GPT Transformer Model

November 21, 2024

1 Introduction

This document provides a comprehensive explanation of how to build a GPT transformer model from scratch using PyTorch tensors, without utilizing the high-level modules like `torch.nn` or `torch.nn.functional`. We will:

1. Use the Shakespeare dataset (`input.txt`) and tokenize it using a simple symbol tokenizer.
2. Implement an embedding layer with a learned embedding matrix and fixed positional encoding, applying layer normalization afterward.
3. Construct a multi-head attention block with learnable weight matrices W_Q, W_K, W_V.
4. Train the model using a specified context window, predicting the next token in a sequence.
5. Demonstrate text generation using the trained model.
6. Ensure that operations are primarily performed using matrix multiplication.

2 Data Preparation

2.1 Reading the Dataset

We read the Shakespeare dataset from a local file `input.txt`.

```
# Read the text from 'input.txt'
with open('input.txt', 'r', encoding='utf-8') as f:
    text = f.read()
```

2.2 Tokenization

We create a simple symbol tokenizer by mapping each unique character (symbol) to a unique integer.

$$\begin{aligned} \text{vocab} &= \{\text{unique characters in text}\} \\ \text{vocab\_size} &= |\text{vocab}| \\ \text{char\_to\_idx} &: \text{character} \to \text{integer index} \\ \text{idx\_to\_char} &: \text{integer index} \to \text{character} \end{aligned}$$

```
# Tokenization: map unique symbols to integers
vocab = sorted(set(text))
vocab_size = len(vocab)
char_to_idx = {ch: i for i, ch in enumerate(vocab)}
idx_to_char = {i: ch for i, ch in enumerate(vocab)}
```

Convert the entire text into a sequence of integer indices.

```
# Convert text to indices
text_indices = [char_to_idx[ch] for ch in text]
text_indices = torch.tensor(text_indices, dtype=torch.long)
```

3 Model Components

3.1 Hyperparameters

We define the following hyperparameters:

- `context_window`: Maximum context window size.
- `embedding_dim`: Embedding dimension (must be divisible by `num_heads`).
- `num_heads`: Number of attention heads.

- head_dim: Dimension of each attention head.
- learning_rate: Learning rate for parameter updates.
- epochs: Number of epochs for training.

```
# Hyperparameters
context_window = 10  # Maximum context window size
embedding_dim = 60   # Embedding dimension (must be divisible by num_heads)
num_heads = 5
head_dim = embedding_dim // num_heads
learning_rate = 0.001
epochs = 1  # Number of epochs for training
```

3.2 Embedding Layer

An embedding matrix is initialized with random values. It has dimensions (vocab_size, embedding_dim).

```
# Initialize model parameters
embedding_matrix = torch.randn(vocab_size, embedding_dim, requires_grad=True)
```

3.3 Fixed Positional Encoding

We use a fixed sinusoidal positional encoding to inject positional information into the model.

$$\begin{aligned} \mathrm{PE}_{(pos,2i)} &= \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \\ \mathrm{PE}_{(pos,2i+1)} &= \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \end{aligned} \tag{1}$$

```
# Fixed positional encoding
def get_positional_encoding(seq_len, d_model):
    pe = torch.zeros(seq_len, d_model)
    position = torch.arange(0, seq_len, dtype=torch.float).unsqueeze(1)
    div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log
        (10000.0) / d_model))
    pe[:, 0::2] = torch.sin(position * div_term)
    pe[:, 1::2] = torch.cos(position * div_term)
    return pe

max_seq_len = context_window
positional_encoding = get_positional_encoding(max_seq_len, embedding_dim)
```

3.4 Layer Normalization

Layer normalization is applied to stabilize and accelerate training.

