \mathbb{R}^{d \times d'}$, where $d' = d/h$.

These matrices are then used to transform X into a matrix $X_i \in \mathbb{R}^{t \times d'}$ (to be explained in the **next slides**).

The outputs matrices are concatenated to form a matrix $[X_1, \cdots, X_h] \in \mathbb{R}^{t \times d}$.

Then, a linear projection with matrix $W^O \in \mathbb{R}^{d \times d}$ is applied to obtain the final output:

$$X' := [X_1, \cdots, X_h] \, W^O \in \mathbb{R}^{t \times d}.$$

Multi-Head Attention Layer

Let $y_1, \cdots, y_t$ be a set of input tokens, and let $\mathbf{x}_1, \cdots, \mathbf{x}_t \in \mathbb{R}^d$ be the corresponding embedding vectors (after position embedding).

Denote $X := [\mathbf{x}_1, \cdots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$.

The multi-head attention layer consists of h attention heads, where h divides d.

Each attention head i is associated with three matrices $W_k^i, W_q^i, W_v^i \in \mathbb{R}^{d \times d'}$, where $d' = d/h$.

These matrices are then used to transform X into a matrix $X_i \in \mathbb{R}^{t \times d'}$ (to be explained in the **next slides**).

The outputs matrices are concatenated to form a matrix $[X_1, \cdots, X_h] \in \mathbb{R}^{t \times d}$.

Then, a linear projection with matrix $W^O \in \mathbb{R}^{d \times d}$ is applied to obtain the final output:

$$X' := [X_1, \cdots, X_h]\, W^O \in \mathbb{R}^{t \times d}.$$

Note that X' has the same shape as X and will be fed to the next layer.

Multi-Head Attention Layer

Let $y_1, \cdots, y_t$ be a set of input tokens, and let $\mathbf{x}_1, \cdots, \mathbf{x}_t \in \mathbb{R}^d$ be the corresponding embedding vectors (after position embedding).

Denote $X := [\mathbf{x}_1, \cdots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$.

The multi-head attention layer consists of h attention heads, where h divides d.

Each attention head i is associated with three matrices $W_k^i, W_q^i, W_v^i \in \mathbb{R}^{d \times d'}$, where $d' = d/h$.

These matrices are then used to transform X into a matrix $X_i \in \mathbb{R}^{t \times d'}$ (to be explained in the **next slides**).

The outputs matrices are concatenated to form a matrix $[X_1, \cdots, X_h] \in \mathbb{R}^{t \times d}$.

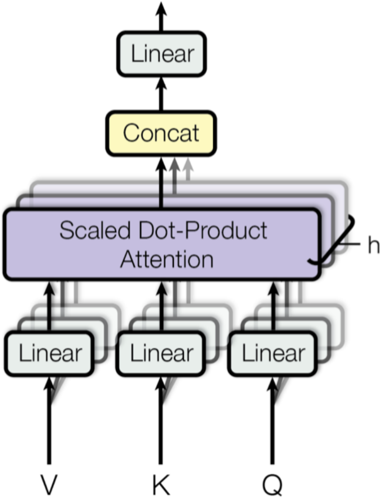
Then, a linear projection with matrix $W^O \in \mathbb{R}^{d \times d}$ is applied to obtain the final output:

$$X' := [X_1, \cdots, X_h]\, W^O \in \mathbb{R}^{t \times d}.$$

Note that X' has the same shape as X and will be fed to the next layer.

The matrices $\{W_k^i, W_q^i, W_v^i, W^O\}_{i \le h}$ are part of the trainable parameters.

Multi-Head Attention Layer



Scaled Dot-Product Attention

The attention head i transform input $X \in \mathbb{R}^{t \times d}$ into $X_i \in \mathbb{R}^{t \times d'}$ as follows:

1. Compute matrices (interpreted as query, key and value):

$$Q_i := XW_q^i, \quad K_i := XW_k^i, \quad V_i := XW_v^i, \quad \in \mathbb{R}^{t \times d'}.$$

2. Compute the scaled attention scores matrix

$$S_i := \frac{(Q_i K_i^\top)}{\sqrt{d'}} \in \mathbb{R}^{t \times t}.$$

3. Denote s_j as the jth row of S_i for $j \leq t$, which is interpreted as the attention scores for the jth token. The output is given by

$$X_i := \begin{bmatrix} \mathrm{softmax}(\mathrm{s}_1) \\ \cdots \\ \mathrm{softmax}(\mathrm{s}_t) \end{bmatrix} V_i \in \mathbb{R}^{t \times d'}$$

