Transformer Attention and Encoding

```
explain code - import torch
import torch.nn as nn
import math
import matplotlib.pyplot as plt
import seaborn as sns

# Positional Encoding
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        pe = torch.zeros(max_len, d_model)
        pos = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(pos * div_term)
        pe[:, 1::2] = torch.cos(pos * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + self.pe[:, :x.size(1)]
        return x

    def plot(self, dims=4):
        plt.figure(figsize=(10, 6))
        pos_enc = self.pe[0, :100, :dims]
        for i in range(dims):
            plt.plot(pos_enc[:, i].cpu().numpy(), label=f"dim {i}")
        plt.title("Positional Encodings")
        plt.xlabel("Position")
        plt.legend()
        plt.show()

# Attention with visualization hook
def scaled_dot_product_attention(q, k, v, mask=None, return_attn=False):
    d_k = q.size(-1)
    scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    attn = torch.softmax(scores, dim=-1)
    output = torch.matmul(attn, v)
    return (output, attn) if return_attn else output

# Multi-Head Attention with optional visualization
class MultiHeadAttention(nn.Module):
```

```
    def __init__(self, d_model, num_heads, visualize=False):
        super().__init__()
        assert d_model % num_heads == 0
        self.d_k = d_model // num_heads
        self.num_heads = num_heads
        self.visualize = visualize
        self.attn_weights = None  # For visualization

        self.q_linear = nn.Linear(d_model, d_model)
        self.k_linear = nn.Linear(d_model, d_model)
        self.v_linear = nn.Linear(d_model, d_model)
        self.out = nn.Linear(d_model, d_model)

    def forward(self, q, k, v, mask=None):
        batch_size = q.size(0)
        q = self.q_linear(q).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
        k = self.k_linear(k).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
        v = self.v_linear(v).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)

        if self.visualize:
            output, attn = scaled_dot_product_attention(q, k, v, mask, return_attn=True)
            self.attn_weights = attn.detach()  # Shape: (B, heads, seq, seq)
        else:
            output = scaled_dot_product_attention(q, k, v, mask)

        concat = output.transpose(1, 2).contiguous().view(batch_size, -1, self.num_heads * self.d_k)
        return self.out(concat)

    def plot_attention(self, index=0, head=0):
        if self.attn_weights is None:
            print("No attention weights stored.")
            return
        attn = self.attn_weights[index, head].cpu().numpy()
        plt.figure(figsize=(6, 5))
        sns.heatmap(attn, cmap="viridis")
        plt.title(f"Attention Weights (Sample {index}, Head {head})")
        plt.xlabel("Key Position")
        plt.ylabel("Query Position")
        plt.show()

# Feed Forward
class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff):
        super().__init__()
        self.linear1 = nn.Linear(d_model, d_ff)
        self.linear2 = nn.Linear(d_ff, d_model)

    def forward(self, x):
        return self.linear2(torch.relu(self.linear1(x)))

# Encoder Layer
```

```
class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1, visualize=False):
        super().__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads, visualize)
        self.feed_forward = FeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        attn_output = self.self_attn(x, x, x, mask)
        x = self.norm1(x + self.dropout(attn_output))
        ff_output = self.feed_forward(x)
        x = self.norm2(x + self.dropout(ff_output))
        return x

# Transformer Encoder
class TransformerEncoder(nn.Module):
    def __init__(self, input_dim, d_model, num_layers, num_heads, d_ff, max_len=100, visualize=False):
        super().__init__()
        self.embedding = nn.Embedding(input_dim, d_model)
        self.pos_encoder = PositionalEncoding(d_model, max_len)
        self.layers = nn.ModuleList([
            EncoderLayer(d_model, num_heads, d_ff, visualize=visualize) for _ in range(num_layers)
        ])
        self.d_model = d_model
        self.visualize = visualize

    def forward(self, src, mask=None):
        x = self.embedding(src) * math.sqrt(self.d_model)
        x = self.pos_encoder(x)
        for layer in self.layers:
            x = layer(x, mask)
        return x

# ==== RUN & VISUALIZE ====
if __name__ == "__main__":
    vocab_size = 1000
    d_model = 128
    num_heads = 4
    d_ff = 512
    num_layers = 1
    max_len = 100
    batch_size = 1
    seq_len = 20

    # Init model with visualization
    model = TransformerEncoder(input_dim=vocab_size, d_model=d_model,
                               num_layers=num_layers, num_heads=num_heads,
                               d_ff=d_ff, max_len=max_len, visualize=True)
```

```
# Dummy input
src = torch.randint(0, vocab_size, (batch_size, seq_len))
out = model(src)

# Plot Positional Encodings
print("Plotting Positional Encodings...")
model.pos_encoder.plot(dims=6)

# Plot Attention from first encoder layer
print("Plotting Attention Weights...")
model.layers[0].self_attn.plot_attention(index=0, head=0)

print("Output shape:", out.shape)
```

