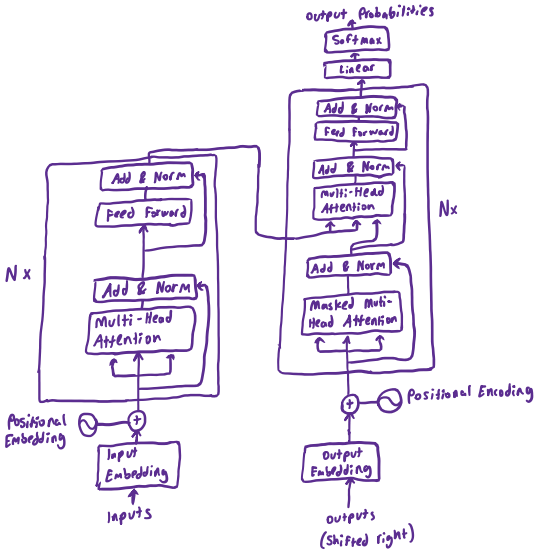
Attention is All You Need

Friday, June 13, 2025 6:25 AM

High Level Overview

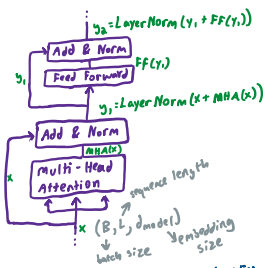
Introduces transformers
- Relies entirely on self-attention - no sequential processing or convolutions
- uses encoder-decoder architecture
- Predicts probability distribution for most likely next token

Model Architecture



Encoder

- Encoder has N=6 identical layers
- Encoder Layer: 2 sub-layers
  1) self-attention
  2) Position-wise fully connected feed-forward network
  - residual connection for both sub-layers, then layer normalization



Layer Norm
- normalizes the inputs across features
- Given input x, where $x = (\text{batch\_size}, \text{num\_features})$
  1) For each row in batch, mean and variance are computed
  2) features are normalized

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

  3) scales & shifts normalized result with learnable parameters

$$y_i = \gamma \hat{x}_i + \beta$$

(γ, β: learned parameters updated during back propagation)

sublayers in model & embedding layers produce outputs of dimension 512

data → EncoderLayer_1 → EncoderLayer_2 → ... → EncoderLayer_N → encoder stack output

Decoder

- Stack of N=6 layers
- Contains same two layers as the encoder and an additional multi-head attention sub-layer on the output of the encoder stack
- self-attention sub-layer in decoder stack is modified for masking
  - Tokens can only see previous tokens in the sequence
    - → all future positions set to $\text{Softmax}(-\infty) = 0$
    - → output embeddings offset by one position - to predict next token only

Attention

Query (Q): what we're trying to find
Key (K): what each word advertises
Value (V): what each word will say if chosen

Q, K, V Analogy

Q: "Looking for subject"
K: "I am subject"
V: "Subject = dog"

Scaled Dot Product Attention

Given:
- d_k = dimensionality of K and Q
- d_v = dimensionality of V
- h = number of heads

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

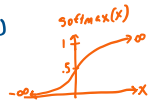
(How relevant is each token to what I'm trying to find? · actual data · scale factor)

$$\Rightarrow \text{Softmax}\left(\underbrace{\frac{(B,h,L,d_k)\times(B,h,d_k,L)}{\text{scalar}}}\right)\times(B,h,L,d_v)$$

$$= \text{Softmax}((B,h,L,L))\times(B,h,L,d_v) = (B,h,L,d_v) := \text{Attention}$$

(batch size · dimensionality of V · # of heads · sequence length)

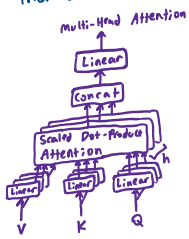
Meaning: How much should I focus on each vector (value) when creating a new output vector?



Insight: the transformer design - especially vectorized attention - is intentionally built to maximize parallelization in hardware

Multi-Head Attention

- Instead of computing Attention all at once, we compute it in parallel and then combine results



$W^O \in \mathbb{R}^{h \cdot d_v \times d_{model}}$

Meaning/Purpose: Weight matrix learned during back propagation
Insight: In implementation, we'd use nn.Linear(), so this includes a bias matrix. For clarity, the paper omits an explicit B^O

(learned weight matrices (like W^Q) to give the model more flexibility)

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Meaning/Purpose: head_i represents how much to pay attention to each value when constructing the output for each head?

Insight: Each head learns to focus on different kinds of relationships. This gives the model richer and more expressive understanding of the input. This makes the model faster & smarter.

$$\text{MultiHead}(Q,K,V) = \text{Concat}(\text{head}_1,\dots,\text{head}_h)W^O$$

Meaning/Purpose: Combines separate understandings into one large vector
Analogy: multiple department heads reporting their understandings to the CEO, who's job is to combine these understandings (attentions) into one cohesive and singular understanding.

Applications of Attention

Multi-Head Attention used in 3 ways

1) Queries come from previous decoder layer and memory keys & values come from output of encoder.
   - Allows every position in decoder to attend to all positions in input sequence
2) Encoder contains self-attention layers
   - Each position in the encoder can attend to all positions in previous layer of the encoder
3) self-attention layers in the decoder allow each position in the decoder to attend to all positions in the decoder up to and including that position

Position-wise Feed Forward Networks

$$FFN(x) = RELU(xW_1+b_1)W_2+b_2$$

$W_1 \in \mathbb{R}^{512\times2048}$
$W_2 \in \mathbb{R}^{2048\times512}$

Positional Encoding

... was in the sequence

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

(pos: position; i: index of the dimension in the model)

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

Why Self Attention?

1) Lower Computational Complexity per Layer
2) Parallelizable compared to LSTM's & RNN's
3) Better for long-range dependencies

Training

Optimizer

- Uses Adam optimizer

Adam Optimizer

- Uses average direction of past gradients and their magnitude to take better per-parameter steps
  - $g_t = \nabla_\theta L(\theta_t)$ = gradient at step t
  - α = learning rate
  - β_1, β_2 = decay rates
  - ϵ = small value to prevent division by 0

Algorithm:

0) Initialize 1st, 2nd moment estimates to 0
   - 1st moment estimate = m_0
   - 2nd moment estimate = v_0
1) Compute Gradient
   $g_t = \nabla_\theta L(\theta_t)$
2) Update first moment estimate (mean of gradients)
   $m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1)\cdot g_t$
   - exponential moving average of gradients
   - has a smoothing effect
3) Update Second Estimate (uncentered Variance of Gradients)
   $v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2)\cdot g_t^2$
   - exponential moving average of squared gradients
   - tracks size of gradients
4) Bias Correction
   $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$ $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$
5) Parameter Update
   $\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$

$\beta_1 = 0.9$ $\beta_2 = 0.98$ $\epsilon = 10^{-9}$

Varied learning rate

$lrate = d_{model}^{-.5} \cdot \min(step\_num^{-.5}, step\_num \cdot warmup\_steps^{-1.5})$

(100 000 for base model; $warmup\_steps$ = 4000)

Regularization

1) Residual Dropout - Sublayer
   - Applied to output of each sublayer before added to sublayer input $P_{drop} = 0.1$
2) Residual Dropout to sums of embeddings & positional encodings in encoder & decoder stacks $P_{drop} = 0.1$
3) Label smoothing
   $\epsilon_{ls} = 0.1$