Here, for any $\mathbf{z} = (z_1, \cdots, z_t)$, $\mathrm{softmax}(\mathbf{z})$ corresponds to a vector $\mathbf{z}'$ such that

$$\forall j \in [t], \; z'_j := \frac{e^{z_j}}{\sum_{r=1}^{t} e^{z_r}}.$$

Scaled Dot-Product Attention

The attention head i transform input $X \in \mathbb{R}^{t \times d}$ into $X_i \in \mathbb{R}^{t \times d'}$ as follows:

1. Compute matrices (interpreted as query, key and value):

$$Q_i := XW_q^i, \quad K_i := XW_k^i, \quad V_i := XW_v^i, \quad \in \mathbb{R}^{t \times d'}.$$

2. Compute the scaled attention scores matrix

$$S_i := \frac{(Q_i K_i^\top)}{\sqrt{d'}} \in \mathbb{R}^{t \times t}.$$

3. Denote s_j as the jth row of S_i for $j \leq t$, which is interpreted as the attention scores for the jth token. The output is given by

$$X_i := \begin{bmatrix} \mathrm{softmax}(\mathrm{s}_1) \\ \cdots \\ \mathrm{softmax}(\mathrm{s}_t) \end{bmatrix} V_i \in \mathbb{R}^{t \times d'}$$

Here, for any $\mathbf{z} = (z_1, \cdots, z_t)$, softmax($\mathbf{z}$) corresponds to a vector $\mathbf{z}'$ such that

$$\forall j \in [t], \ z'_j := \frac{e^{z_j}}{\sum_{r=1}^{t} e^{z_r}}.$$

Goal: the attention head map a sequence of embeddings into another sequence of the same length (t) with improved representation incorporating the context.

Feed-Forward Layer

The feed-forward layer is a two-layer fully connected neural network with ReLU activation applied position-wise.

Feed-Forward Layer

The feed-forward layer is a two-layer fully connected neural network with ReLU activation applied position-wise.

Let $X = [\mathbf{x}_1, \cdots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$ be the input to the feed-forward layer.

The first layer computes outputs:

$$X' = \begin{bmatrix} \mathrm{ReLU}\left(\mathbf{x}_1^\top W_1 + \mathbf{b}_1^\top\right) \\ \vdots \\ \mathrm{ReLU}\left(\mathbf{x}_t^\top W_1 + \mathbf{b}_1^\top\right) \end{bmatrix} \in \mathbb{R}^{t \times d_{\mathrm{ff}}},$$

where $W_1 \in \mathbb{R}^{d \times d_{\mathrm{ff}}}$, $\mathbf{b}_1 \in \mathbb{R}^{d_{\mathrm{ff}}}$, and $\mathrm{ReLU}(x) := \max\{0, x\}$ is applied entry-wise.

Feed-Forward Layer

The feed-forward layer is a two-layer fully connected neural network with ReLU activation applied position-wise.

Let $X = [\mathbf{x}_1, \cdots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$ be the input to the feed-forward layer.

The first layer computes outputs:

$$X' = \begin{bmatrix} \mathrm{ReLU}\left(\mathbf{x}_1^\top W_1 + \mathbf{b}_1^\top\right) \\ \vdots \\ \mathrm{ReLU}\left(\mathbf{x}_t^\top W_1 + \mathbf{b}_1^\top\right) \end{bmatrix} \in \mathbb{R}^{t \times d_{\mathrm{ff}}},$$

where $W_1 \in \mathbb{R}^{d \times d_{\mathrm{ff}}}$, $\mathbf{b}_1 \in \mathbb{R}^{d_{\mathrm{ff}}}$, and $\mathrm{ReLU}(x) := \max\{0, x\}$ is applied entry-wise.

Denote $X' = [\mathbf{x}'_1, \cdots, \mathbf{x}'_t]^\top$. The final output is given by:

$$\begin{bmatrix} \mathbf{x}_1'^\top W_2 + \mathbf{b}_2^\top \\ \vdots \\ \mathbf{x}_t'^\top W_2 + \mathbf{b}_2^\top \end{bmatrix} \in \mathbb{R}^{t \times d},$$

where $W_2 \in \mathbb{R}^{d_{\mathrm{ff}} \times d}$, $\mathbf{b}_2 \in \mathbb{R}^{d}$.

Feed-Forward Layer

The feed-forward layer is a two-layer fully connected neural network with ReLU activation applied position-wise.

Let $X = [\mathbf{x}_1, \cdots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$ be the input to the feed-forward layer.

The first layer computes outputs:

$$X' = \begin{bmatrix} \mathrm{ReLU}\left(\mathbf{x}_1^\top W_1 + \mathbf{b}_1^\top\right) \\ \vdots \\ \mathrm{ReLU}\left(\mathbf{x}_t^\top W_1 + \mathbf{b}_1^\top\right) \end{bmatrix} \in \mathbb{R}^{t \times d_{\mathrm{ff}}},$$

where $W_1 \in \mathbb{R}^{d \times d_{\mathrm{ff}}}$, $\mathbf{b}_1 \in \mathbb{R}^{d_{\mathrm{ff}}}$, and $\mathrm{ReLU}(x) := \max\{0, x\}$ is applied entry-wise.

