Assignment #4: Transformer Implementation

Paul Hongsuck Seo

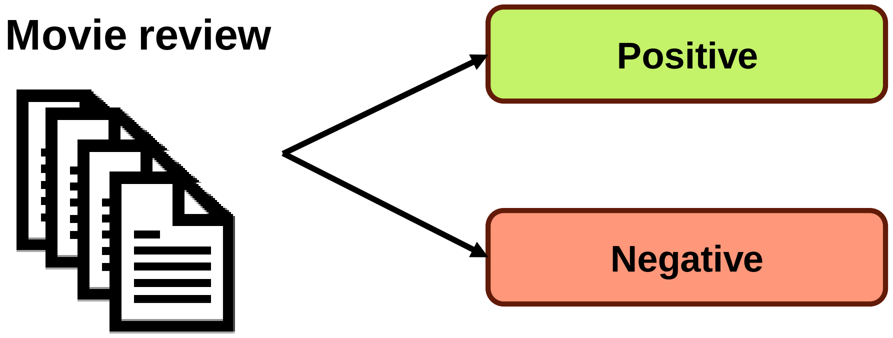
Korea University

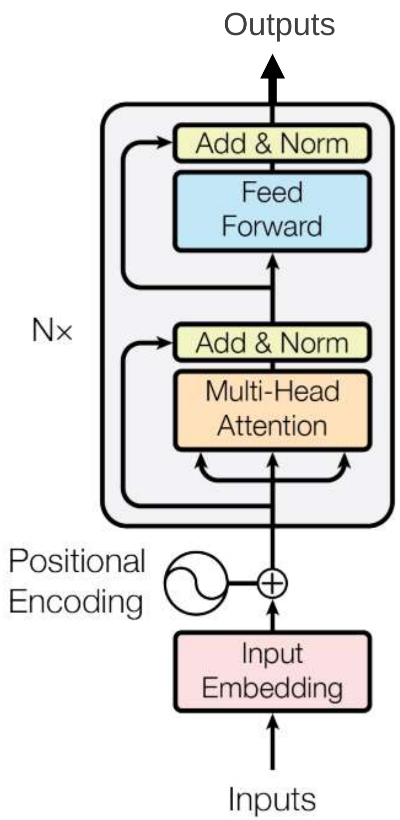
KOREA UNIVERSITY

MIL Multimodal Interactive Intelligence Laboratory

Transformer Implementation

Implement Transformer Classifier





- Perform the text classification using "IMDb Movie Reviews" dataset with transformer
- The Ipython Notebook "Transformer_Implementation.ipynb" will walk you through the implementation of transformer classifier.

Transformer Implementation

Instructions

- Follow the instructions in the **Transformer_Implementation.ipynb** notebook to complete the assignment.
    - Load the **IMDb** data **(No need for any modifications)**
    - Preprocessing the data **(No need for any modifications)**
    - Complete the transformer code and train the transformer model
    - Use the pre-trained BERT model weights and fine-tune them for the IMDb dataset.
    - Complete **transformer_skeleton.py**

      → same as the cells in Transformer_Implementation.ipynb

IMDb movie review Dataset

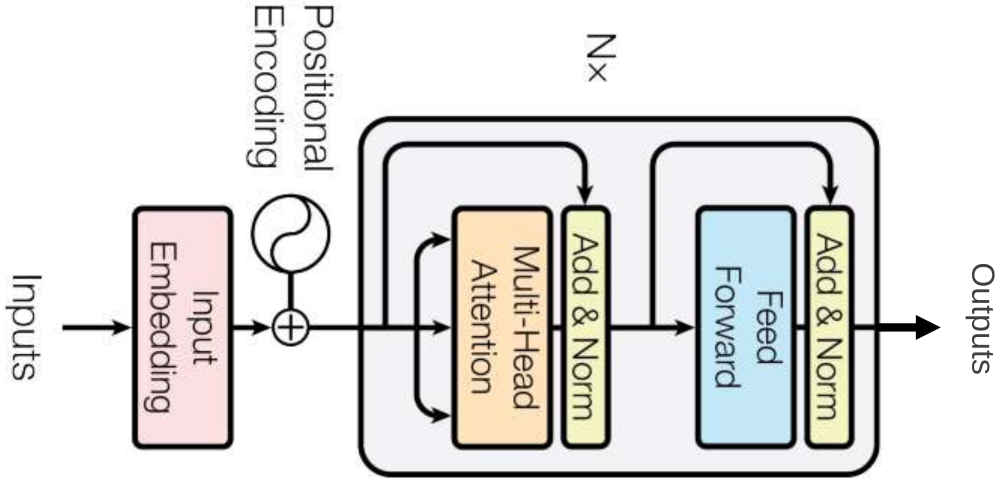
- IMDb Movie Reviews dataset
  - Contains 50,000 movie reviews taken from IMDb (Internet Movie Database)
  - https://ai.stanford.edu/~amaas/data/sentiment/
- Dataset Composition
  - 25,000 reviews are used for training and 25,000 reviews are used for testing.
  - Reviews are labeled as either positive or negative