$$\begin{aligned} \mu &= \frac{1}{d}\sum_{i=1}^{d} x_i \\ \sigma^2 &= \frac{1}{d}\sum_{i=1}^{d}(x_i-\mu)^2 \\ \mathrm{LayerNorm}(x) &= \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}} \end{aligned} \tag{2}$$

```
# Layer normalization
def layer_norm(x, epsilon=1e-5):
    mean = x.mean(-1, keepdim=True)
    var = x.var(-1, unbiased=False, keepdim=True)
    x_normalized = (x - mean) / torch.sqrt(var + epsilon)
    return x_normalized
```

3.5 Multi-Head Attention

We implement multi-head attention with learnable weight matrices W_Q, W_K, W_V.

```
W_Q = torch.randn(embedding_dim, embedding_dim, requires_grad=True)
W_K = torch.randn(embedding_dim, embedding_dim, requires_grad=True)
W_V = torch.randn(embedding_dim, embedding_dim, requires_grad=True)
```

3.5.1 Scaled Dot-Product Attention

For each head, we compute the attention scores and apply a causal mask to prevent attending to future tokens.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + \text{mask}\right)V \tag{3}$$

```
# Multi-head attention
def multi_head_attention(Q, K, V, num_heads):
    seq_len, embedding_dim = Q.size()
    head_dim = embedding_dim // num_heads
    assert embedding_dim % num_heads == 0

    def split_heads(x):
        x = x.view(seq_len, num_heads, head_dim)
        x = x.transpose(0, 1)  # (num_heads, seq_len, head_dim)
        return x

    Q = split_heads(Q)
    K = split_heads(K)
    V = split_heads(V)

    scores = Q @ K.transpose(-2, -1) / math.sqrt(head_dim)  # (num_heads,
        seq_len, seq_len)

    # Causal mask to prevent attending to future tokens
    mask = torch.tril(torch.ones(seq_len, seq_len))
    mask = mask.unsqueeze(0)  # (1, seq_len, seq_len)
    scores = scores.masked_fill(mask == 0, -1e9)

    attn_weights = softmax(scores, dim=-1)
    attn_output = attn_weights @ V  # (num_heads, seq_len, head_dim)

    # Concatenate heads
    attn_output = attn_output.transpose(0, 1).contiguous().view(seq_len,
        embedding_dim)
    return attn_output
```

3.6 Output Projection

We project the output of the attention block to the vocabulary size to obtain logits for the next token prediction.

```
output_weights = torch.randn(embedding_dim, vocab_size, requires_grad=True)
output_bias = torch.zeros(vocab_size, requires_grad=True)
```

4 Training Process

4.1 Preparing Text Chunks

We split the text into chunks of size `context_window + 1`.

```python
# Prepare text chunks
text_chunks = [text_indices[i:i+context_window+1] for i in range(len(
    text_indices)-context_window)]
```

4.2 Loss Function

We use the cross-entropy loss function.

$$\text{Loss} = -\log(p_{\text{target}}) \tag{4}$$

```python
# Softmax function
def softmax(x, dim=-1):
    x_exp = torch.exp(x - x.max(dim=dim, keepdim=True)[0])
    return x_exp / x_exp.sum(dim=dim, keepdim=True)

# Cross-entropy loss
def cross_entropy_loss(logits, target):
    log_probs = torch.log(softmax(logits, dim=-1))
    loss = -log_probs[target]
    return loss
```

4.3 Training Loop

We iterate over each text chunk and, for each position, predict the next token and update the model parameters.

```python
# Training loop
parameters = [embedding_matrix, W_Q, W_K, W_V, output_weights, output_bias]

for epoch in range(epochs):
    total_loss = 0
    for chunk in text_chunks:
        tokens = chunk
        # For positions from 1 to context_window
        for i in range(1, context_window+1):
            input_tokens = tokens[:i]
            target_token = tokens[i]

            # Embeddings
            embeddings = embedding_matrix[input_tokens]

            # Add positional encoding
            embeddings = embeddings + positional_encoding[:i]

            # Apply layernorm
            embeddings = layer_norm(embeddings)

            # Compute Q, K, V
            Q = embeddings @ W_Q
            K = embeddings @ W_K
            V = embeddings @ W_V

            # Multi-head attention
            attn_output = multi_head_attention(Q, K, V, num_heads)

            # Output projection
            logits = attn_output[-1] @ output_weights + output_bias  # (
                vocab_size)

            # Compute loss
            loss = cross_entropy_loss(logits, target_token)
            # Backpropagate
            loss.backward()

            # Update parameters
            with torch.no_grad():
                for param in parameters:
                    param -= learning_rate * param.grad
                    param.grad.zero_()

            total_loss += loss.item()
    avg_loss = total_loss / (len(text_chunks) * context_window)
    print(f"Epoch {epoch+1}, Loss: {avg_loss}")
```