Denote $X' = [\mathbf{x}'_1, \cdots, \mathbf{x}'_t]^\top$. The final output is given by:

$$\begin{bmatrix} \mathbf{x}_1'^\top W_2 + \mathbf{b}_2^\top \\ \vdots \\ \mathbf{x}_t'^\top W_2 + \mathbf{b}_2^\top \end{bmatrix} \in \mathbb{R}^{t \times d},$$

where $W_2 \in \mathbb{R}^{d_{\mathrm{ff}} \times d}$, $\mathbf{b}_2 \in \mathbb{R}^{d}$.

The matrices W_1, W_2 and vectors $\mathbf{b}_1, \mathbf{b}_2$ are trainable parameters.

Final Output Distribution

The final output layer transforms the final embeddings into a probability distribution over the token space $\mathcal{Y}$.

Final Output Distribution

The final output layer transforms the final embeddings into a probability distribution over the token space $\mathcal{Y}$.

There are various methods to implement this transformation.

Here, we describe how GPT models achieve this.

Final Output Distribution

The final output layer transforms the final embeddings into a probability distribution over the token space $\mathcal{Y}$.

There are various methods to implement this transformation.

Here, we describe how GPT models achieve this.

Let $X = [\mathbf{x}_1, \cdots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$ be the final embeddings after passing through all the internal layers.

Final Output Distribution

The final output layer transforms the final embeddings into a probability distribution over the token space $\mathcal{Y}$.

There are various methods to implement this transformation.

Here, we describe how GPT models achieve this.

Let $X = [\mathbf{x}_1, \cdots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$ be the final embeddings after passing through all the internal layers.

GPT models select the embedding at the last position, $\mathbf{x}_t$, and compute:

$$\mathbf{z}_t = \mathbf{x}_t^\top M \in \mathbb{R}^{|\mathcal{Y}|},$$

where $M \in \mathbb{R}^{d \times |\mathcal{Y}|}$ is the initial embedding matrix (weight tying is applied).

Final Output Distribution

The final output layer transforms the final embeddings into a probability distribution over the token space $\mathcal{Y}$.

There are various methods to implement this transformation.

Here, we describe how GPT models achieve this.

Let $X = [\mathbf{x}_1, \cdots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$ be the final embeddings after passing through all the internal layers.

GPT models select the embedding at the last position, $\mathbf{x}_t$, and compute:

$$\mathbf{z}_t = \mathbf{x}_t^\top M \in \mathbb{R}^{|\mathcal{Y}|},$$

where $M \in \mathbb{R}^{d \times |\mathcal{Y}|}$ is the initial embedding matrix (weight tying is applied).

The probabilities over $\mathcal{Y}$ are obtained using the softmax function:

$$P(y \mid \text{context}) = \frac{\exp(z_t[y])}{\sum_{y' \in \mathcal{Y}} \exp(z_t[y'])}.$$

Positional Embedding

Transformer blocks, such as multi-head attention and feed-forward layers, are permutation invariant.

Positional Embedding

Transformer blocks, such as multi-head attention and feed-forward layers, are permutation invariant.

This means they cannot capture positional information, which is crucial for language modeling.

Positional Embedding

Transformer blocks, such as multi-head attention and feed-forward layers, are permutation invariant.

This means they cannot capture positional information, which is crucial for language modeling.

To address this, positional embeddings are introduced.

Positional Embedding

Transformer blocks, such as multi-head attention and feed-forward layers, are permutation invariant.

This means they cannot capture positional information, which is crucial for language modeling.

To address this, positional embeddings are introduced.

Let $X = [\mathbf{x}_1, \ldots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$ be the input token embeddings.

Positional Embedding

Transformer blocks, such as multi-head attention and feed-forward layers, are permutation invariant.

This means they cannot capture positional information, which is crucial for language modeling.

To address this, positional embeddings are introduced.

Let $X = [\mathbf{x}_1, \ldots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$ be the input token embeddings.

Positional embeddings define, for each position i, a vector $\mathbf{p}_i \in \mathbb{R}^d$.

Positional Embedding

Transformer blocks, such as multi-head attention and feed-forward layers, are permutation invariant.

This means they cannot capture positional information, which is crucial for language modeling.

To address this, positional embeddings are introduced.

Let $X = [\mathbf{x}_1, \ldots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$ be the input token embeddings.

Positional embeddings define, for each position i, a vector $\mathbf{p}_i \in \mathbb{R}^d$.