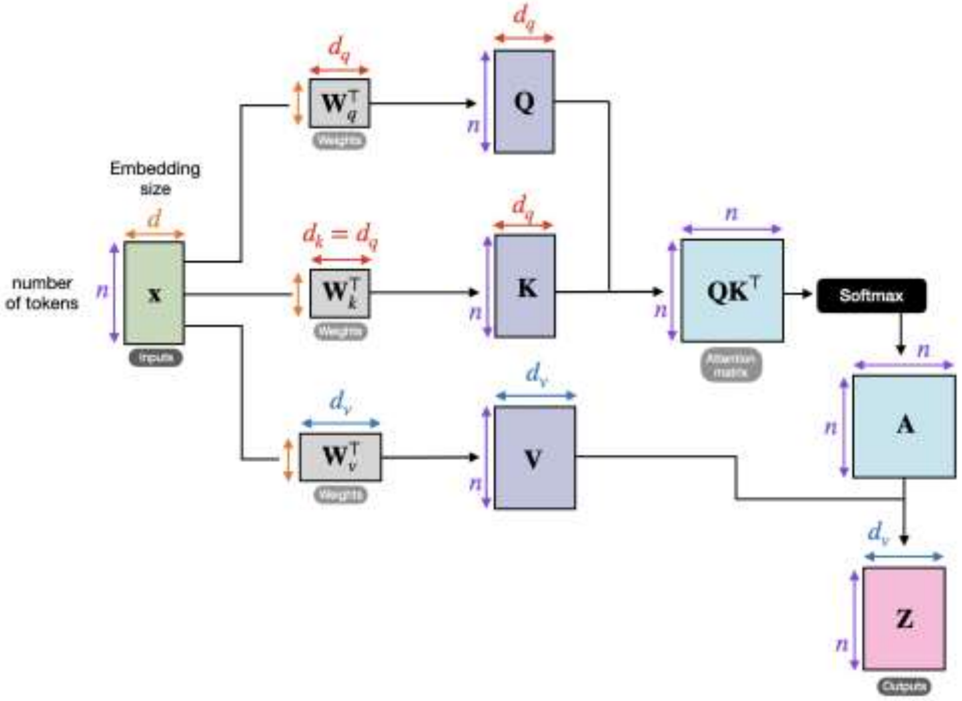
```
After seeing this film months ago, it keeps jumping back
into my consciousness and I feel I must buy it or at
least see it again, even though I watched it at least 3
times when I rented it at that point…
```

→ **Label: 1 (Positive)**

Transformer

- Transformers are neural networks designed for **sequential data processing**.
- They process entire sequences simultaneously, using **self-attention** mechanisms to handle dependencies between tokens.
- **Multi-head attention** enables the model to focus on different parts of the sequence simultaneously.
- We will explore Transformer architecture.





Transformer implementation

Instructions

- You need to complete the **Transformer Model** code accurately based on the explanation of the transformer provided in the following slide.
- It includes **Positional Encoding**, **Multi-Head Attention**, and the internal operations of the transformer, among others.

