Quiz [5 min]

Architecture [25 min]

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

\[
\begin{align*}
\text{Input tokens from vocab } V & \Rightarrow 1, \ldots, |V| \\
\text{Static embeddings} & \Rightarrow \mathbb{R}^{n \times d} \\
\text{Contextualize} & \Rightarrow \mathbb{R}^{n \times d} \\
\text{Output proj. } W & \in \mathbb{R}^{d \times |V|} \\
& \Rightarrow \text{Categorical distr. } \in \mathbb{R}^{n \times |V|}
\end{align*}
\]

- 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 \( \mathcal{F} \)

\[
R(X) = F(X) + X
\]
Feed-forward Sublayer

\[
\text{FeedForward}(X) = W_2 \cdot a(W_1 \cdot X + b_1) + b_2
\]  

1. \(X\) is a sequence of states corresponding to words \((x_1, \ldots, x_n)\) of dimension \(d\)
2. 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

1. Brainstorm: what information we would need to estimate the next character
2. Attention is \texttt{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
3. Multi-headed = having \texttt{multiple communication channels}

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

\[
\text{head}_i = \text{Attention}(QW_i, KW_{Ki}, VW_{Vi}) = \text{softmax} \left( \frac{QW_i(KW_{Ki})^T}{\sqrt{d_k}} \right) VW_{Vi}
\]

\[
\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_O
\]

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 ⇒ 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 \(d_k\).

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

Position Embeddings

1. Self-attention treats inputs \(X\) as a \texttt{unordered 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.
  - \(\mu\) is the mean of the input vector.
  - \(\sigma\) is the standard deviation of the input vector.
  - \(\epsilon\) is a small constant to prevent division by zero.
  - \(\gamma\) and \(\beta\) 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.
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)
        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
Differences in Architectures of Notable Models [15 minutes]