Let $P = [\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_t]^\top \in \mathbb{R}^{t \times d}$. The resulting combined embeddings are:

$$X' = X + P \in \mathbb{R}^{t \times d}.$$

Positional Embedding

Transformer blocks, such as multi-head attention and feed-forward layers, are permutation invariant.

This means they cannot capture positional information, which is crucial for language modeling.

To address this, positional embeddings are introduced.

Let $X = [\mathbf{x}_1, \dots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$ be the input token embeddings.

Positional embeddings define, for each position i, a vector $\mathbf{p}_i \in \mathbb{R}^d$.

Let $P = [\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_t]^\top \in \mathbb{R}^{t \times d}$. The resulting combined embeddings are:

$$X' = X + P \in \mathbb{R}^{t \times d}.$$

The positional embeddings P can either be learned or predefined, such as with sinusoidal embeddings, where for $2k, 2k+1 \in [d]$:

$$\mathbf{p}_i[2k] = \sin\left(\frac{i}{10000^{2k/d}}\right), \quad \mathbf{p}_i[2k+1] = \cos\left(\frac{i}{10000^{2k/d}}\right).$$

Layer Normalization

After each multi-head attention or feed-forward layer, an operation named layer normalization is applied position-wise to stabilize processed embeddings.

Layer Normalization

After each multi-head attention or feed-forward layer, an operation named layer normalization is applied position-wise to stabilize processed embeddings.

Let $X = [\mathbf{x}_1, \ldots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$ be the processed embeddings.

For any $j \leq t$, the layer normalization computes:

$$\mu_j = \frac{1}{d} \sum_{i=1}^{d} \mathbf{x}_j[i], \quad \sigma_j^2 = \frac{1}{d} \sum_{i=1}^{d} (\mathbf{x}_j[i] - \mu_j)^2$$

Layer Normalization

After each multi-head attention or feed-forward layer, an operation named layer normalization is applied position-wise to stabilize processed embeddings.

Let $X = [\mathbf{x}_1, \ldots, \mathbf{x}_t]^\top \in \mathbb{R}^{t \times d}$ be the processed embeddings.

For any $j \leq t$, the layer normalization computes:

$$\mu_j = \frac{1}{d} \sum_{i=1}^{d} \mathbf{x}_j[i], \quad \sigma_j^2 = \frac{1}{d} \sum_{i=1}^{d} (\mathbf{x}_j[i] - \mu_j)^2$$

Then, the normalized output is:

$$\mathbf{x}_j' = \gamma \odot \left(\frac{\mathbf{x}_j - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} \right) + \beta$$

where $\gamma, \beta \in \mathbb{R}^d$ are learnable parameters, and ϵ is a small constant.

Encoder vs. Decoder

There are two types of block stacks in the Transformer architecture: the encoder and the decoder.

Encoder vs. Decoder

There are two types of block stacks in the Transformer architecture: the encoder and the decoder.

The encoder stack consists of multi-head self-attention layers and feed-forward layers.

Encoder vs. Decoder

There are two types of block stacks in the Transformer architecture: the encoder and the decoder.

The encoder stack consists of multi-head self-attention layers and feed-forward layers.

The decoder stack consists of multi-head self-attention, masked multi-head self-attention, and feed-forward layers.

Encoder vs. Decoder

There are two types of block stacks in the Transformer architecture: the encoder and the decoder.

The encoder stack consists of multi-head self-attention layers and feed-forward layers.

The decoder stack consists of multi-head self-attention, masked multi-head self-attention, and feed-forward layers.

Let $S_i = \frac{Q_i K_i^\top}{\sqrt{d'}} \in \mathbb{R}^{t \times t}$ be the attention score matrix. The masked attention modifies the score matrix to

$$S_i + \underbrace{\begin{bmatrix} 0 & -\infty & -\infty & \cdots \\ 0 & 0 & -\infty & \cdots \\ 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}}_{\text{mask matrix}} \in \mathbb{R}^{t \times t},$$

Encoder vs. Decoder

There are two types of block stacks in the Transformer architecture: the encoder and the decoder.

The encoder stack consists of multi-head self-attention layers and feed-forward layers.

The decoder stack consists of multi-head self-attention, masked multi-head self-attention, and feed-forward layers.

Let $S_i = \frac{Q_i K_i^\top}{\sqrt{d'}} \in \mathbb{R}^{t \times t}$ be the attention score matrix. The masked attention modifies the score matrix to

$$S_i + \underbrace{\begin{bmatrix} 0 & -\infty & -\infty & \cdots \\ 0 & 0 & -\infty & \cdots \\ 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}}_{\text{mask matrix}} \in \mathbb{R}^{t \times t},$$

The mask matrix ensures that the attention score at each position attends only to previous tokens.