5 Text Generation

After training, we can generate text by predicting one token at a time.

```
# Text generation
def generate(model_input, max_length=100):
    model_input = [char_to_idx[ch] for ch in model_input]
    generated = model_input.copy()

    for _ in range(max_length):
        input_tokens = torch.tensor(generated[-context_window:], dtype=torch.
            long)
        embeddings = embedding_matrix[input_tokens]

        seq_len = embeddings.size(0)
        embeddings = embeddings + positional_encoding[:seq_len]

        embeddings = layer_norm(embeddings)

        Q = embeddings @ W_Q
        K = embeddings @ W_K
        V = embeddings @ W_V

        attn_output = multi_head_attention(Q, K, V, num_heads)

        logits = attn_output[-1] @ output_weights + output_bias  # (vocab_size)

        probs = softmax(logits, dim=-1)

        # Select the token with the highest probability
        next_token = torch.argmax(probs).item()

        generated.append(next_token)

    generated_text = ''.join(idx_to_char[idx] for idx in generated)
    return generated_text
```

5.1 Example Usage

```
# Example usage
seed_text = "To be or not "
generated_text = generate(seed_text, max_length=100)
print("Generated text:")
print(generated_text)
```

6 Mathematical Explanation

6.1 Tokenization

We map each unique character in the text to a unique integer index. For example, if our vocabulary consists of characters $\{a, b, c\}$, we might have:

$$\begin{aligned}\text{char\_to\_idx} &= \{'a' : 0, 'b' : 1, 'c' : 2\} \\ \text{idx\_to\_char} &= \{0 : 'a', 1 : 'b', 2 : 'c'\}\end{aligned}$$

The text "abc" would then be tokenized as $[0, 1, 2]$.

6.2 Embeddings

Each token t_i is mapped to an embedding vector $e_i \in \mathbb{R}^{d_{\text{model}}}$, using the embedding matrix embedding_matrix $\in \mathbb{R}^{V \times d_{\text{model}}}$, where V is the vocabulary size and d_{model} is the embedding dimension.

$$e_i = \text{embedding\_matrix}[t_i], \quad e_i \in \mathbb{R}^{d_{\text{model}}} \tag{5}$$

6.3 Positional Encoding

We add positional information to the embeddings using fixed positional encoding. The positional encoding matrix $\text{PE} \in \mathbb{R}^{n \times d_{\text{model}}}$, where n is the maximum sequence length.

For each position *pos* and dimension $0 \leq i < d_{\text{model}}$:

$$\begin{aligned}\text{PE}_{pos,2i} &= \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \\ \text{PE}_{pos,2i+1} &= \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)\end{aligned}$$

The embeddings with positional encoding are computed as:

$$e'_i = e_i + \text{PE}_{pos_i}, \quad e'_i \in \mathbb{R}^{d_{\text{model}}} \tag{6}$$

6.4 Layer Normalization

We normalize the embeddings to have zero mean and unit variance along the embedding dimension.

For each embedding vector e'_i:

$$\begin{aligned}\mu &= \frac{1}{d_{\text{model}}} \sum_{k=1}^{d_{\text{model}}} e'_{i,k} \\ \sigma^2 &= \frac{1}{d_{\text{model}}} \sum_{k=1}^{d_{\text{model}}} (e'_{i,k} - \mu)^2 \\ \hat{e}_i &= \frac{e'_i - \mu}{\sqrt{\sigma^2 + \epsilon}}, \quad \hat{e}_i \in \mathbb{R}^{d_{\text{model}}}\end{aligned}$$

6.5 Multi-Head Attention

The multi-head attention mechanism allows the model to attend to different positions within the sequence from multiple representation subspaces.

