NPFL140 Lecture Notes: Transformer Architecture

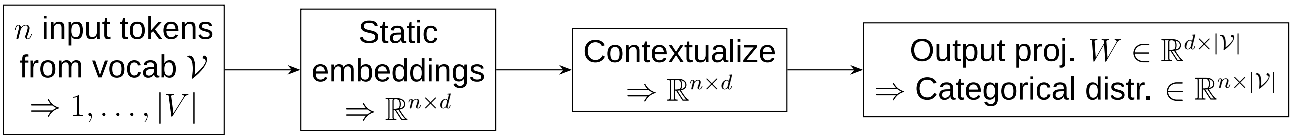
Jindřich Libovický

March 7, 2024

Quiz [5 min]

Architecture [25 min]

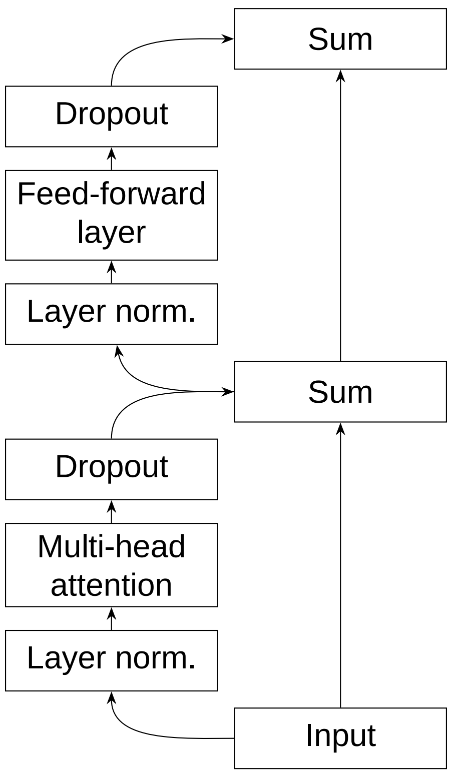
- Deep learning NLP in general: input embeddings, contextualizer, output generation



- Input and output require a fixed-sized vocabulary
- Output is a projection with softmax
- Contextualizer: Transformer blocks of two sub-layers — feedforward-layers, self-attention

Overall structure

Residual pathway / information highway with blocks:



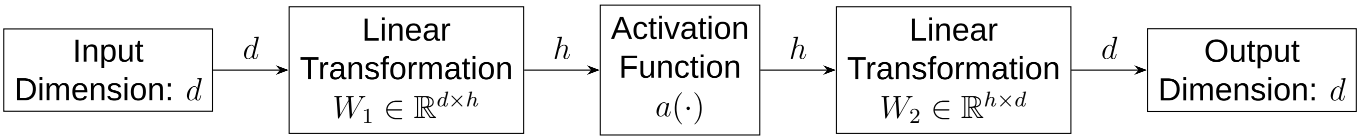
Residual connections for block implementing function $\mathcal{F}$

$$\mathcal{R}(X) = \mathcal{F}(X) + X$$

Feed-forward Sublayer

$$\text{FeedForward}(X) = W_2 \cdot a(W_1 \cdot X + b_1) + b_2 \tag{1}$$

- X is a sequence of states corresponding to words $(x_1, \ldots, x_n)$ of dimension d
- Trainable parameters: $W_1 \in \mathbb{R}^{d \times 4d}$, $W_2 \in \mathbb{R}^{4d \times d}$, $b_1 \in \mathbb{R}^{4d}$, $b_2 \in \mathbb{R}^{d}$



Self-attention Sublayer

- Brainstorm: what information we would need to estimate the next character
- Attention is **communication** mechanism – tokens ask other tokens questions and get some answers
  - Here is what I have, my identity = values
  - Here is what I want = query
  - Here is what I offer - key
- Multi-headed = having **multiple communication channels**

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2}$$

$$\text{head}_i = \text{Attention}(QW_{Qi}, KW_{Ki}, VW_{Vi}) = \text{softmax}\left(\frac{QW_{Qi}(KW_{Ki})^T}{\sqrt{d_k}}\right)VW_{Vi} \tag{3}$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_O \tag{4}$$

Why divide by $\sqrt{d_k}$?
q and k are d_k-dimensional vectors with components independent random variables with mean 0 and variance 1 $\Rightarrow$ their dot product, $q^T k$, has mean and variance of d_k. Since we would prefer these values to have variance 1, we divide by d_k.