Additional Remarks

- The attention layer is the only mechanism in the Transformer that incorporates context information between different embedding positions.
- All other blocks are applied position-wise.
- Technically, there are also residual connections that add the input of each layer to its output to help prevent vanishing gradient issues.
- There are many variants of the Transformer architecture. For example, in GPT, only the decoder block is used.

Overview

- **The Next-Token Prediction Paradigm**
  - Stochastic modeling of languages
  - Conditional distribution estimation
- **The Transformer Architecture**
  - Vector embedding
  - Attention mechanism: Self-attention and multi-head attention
  - Positional encoding and why it's needed
- **Generative Pre-trained Transformer (GPT) Models**
  - Pretraining: Learning from large, diverse datasets
  - Fine-tuning: Specializing GPT for specific tasks
  - GPT-3 and beyond: Scaling and capabilities

The Generative Pre-trained Transformer (GPT) Models

The Generative Pre-trained Transformer (GPT) models are Transformer architectures that utilize only the decoder blocks.

The Generative Pre-trained Transformer (GPT) Models

The Generative Pre-trained Transformer (GPT) models are Transformer architectures that utilize only the decoder blocks.

- E.g., GPT-3 Small model has 125 million parameters, consisting of 12 layers, an embedding dimension of 768, a feed-forward dimension (d_{ff}) of $3,072$, 12 attention heads, and each head has an output dimension of 64.

The Generative Pre-trained Transformer (GPT) Models

The Generative Pre-trained Transformer (GPT) models are Transformer architectures that utilize only the decoder blocks.

- E.g., GPT-3 Small model has 125 million parameters, consisting of 12 layers, an embedding dimension of 768, a feed-forward dimension (d_{ff}) of $3,072$, 12 attention heads, and each head has an output dimension of 64.

The model is first pre-trained on a large amount of data (such as Wikipedia) using autoregressive training.

The Generative Pre-trained Transformer (GPT) Models

The Generative Pre-trained Transformer (GPT) models are Transformer architectures that utilize only the decoder blocks.

- E.g., GPT-3 Small model has 125 million parameters, consisting of 12 layers, an embedding dimension of 768, a feed-forward dimension (d_{ff}) of $3,072$, 12 attention heads, and each head has an output dimension of 64.

The model is first pre-trained on a large amount of data (such as Wikipedia) using autoregressive training.

The pre-trained model is then fine-tuned for downstream tasks (such as text classification, translation, and chatbots) using one of two methods:

The Generative Pre-trained Transformer (GPT) Models

The Generative Pre-trained Transformer (GPT) models are Transformer architectures that utilize only the decoder blocks.

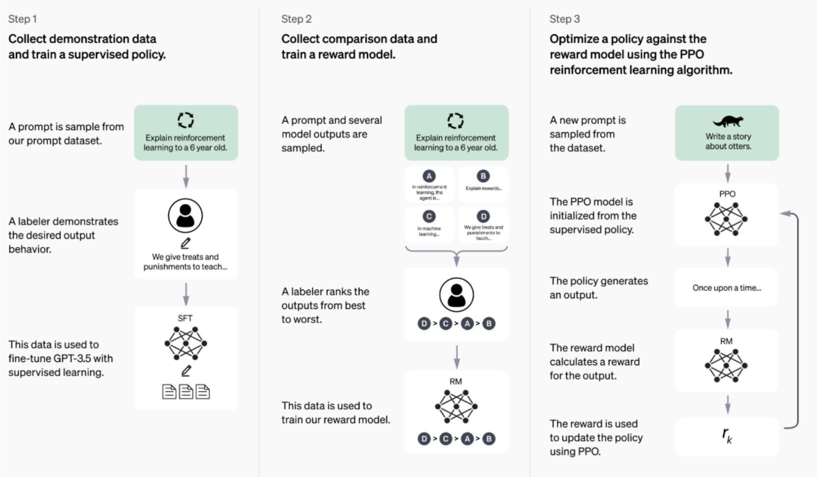
- E.g., GPT-3 Small model has 125 million parameters, consisting of 12 layers, an embedding dimension of 768, a feed-forward dimension (d_{ff}) of $3,072$, 12 attention heads, and each head has an output dimension of 64.

The model is first pre-trained on a large amount of data (such as Wikipedia) using autoregressive training.

The pre-trained model is then fine-tuned for downstream tasks (such as text classification, translation, and chatbots) using one of two methods:

- Supervised fine-tuning, which retrains the model on a small amount of well-organized data specific to the task.
- Reinforcement Learning from Human Feedback (RLHF), which uses human annotations to train a reward model, further employed to fine-tune the original model.