6.5.1 Computing Queries, Keys, and Values

We project the normalized embeddings $\hat{e}_i$ to obtain the queries Q, keys K, and values V:

$$\begin{aligned} Q &= \hat{E}W_Q, \quad Q \in \mathbb{R}^{n \times d_{\text{model}}} \\ K &= \hat{E}W_K, \quad K \in \mathbb{R}^{n \times d_{\text{model}}} \\ V &= \hat{E}W_V, \quad V \in \mathbb{R}^{n \times d_{\text{model}}} \end{aligned}$$

where $\hat{E} \in \mathbb{R}^{n \times d_{\text{model}}}$ is the matrix of normalized embeddings, and $W_Q, W_K, W_V \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ are learnable weight matrices.

6.5.2 Splitting into Heads

We split Q, K, V into h heads, where each head has a dimension $d_{\text{head}} = d_{\text{model}}/h$.

Given:

- Number of tokens (sequence length): $n = 4$
- Embedding dimension (model dimension): $d_{\text{model}} = 6$
- Number of heads: $h = 2$

Therefore:

$$d_{\text{head}} = \frac{d_{\text{model}}}{h} = \frac{6}{2} = 3$$

Matrices Q, K, V are initially of size $n \times d_{\text{model}}$:

$$Q, K, V \in \mathbb{R}^{n \times d_{\text{model}}} = \mathbb{R}^{4 \times 6}$$

Splitting Process:
1. Reshape each matrix from (n, d_{model}) to (n, h, d_{head}):

$$Q \to \text{reshape to } (4, 2, 3)$$

2. Transpose to bring the head dimension forward:

$$Q \to \text{transpose to } (h, n, d_{\text{head}}) = (2, 4, 3)$$

This results in $h = 2$ separate matrices for Q, K, V, each of size $n \times d_{\text{head}}$:

$$Q^{(i)}, K^{(i)}, V^{(i)} \in \mathbb{R}^{n \times d_{\text{head}}} = \mathbb{R}^{4 \times 3}, \quad \text{for } i = 1, 2$$

Example:
Suppose Q is:

$$Q = \begin{bmatrix} q_{1,1} & q_{1,2} & q_{1,3} & q_{1,4} & q_{1,5} & q_{1,6} \\ q_{2,1} & q_{2,2} & q_{2,3} & q_{2,4} & q_{2,5} & q_{2,6} \\ q_{3,1} & q_{3,2} & q_{3,3} & q_{3,4} & q_{3,5} & q_{3,6} \\ q_{4,1} & q_{4,2} & q_{4,3} & q_{4,4} & q_{4,5} & q_{4,6} \end{bmatrix} \in \mathbb{R}^{4 \times 6}$$

We reshape Q to $(4, 2, 3)$ and transpose to $(2, 4, 3)$, resulting in:

$$Q^{(1)} = \begin{bmatrix} q_{1,1} & q_{1,2} & q_{1,3} \\ q_{2,1} & q_{2,2} & q_{2,3} \\ q_{3,1} & q_{3,2} & q_{3,3} \\ q_{4,1} & q_{4,2} & q_{4,3} \end{bmatrix}, \quad Q^{(2)} = \begin{bmatrix} q_{1,4} & q_{1,5} & q_{1,6} \\ q_{2,4} & q_{2,5} & q_{2,6} \\ q_{3,4} & q_{3,5} & q_{3,6} \\ q_{4,4} & q_{4,5} & q_{4,6} \end{bmatrix}$$

Similarly for K and V.

6.5.3 Scaled Dot-Product Attention

For each head i, we compute the attention scores and apply them to the values.