- Triangular mask in the decoder to prevent attending to the right
- Encoder-only models: no mask, but we do input masking at training time
- Encoder-decoder models: add a cross-attention module to the decoder

Position Embeddings

- Self-attention treats inputs X as a **unrodered set** of vectors

Connecting the blocks

- **Dropout**: at the end of each block, typical value 0.1
- **Layer normalization**

$$\text{LayerNormalization}(x) = \gamma \odot \left(\frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) + \beta$$

Where:

  - x is the input vector to the layer.
  - μ is the mean of the input vector.
  - σ is the standard deviation of the input vector.
  - ϵ is a small constant to prevent division by zero.
  - γ and β are learnable parameters (scale and shift, respectively).

The original paper did post-normalization; more recent ones do pre-normalization, so it does not block the residual path.

Interactive Coding Session [45 minutes]

Tokenization

```python
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("gp2")
tokenizer.tokenize("I am the walrus.")
tokenizer("I am the walrus.")
tokenizer.convert_ids_to_tokens(40)
```

Feedforward Sublayer

```python
class MLP(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.c_fc = nn.Linear(config.n_embd, 4 * config.n_embd, bias=config.bias)
        self.gelu = nn.GELU()
        self.c_proj = nn.Linear(4 * config.n_embd, config.n_embd, bias=config.bias)
        self.dropout = nn.Dropout(config.dropout)

    def forward(self, x):
        x = self.c_fc(x)
        x = self.gelu(x)
        x = self.c_proj(x)
        x = self.dropout(x)
        return x
```

Self-attention Sublayer

Don't forget to explain `torch.tril` in iPython.

```python
class CausalSelfAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        assert config.n_embd % config.n_head == 0
        # key, query, value projections for all heads, but in a batch
        self.c_attn = nn.Linear(config.n_embd, 3 * config.n_embd, bias=config.bias)
        # output projection
        self.c_proj = nn.Linear(config.n_embd, config.n_embd, bias=config.bias)
        # regularization
        self.attn_dropout = nn.Dropout(config.dropout)
        self.resid_dropout = nn.Dropout(config.dropout)
        self.n_head = config.n_head
        self.n_embd = config.n_embd
        self.dropout = config.dropout
        self.register_buffer(
            "bias", torch.tril(
            torch.ones(config.block_size, config.block_size)).view(
                1, 1, config.block_size, config.block_size))

    def forward(self, x):
        # batch size, sequence length, embedding dimensionality (n_embd)
        B, T, C = x.size()

        # Calculate query, key, values for all heads in batch and
        # move head forward to be the batch dim
        q, k, v  = self.c_attn(x).split(self.n_embd, dim=2)
        # All k, q and v need shape (B, nh, T, hs)
        k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)
        q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)
        v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)

        # Causal self-attention; (B, nh, T, hs) x (B, nh, hs, T) -> (B, nh, T, T)
        att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
        att = att.masked_fill(self.bias[:,:,:T,:T] == 0, float('-inf'))
        att = F.softmax(att, dim=-1)
        att = self.attn_dropout(att)
        y = att @ v # (B, nh, T, T) x (B, nh, T, hs) -> (B, nh, T, hs)
        # Now, re-assemble all head outputs side by side
        y = y.transpose(1, 2).contiguous().view(B, T, C)

        # output projection
        y = self.resid_dropout(self.c_proj(y))
        return y
```

Entire Transformer Block

```python
class Block(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.ln_1 = LayerNorm(config.n_embd, bias=config.bias)
        self.attn = CausalSelfAttention(config)
        self.ln_2 = LayerNorm(config.n_embd, bias=config.bias)
        self.mlp = MLP(config)

    def forward(self, x):
        x = x + self.attn(self.ln_1(x))
        x = x + self.mlp(self.ln_2(x))
        return x
```

Differences in Architectures of Notable Models [15 minutes]