Fine-tuning Pipeline of Training GPT models



Pre-training Process of GPT

Autoregressive Modeling: GPT models the probability of a sequence of tokens $y_1, y_2, \ldots, y_T$ as the product of conditional probabilities:

$$P(y_1, y_2, \ldots, y_T) = \prod_{t=1}^{T} P(y_t \mid y^{t-1}).$$

Objective: Minimize the negative log-likelihood loss over the training data $\mathcal{D} \subset \mathcal{Y}^*$:

$$\mathcal{L}(\theta) = -\frac{1}{|\mathcal{D}|} \sum_{(y_1, y_2, \ldots, y_T) \in \mathcal{D}} \sum_{t=1}^{T} \log P_\theta(y_t \mid y^{t-1})$$

where θ represents the model parameters.

Training: The model minimizes the loss $\mathcal{L}(\theta)$ by updating the parameters θ using backpropagation and gradient descent.

Supervised Fine-tuning (SFT) Process of GPT

Objective: After pre-training, GPT is fine-tuned on a specific downstream task by minimizing a task-specific loss function.

Supervised Fine-tuning (SFT) Process of GPT

Objective: After pre-training, GPT is fine-tuned on a specific downstream task by minimizing a task-specific loss function.

For example, for chatbots (like ChatGPT), the objective is:

$$\mathcal{L}_{\text{fine-tune}}(\theta) = -\frac{1}{|\mathcal{D}_{\text{fine-tune}}|} \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{D}_{\text{fine-tune}}} \log P_\theta(\mathbf{y} \mid \mathbf{x})$$

where $\mathbf{x} \in \mathcal{Y}^*$ is the prompt and $\mathbf{y} \in \mathcal{Y}^*$ is the human-annotated answer.

Supervised Fine-tuning (SFT) Process of GPT

Objective: After pre-training, GPT is fine-tuned on a specific downstream task by minimizing a task-specific loss function.

For example, for chatbots (like ChatGPT), the objective is:

$$\mathcal{L}_{\text{fine-tune}}(\theta) = -\frac{1}{|\mathcal{D}_{\text{fine-tune}}|} \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{D}_{\text{fine-tune}}} \log P_\theta(\mathbf{y} \mid \mathbf{x})$$

where $\mathbf{x} \in \mathcal{Y}^*$ is the prompt and $\mathbf{y} \in \mathcal{Y}^*$ is the human-annotated answer.

Here, $\mathcal{D}_{\text{fine-tune}}$ is a small dataset with human-crafted answers.

Supervised Fine-tuning (SFT) Process of GPT

Objective: After pre-training, GPT is fine-tuned on a specific downstream task by minimizing a task-specific loss function.

For example, for chatbots (like ChatGPT), the objective is:

$$\mathcal{L}_{\text{fine-tune}}(\theta) = -\frac{1}{|\mathcal{D}_{\text{fine-tune}}|} \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{D}_{\text{fine-tune}}} \log P_\theta(\mathbf{y} \mid \mathbf{x})$$

where $\mathbf{x} \in \mathcal{Y}^*$ is the prompt and $\mathbf{y} \in \mathcal{Y}^*$ is the human-annotated answer.

Here, $\mathcal{D}_{\text{fine-tune}}$ is a small dataset with human-crafted answers.

Training: The model minimizes the loss $\mathcal{L}_{\text{fine-tune}}(\theta)$ by updating the parameters θ using backpropagation and gradient descent, with θ initialized as the pre-trained model.

Reinforcement Learning with Human Feedback (RLHF)

Objective: Fine-tune a pretrained language model using human feedback to align model outputs with desired behaviors.

Reinforcement Learning with Human Feedback (RLHF)

Objective: Fine-tune a pretrained language model using human feedback to align model outputs with desired behaviors.

The process starts by training a reward model, which is initialized from the pretrained model after supervised fine-tuning, with its final layer replaced by a linear map that produces a scalar output.

Reinforcement Learning with Human Feedback (RLHF)

Objective: Fine-tune a pretrained language model using human feedback to align model outputs with desired behaviors.

The process starts by training a reward model, which is initialized from the pretrained model after supervised fine-tuning, with its final layer replaced by a linear map that produces a scalar output.

The input to the reward model $r_\theta(\mathbf{x}, \mathbf{y})$ is a prompt-response pair $(\mathbf{x}, \mathbf{y})$, and the output is a scalar that evaluates the quality of the response.

Reinforcement Learning with Human Feedback (RLHF)

Objective: Fine-tune a pretrained language model using human feedback to align model outputs with desired behaviors.

The process starts by training a reward model, which is initialized from the pretrained model after supervised fine-tuning, with its final layer replaced by a linear map that produces a scalar output.

The input to the reward model $r_\theta(\mathbf{x}, \mathbf{y})$ is a prompt-response pair $(\mathbf{x}, \mathbf{y})$, and the output is a scalar that evaluates the quality of the response.