Dimensions:

- $Q^{(i)}, K^{(i)}, V^{(i)} \in \mathbb{R}^{n \times d_{\text{head}}} = \mathbb{R}^{4\times 3}$

Step-by-Step Computation:

1. Compute Unnormalized Attention Scores:

$$\text{scores}^{(i)} = \frac{Q^{(i)}(K^{(i)})^\top}{\sqrt{d_{\text{head}}}}$$

- $Q^{(i)}(K^{(i)})^\top \in \mathbb{R}^{n\times n} = \mathbb{R}^{4\times 4}$ - $\sqrt{d_{\text{head}}} = \sqrt{3}$

2. Apply Causal Mask:

$$\text{scores}^{(i)} = \text{scores}^{(i)} + M$$

- $M \in \mathbb{R}^{n\times n}$ is the causal mask (explained in the next subsection).

3. Compute Attention Weights:

$$\text{attn\_weights}^{(i)} = \text{softmax}(\text{scores}^{(i)}), \quad \text{along each row}$$

- Ensures that $\sum_{k=1}^{n} \text{attn\_weights}^{(i)}_{j,k} = 1$ for each j.

4. Compute Head Output:

$$\text{head\_output}^{(i)} = \text{attn\_weights}^{(i)} V^{(i)}$$

- $\text{head\_output}^{(i)} \in \mathbb{R}^{n\times d_{\text{head}}} = \mathbb{R}^{4\times 3}$

Example Calculation:

Let's compute $\text{scores}^{(i)}$ for head i:

- Let $Q^{(i)}$ and $K^{(i)}$ be:

$$Q^{(i)} = \begin{bmatrix} q^{(i)}_{1,1} & q^{(i)}_{1,2} & q^{(i)}_{1,3} \\ q^{(i)}_{2,1} & q^{(i)}_{2,2} & q^{(i)}_{2,3} \\ q^{(i)}_{3,1} & q^{(i)}_{3,2} & q^{(i)}_{3,3} \\ q^{(i)}_{4,1} & q^{(i)}_{4,2} & q^{(i)}_{4,3} \end{bmatrix}, \quad K^{(i)} = \begin{bmatrix} k^{(i)}_{1,1} & k^{(i)}_{1,2} & k^{(i)}_{1,3} \\ k^{(i)}_{2,1} & k^{(i)}_{2,2} & k^{(i)}_{2,3} \\ k^{(i)}_{3,1} & k^{(i)}_{3,2} & k^{(i)}_{3,3} \\ k^{(i)}_{4,1} & k^{(i)}_{4,2} & k^{(i)}_{4,3} \end{bmatrix}$$

- Compute $\text{scores}^{(i)}$ by multiplying $Q^{(i)}$ with $(K^{(i)})^\top$:

$$\text{scores}^{(i)}_{j,k} = \frac{(Q^{(i)}_{j,:}) \cdot (K^{(i)}_{k,:})^\top}{\sqrt{d_{\text{head}}}}$$

- For example, $\text{scores}^{(i)}_{2,3}$ is computed as:

$$\text{scores}^{(i)}_{2,3} = \frac{q^{(i)}_{2,1}k^{(i)}_{3,1} + q^{(i)}_{2,2}k^{(i)}_{3,2} + q^{(i)}_{2,3}k^{(i)}_{3,3}}{\sqrt{3}}$$

Applying the Mask:

- The causal mask M ensures that each position can only attend to previous positions (including itself).

6.5.4 Causal Masking

The causal mask $M \in \mathbb{R}^{n\times n}$ is defined as:

$$M_{j,k} = \begin{cases} 0, & \text{if } k \le j \\ -\infty, & \text{if } k > j \end{cases}$$

For $n = 4$, the mask M is:

$$M = \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Explanation:

- Position 1 can attend to positions ≤ 1 (only itself). - Position 2 can attend to positions ≤ 2 (positions 1 and 2). - Position 3 can attend to positions ≤ 3 (positions 1, 2, and 3). - Position 4 can attend to positions ≤ 4 (positions 1, 2, 3, and 4).