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, nhead, dropout=0.1):
        super(MultiHeadAttention, self).__init__()
        assert d_model % nhead == 0, "d_model must be divisible by nhead"

        self.d_model = d_model
        self.nhead = nhead
        self.d_k = d_model // nhead
        self.dropout = nn.Dropout(dropout)

        # Hint: Define the linear layers to project the input for query, key, and value
        self.w_q = nn.Linear(###blank###, ###blank###)
        self.w_k = nn.Linear(###blank###, ###blank###)
        self.w_v = nn.Linear(###blank###, ###blank###)

        self.w_o = nn.Linear(d_model, d_model)
```

```python
    def forward(self, src, src_mask=None, src_key_padding_mask=None):
        # Hint: Apply self-attention mechanism on the src
        src2 = self.self_attn(query=###blank###, key=###blank###, value=###blank###, attn_mask=src_mask,
                              key_padding_mask=src_key_padding_mask)[0]

        # Hint: Add residual connection followed by normalization
        src = ###fill in the blank###
        src = ###fill in the blank###

        #Apply the feed-forward network
        src2 = self.linear2(self.dropout(self.linear1(src)))

        # Hint: Add residual connection followed by normalization
        src = ###fill in the blank###
        src = ###fill in the blank###)
        return src
```

Transformer implementation

Positional encoding

- Transformers do not have an inherent sense of word order, unlike RNNs.
- Positional encoding provide position information to the model using sine and cosine functions.
- Formula:

  - Even indices: $PE(pos, 2i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$
  - Odd indices: $PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$
  - $x = x + PE(x)$

Transformer implementation

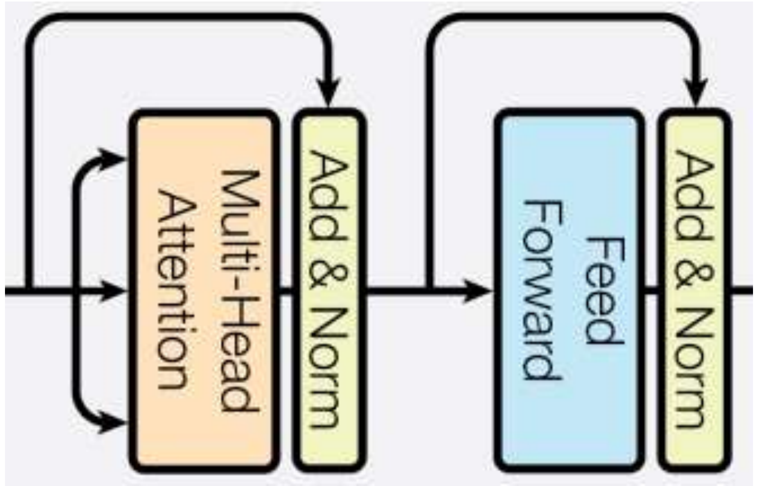
Multi-Head Attention

- Allows the model to attend to different positions within the sequence **in parallel**.
- Each "head" in the attention mechanism processes the input data in a different way, learning multiple relationships simultaneously.
- Instead of applying a single attention mechanism, we apply multiple ones (hence, "multi-head"), each learning a different representation of the input.
- Formula:
  - $scores = \frac{Q \cdot K^T}{\sqrt{d_k}}$
  - $Attention\ Weights = softmax(scores)$
  - $Attention\ Output = dropout(Attention\ Weights) \cdot V$
  - $Multi\ head\ Attention\ output = W_O \times Attention\ Output$

Transformer implementation

Transformer Encoder Layer

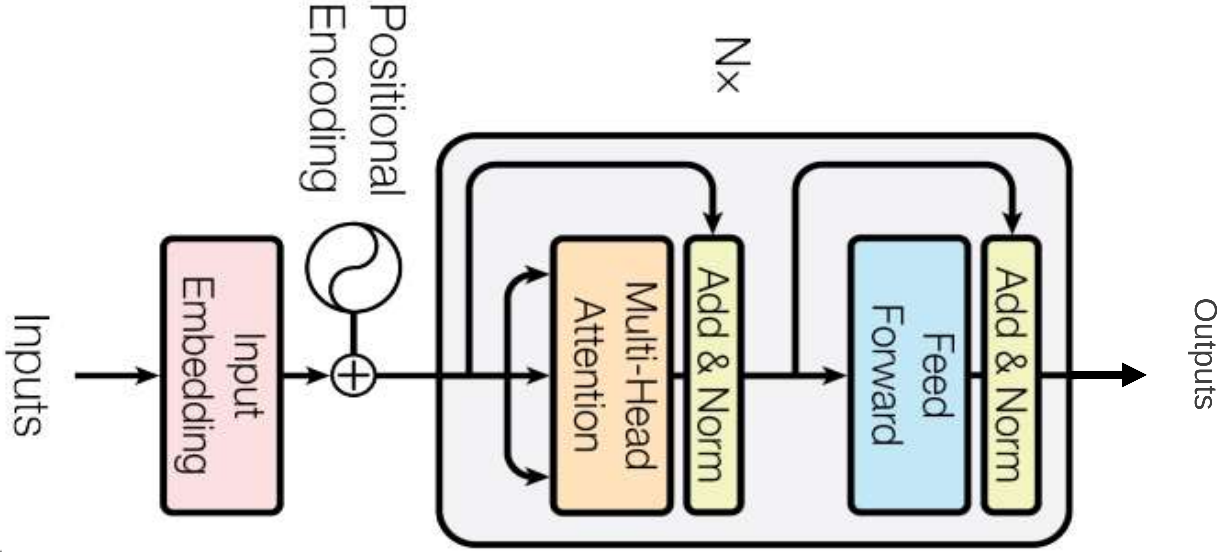
- Consists of two main operations: **Self-Attention** and a **Feed-Forward Neural Network**
- After **Self-Attention**, a **residual connection**, **Dropout**, and **Layer Normalization** are applied to the attention output.
- The input is passed through the **FFN**, followed by a second **residual connection**, **Dropout**, and **Layer Normalization** applied to the FFN output.
- Formula:
  - $Output1 \ = normaliztion(input + dropout(attention\ output))$
  - $Output2 = normaliztion(output1 + dropout(feedforward\ output))$



Transformer implementation

Transformer

- The **Transformer** first applies **Positional Encoding (PE)** to the input to provide information about the order of tokens in the sequence.
- After positional encoding, the input passes through **N Transformer layers**, where each layer performs self-attention and feed-forward operations with residual connections and normalization.
- Finally, the output passes through a **fully connected (FC)** layer, producing the final output.



Transformer implementation

Train the transformer model

- Train the Transformer model using the provided code.

Load pretrained BERT

- Load the pretrained BERT model using the provided code and train it.
- Analyze the results and write a report based on your findings.

Using pretrained weight for BERT

Let's load the pretrained weights from BERT and see how much the accuracy improves.