Training the Reward Model: For each prompt $\mathbf{x}$, K responses $\mathbf{y}_1, \ldots, \mathbf{y}_K$ are sampled from the pretrained model after SFT.

Reinforcement Learning with Human Feedback (RLHF)

Objective: Fine-tune a pretrained language model using human feedback to align model outputs with desired behaviors.

The process starts by training a reward model, which is initialized from the pretrained model after supervised fine-tuning, with its final layer replaced by a linear map that produces a scalar output.

The input to the reward model $r_\theta(\mathbf{x}, \mathbf{y})$ is a prompt-response pair $(\mathbf{x}, \mathbf{y})$, and the output is a scalar that evaluates the quality of the response.

Training the Reward Model: For each prompt $\mathbf{x}$, K responses $\mathbf{y}_1, \ldots, \mathbf{y}_K$ are sampled from the pretrained model after SFT. Human labelers rank these responses, and the reward model is updated using the following loss:

$$\text{loss}(\theta) = -\mathbb{E}_{(\mathbf{x}, \mathbf{y}_w, \mathbf{y}_l) \sim \mathcal{D}} \left[\log \sigma \left(r_\theta(\mathbf{x}, \mathbf{y}_w) - r_\theta(\mathbf{x}, \mathbf{y}_l) \right) \right],$$

Reinforcement Learning with Human Feedback (RLHF)

Objective: Fine-tune a pretrained language model using human feedback to align model outputs with desired behaviors.

The process starts by training a reward model, which is initialized from the pretrained model after supervised fine-tuning, with its final layer replaced by a linear map that produces a scalar output.

The input to the reward model $r_\theta(\mathbf{x}, \mathbf{y})$ is a prompt-response pair $(\mathbf{x}, \mathbf{y})$, and the output is a scalar that evaluates the quality of the response.

Training the Reward Model: For each prompt $\mathbf{x}$, K responses $\mathbf{y}_1, \ldots, \mathbf{y}_K$ are sampled from the pretrained model after SFT. Human labelers rank these responses, and the reward model is updated using the following loss:

$$\text{loss}(\theta) = -\mathbb{E}_{(\mathbf{x}, \mathbf{y}_w, \mathbf{y}_l) \sim \mathcal{D}} \left[\log \sigma \left(r_\theta(\mathbf{x}, \mathbf{y}_w) - r_\theta(\mathbf{x}, \mathbf{y}_l) \right) \right],$$

where $\mathbf{y}_w$ is ranked higher than $\mathbf{y}_l$, and the pairs are sampled from a dataset $\mathcal{D}$ of human comparisons.

Reinforcement Learning with Human Feedback (RLHF)

Objective: Fine-tune a pretrained language model using human feedback to align model outputs with desired behaviors.

The process starts by training a reward model, which is initialized from the pretrained model after supervised fine-tuning, with its final layer replaced by a linear map that produces a scalar output.

The input to the reward model $r_\theta(\mathbf{x}, \mathbf{y})$ is a prompt-response pair $(\mathbf{x}, \mathbf{y})$, and the output is a scalar that evaluates the quality of the response.

Training the Reward Model: For each prompt $\mathbf{x}$, K responses $\mathbf{y}_1, \ldots, \mathbf{y}_K$ are sampled from the pretrained model after SFT. Human labelers rank these responses, and the reward model is updated using the following loss:

$$\text{loss}(\theta) = -\mathbb{E}_{(\mathbf{x}, \mathbf{y}_w, \mathbf{y}_l) \sim \mathcal{D}} \left[\log \sigma \left(r_\theta(\mathbf{x}, \mathbf{y}_w) - r_\theta(\mathbf{x}, \mathbf{y}_l) \right) \right],$$

where $\mathbf{y}_w$ is ranked higher than $\mathbf{y}_l$, and the pairs are sampled from a dataset $\mathcal{D}$ of human comparisons.

Finally, the reward model is used to fine-tune a policy model using Proximal Policy Optimization (PPO).

Proximal Policy Optimization (PPO)

Let π_ϕ be the policy model, which is initially set to π^{SFT}, the pre-trained model after supervised fine-tuning, and let r_θ be the reward model.

Proximal Policy Optimization (PPO)

Let π_ϕ be the policy model, which is initially set to π^{SFT}, the pre-trained model after supervised fine-tuning, and let r_θ be the reward model.

The Proximal Policy Optimization (PPO) algorithm maximizes the following objective function:

$$\mathsf{obj}(\phi) = \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}_{\pi_\phi}}\left[r_\theta(\mathbf{x},\mathbf{y}) - \beta\log\left(\frac{\pi_\phi(\mathbf{y}\mid\mathbf{x})}{\pi^{\mathsf{SFT}}(\mathbf{y}\mid\mathbf{x})}\right)\right] - \gamma\mathbb{E}_{\mathbf{x}\sim\mathcal{D}_{\mathsf{pretrain}}}[\log(\pi_\phi(\mathbf{x}))],$$

Proximal Policy Optimization (PPO)

Let π_ϕ be the policy model, which is initially set to π^{SFT}, the pre-trained model after supervised fine-tuning, and let r_θ be the reward model.