By adding M to scores$^{(i)}$, we set the scores for future positions to $-\infty$, which after applying softmax become zero.

6.5.5 Concatenation and Output Projection

After computing the attention outputs for all heads, we concatenate them to form the final output of the multi-head attention layer.

Dimensions:

- Each head_output$^{(i)} \in \mathbb{R}^{n \times d_{\text{head}}} = \mathbb{R}^{4\times 3}$ - Concatenated output:

$$\text{ConcatOutput} = \text{concat}\left(\text{head\_output}^{(1)}, \text{head\_output}^{(2)}\right) \in \mathbb{R}^{n\times d_{\text{model}}} = \mathbb{R}^{4\times 6}$$

Purpose of Multi-Head Attention:

- Parallelization: Multiple attention heads allow the model to focus on different positions and representation subspaces simultaneously. - Representation Learning: Each head can capture different types of relationships and patterns in the data. - Expressiveness: Multi-head attention increases the model's ability to capture complex patterns compared to a single attention mechanism.

Aggregation for Final Attention Score:

- The outputs from all heads are concatenated and can be optionally transformed through a linear layer to combine the information from all heads. - In our simplified model, we directly use the concatenated output for further processing.

Sample Calculation with Small Numbers:

Assume we have the following simplified $Q^{(i)}$ and $K^{(i)}$ for head 1:

$$Q^{(1)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \quad K^{(1)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Compute Unnormalized Scores for Position 3:

- For position $j = 3$:

$$\frac{Q^{(1)}_{3,:} \cdot K^{(1)}_{k,:}}{\sqrt{d_{\text{head}}}}$$

- Calculations:

$$Q^{(1)}_{3,:} = [0, 0, 1]$$

- scores$^{(1)}_{3,1} = \frac{(0)(1)+(0)(0)+(1)(0)}{\sqrt{3}} = 0$ - scores$^{(1)}_{3,2} = \frac{(0)(0)+(0)(1)+(1)(0)}{\sqrt{3}} = 0$ - scores$^{(1)}_{3,3} = \frac{(0)(0)+(0)(0)+(1)(1)}{\sqrt{3}} = \frac{1}{\sqrt{3}} \approx 0.577$ - scores$^{(1)}_{3,4} = \frac{(0)(1)+(0)(1)+(1)(1)}{\sqrt{3}} = \frac{1}{\sqrt{3}} \approx 0.577$

Apply Mask:

- For position $j = 3$, the mask allows attending to positions $k \leq 3$.
- Adjusted scores:

$$\text{scores}^{(1)}_3 = [0, 0, 0.577, -\infty]$$

Compute Attention Weights:

- Apply softmax to scores$^{(1)}_3$:

$$\text{attn\_weights}^{(1)}_{3,k} = \frac{\exp(\text{scores}^{(1)}_{3,k})}{\sum_{l=1}^{3} \exp(\text{scores}^{(1)}_{3,l})}$$

- Since $\exp(-\infty) = 0$, position 4 does not contribute.
- Calculations:

$$\exp(\text{scores}_3^{(1)}) = [1, 1, e^{0.577}, 0]$$

$$\sum_{l=1}^{3} \exp(\text{scores}_{3,l}^{(1)}) = 1 + 1 + e^{0.577} \approx 1 + 1 + 1.78 = 3.78$$

$$\text{attn\_weights}_3^{(1)} = \left[\frac{1}{3.78}, \frac{1}{3.78}, \frac{1.78}{3.78}, 0\right] \approx [0.265, 0.265, 0.47, 0]$$

Compute Head Output:
- Assume $V^{(1)} = K^{(1)}$ for simplicity.
- For position $j = 3$:

$$\text{head\_output}_3^{(1)} = \sum_{k=1}^{3} \text{attn\_weights}_{3,k}^{(1)} V_{k,:}^{(1)}$$

- Calculated as weighted sum of $V_{1,:}^{(1)}, V_{2,:}^{(1)}, V_{3,:}^{(1)}$.
Repeat for Head 2 and Other Positions:
- The same process is applied for head 2 and for other positions j.