```
[ ] from transformers import BertModel

    class TransformerModelWithBERT(nn.Module):
        def __init__(self, vocab_size, d_model, nhead, num_encoder_layers, num_classes, dim_feedforward=2048, dropout=0.1, max_len=512
            super(TransformerModelWithBERT, self).__init__()

            self.bert = BertModel.from_pretrained('bert-base-uncased')

            self.embedding = nn.Embedding(vocab_size, d_model)
            self.embedding.weight = nn.Parameter(self.bert.embeddings.word_embeddings.weight)

            self.pos_encoder = PositionalEncoding(d_model, max_len)
            self.pos_encoder.pe = nn.Parameter(self.bert.embeddings.position_embeddings.weight.unsqueeze(0), requires_grad=False)

            encoder_layers = TransformerEncoderLayer(d_model, nhead, dim_feedforward, dropout)
            self.transformer_encoder = TransformerEncoder(encoder_layers, num_encoder_layers)

            self.fc = nn.Linear(d_model, num_classes)
            self.d_model = d_model
```

Transformer Implementation

- You must submit **“transformer_skeleton.py”** along with the **report**.
  (Do not modify the name of the Python file.)
- Include a **1 page** report in **CVPR** format that describes your code, results, and discussions.
- The report should be written in **English**.

CVPR format : https://cvpr.thecvf.com/Conferences/2024/AuthorGuidelines
→ Download CVPR 2024 Author Kit

Transformer Implementation

Please do NOT copy your friends' and internet sources.

Please start your assignment EARLY.

"Late submissions will not be accepted"