The Proximal Policy Optimization (PPO) algorithm maximizes the following objective function:

$$\text{obj}(\phi) = \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}_{\pi_\phi}}\left[r_\theta(\mathbf{x},\mathbf{y}) - \beta\log\left(\frac{\pi_\phi(\mathbf{y}\mid\mathbf{x})}{\pi^{\text{SFT}}(\mathbf{y}\mid\mathbf{x})}\right)\right] - \gamma\mathbb{E}_{\mathbf{x}\sim\mathcal{D}_{\text{pretrain}}}[\log(\pi_\phi(\mathbf{x}))],$$

where:

- $\mathcal{D}_{\pi_\phi}$ is the dataset of prompts and responses generated by the current policy model π_ϕ,
- $r_\theta(\mathbf{x},\mathbf{y})$ is the reward from the reward model,
- β controls the regularization term that penalizes the KL-divergence between π_ϕ and the supervised fine-tuned policy π^{SFT}, preventing the policy from deviating too much from π^{SFT},
- γ controls the entropy regularization term, which encourages exploration by promoting higher entropy in the policy, preventing it from becoming too deterministic.

Proximal Policy Optimization (PPO)

Let π_ϕ be the policy model, which is initially set to π^{SFT}, the pre-trained model after supervised fine-tuning, and let r_θ be the reward model.

The Proximal Policy Optimization (PPO) algorithm maximizes the following objective function:

$$\mathrm{obj}(\phi) = \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}_{\pi_\phi}}\left[r_\theta(\mathbf{x},\mathbf{y}) - \beta\log\left(\frac{\pi_\phi(\mathbf{y}\mid\mathbf{x})}{\pi^{\mathsf{SFT}}(\mathbf{y}\mid\mathbf{x})}\right)\right] - \gamma\mathbb{E}_{\mathbf{x}\sim\mathcal{D}_{\mathsf{pretrain}}}[\log(\pi_\phi(\mathbf{x}))],$$

where:

- $\mathcal{D}_{\pi_\phi}$ is the dataset of prompts and responses generated by the current policy model π_ϕ,
- $r_\theta(\mathbf{x},\mathbf{y})$ is the reward from the reward model,
- β controls the regularization term that penalizes the KL-divergence between π_ϕ and the supervised fine-tuned policy π^{SFT}, preventing the policy from deviating too much from π^{SFT},
- γ controls the entropy regularization term, which encourages exploration by promoting higher entropy in the policy, preventing it from becoming too deterministic.

PPO iteratively updates the policy π_ϕ using a gradient-based approach by sampling batches from $\mathcal{D}_{\pi_\phi}$ and $\mathcal{D}_{\mathsf{pretrain}}$.

The Scaling Law and Beyond

It has been observed that the performance of a large language model (LLM) scales with its model size, which is hypothesized as the scaling law.

The Scaling Law and Beyond

It has been observed that the performance of a large language model (LLM) scales with its model size, which is hypothesized as the scaling law.

However, it remains an active area of research to understand how the structure of LLMs, the data, and training processes impact their capabilities, such as reasoning, generalization, and interpretability, among other tasks.

The Scaling Law and Beyond

It has been observed that the performance of a large language model (LLM) scales with its model size, which is hypothesized as the scaling law.

However, it remains an active area of research to understand how the structure of LLMs, the data, and training processes impact their capabilities, such as reasoning, generalization, and interpretability, among other tasks.

Future work aims to uncover the fundamental principles governing these relationships and to identify optimal strategies for enhancing LLM performance across various domains.

The Scaling Law and Beyond

It has been observed that the performance of a large language model (LLM) scales with its model size, which is hypothesized as the scaling law.

However, it remains an active area of research to understand how the structure of LLMs, the data, and training processes impact their capabilities, such as reasoning, generalization, and interpretability, among other tasks.

Future work aims to uncover the fundamental principles governing these relationships and to identify optimal strategies for enhancing LLM performance across various domains.

There is also active research focused on understanding the undesired behaviors of LLMs, such as hallucination, and exploring the impact of alignment to mitigate these issues.

Concluding Remark

- In this lecture, we introduced the basic foundations of LLMs, from their principal objective (next token prediction), to their underlying structure (the transformer), to the training pipeline, and discussed their scalability and key challenges.
- There has been significant recent research focused on understanding the abilities of LLMs from both theoretical and empirical perspectives.
- We hope this lecture has provided readers with the basic knowledge of this rapidly evolving field.