6.5.6 Final Output

- After computing head_output$^{(1)}$ and head_output$^{(2)}$, we concatenate them:

$$\text{ConcatOutput}_j = [\text{head\_output}_j^{(1)}, \text{head\_output}_j^{(2)}] \in \mathbb{R}^{d_{\text{model}}} = \mathbb{R}^6$$

- This output can be used for further processing, such as passing through feed-forward layers or generating logits for prediction.

Key Points:

- Multi-Head Attention allows the model to attend to different positions and capture various types of dependencies. - Splitting into heads reduces the dimensionality per head, enabling the model to learn multiple representations efficiently. - Causal Masking ensures the autoregressive property, making the model suitable for sequence generation tasks.

—

By specifying matrix dimensions and providing sample calculations, we've elaborated on how multi-head attention operates within the transformer model, and how the attention scores are computed, masked, and used to generate the final output.

6.6 Output Projection

We take the output corresponding to the last token in the sequence (position n) and project it to the vocabulary size to obtain the logits for the next token prediction:

$$\text{logits} = \text{ConcatOutput}_n W_O + b_O, \quad \text{logits} \in \mathbb{R}^V$$

where $W_O \in \mathbb{R}^{d_{\text{model}} \times V}$ is the output weight matrix, and $b_O \in \mathbb{R}^V$ is the output bias.

6.7 Training Objective

Our objective is to minimize the cross-entropy loss between the predicted probabilities and the true next token:

$$\begin{aligned} p_{\text{pred}} &= \text{softmax}(\text{logits}) \\ \text{Loss} &= -\log(p_{\text{pred}}[t_{\text{target}}]) \end{aligned}$$

where t_{target} is the index of the true next token.

6.8 Backpropagation and Parameter Updates

We compute gradients with respect to the loss and update the model parameters using gradient descent:

$$\theta \leftarrow \theta - \eta \nabla_\theta \text{Loss} \tag{7}$$

where θ represents all the model parameters $(\text{embedding\_matrix}, W_Q, W_K, W_V, W_O, b_O)$ and η is the learning rate.

6.9 Purpose of Multi-Head Attention

The multi-head attention mechanism allows the model to attend to different positions within the sequence from multiple representation subspaces. Each head can focus on different types of relationships and dependencies between tokens.

By having multiple heads, the model can capture various aspects of the input sequence simultaneously, improving its ability to model complex patterns.

6.9.1 Computation Summary

1. Compute Q, K, V:

$$Q = \hat{E}W_Q, \quad K = \hat{E}W_K, \quad V = \hat{E}W_V$$

2. Split into h heads:

$$Q \to [Q^{(1)}, Q^{(2)}, \dots, Q^{(h)}], \quad \text{each } Q^{(i)} \in \mathbb{R}^{n \times d_{\text{head}}}$$

3. For each head i, compute attention:

$$\text{scores}^{(i)} = \frac{Q^{(i)}(K^{(i)})^\top}{\sqrt{d_{\text{head}}}} + M$$

$$\text{attn\_weights}^{(i)} = \text{softmax}(\text{scores}^{(i)})$$

$$\text{head\_output}^{(i)} = \text{attn\_weights}^{(i)} V^{(i)}$$

4. Concatenate heads and project:

$$\text{ConcatOutput} = [\text{head\_output}^{(1)}, \dots, \text{head\_output}^{(h)}] \in \mathbb{R}^{n \times d_{\text{model}}}$$

5. Output projection:

$$\text{logits} = \text{ConcatOutput}_n W_O + b_O$$

6. Compute loss and update parameters.

6.9.2 Aggregation of Attention Outputs

The attention outputs from each head are concatenated, not the attention scores themselves. Each head independently computes its attention output, and these outputs are combined to form the final representation.

The concatenation allows the model to integrate information from different representation subspaces. This multi-faceted representation enhances the model's capacity to understand complex relationships in the data.