This code implements a simplified **Transformer Encoder** in PyTorch, complete with **positional encodings**, **multi-head self-attention**, **feedforward networks**, and **visualization tools** for understanding positional encoding and attention weights.

Let's walk through each component step-by-step:

---

1. Imports

```python
import torch
import torch.nn as nn
import math
import matplotlib.pyplot as plt
import seaborn as sns
```

- Core libraries used for tensor computations (`` `torch` ``), neural networks (`` `nn` ``), math functions (`` `math` ``), and plotting (`` `matplotlib` ``, `` `seaborn` ``).

---

2. Positional Encoding

```python
class PositionalEncoding(nn.Module):
```

Since Transformers don’t have recurrence or convolution, they rely on **positional encodings** to give the model a sense of word order.

- Uses sine and cosine functions of different frequencies to encode positions.
- `` `self.register_buffer('pe', pe)` `` ensures the encodings are part of the model but **not trainable**.
- `` `plot()` `` visualizes the positional encoding for a few dimensions.

---

3. Scaled Dot-Product Attention

```python
def scaled_dot_product_attention(q, k, v, mask=None, return_attn=False):
```

Performs the key attention mechanism:

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

- If `mask` is used, it hides certain positions (like padding or future tokens in decoding).
- Returns attention weights optionally for visualization.

---

4. Multi-Head Attention

```python
class MultiHeadAttention(nn.Module):
```

Allows the model to jointly attend to information from different representation subspaces.

- Projects `q`, `k`, `v` into multiple "heads" → applies attention → concatenates → linear layer.
- `plot_attention()` allows visualizing the attention map per head.
- `self.visualize = True` enables storage of attention weights for later inspection.

---

5. Feed Forward Network

```python
class FeedForward(nn.Module):
```

Standard MLP layer used in Transformers:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

---

6. Encoder Layer

```python
class EncoderLayer(nn.Module):
```

One layer of the Transformer encoder:

- Applies **multi-head self-attention** followed by **feedforward**.
- Uses **residual connections** + **layer normalization**.
- `Dropout` adds regularization.

---

7. Transformer Encoder

```python
class TransformerEncoder(nn.Module):
```

Stacks multiple encoder layers:

- First embeds input tokens (`nn.Embedding`) and scales them by `sqrt(d_model)`.
- Adds positional encodings.
- Applies several `EncoderLayer` modules.

---

8. Main Run & Visualization

```python
if __name__ == "__main__":
```

Runs the model and shows how to visualize its internals:

- Sets hyperparameters.
- Builds the Transformer encoder.
- Generates dummy input tokens (`src`).
- Runs a forward pass.
- Visualizes:
    - Positional encodings using `.plot()`
    - Attention weights using `.plot_attention()`

📌 Output

```python
print("Output shape:", out.shape)
```

Prints the shape of the final encoder output, which will be:

```r
(batch_size, seq_len, d_model)
```

Summary Diagram

```text
Input Tokens
      ↓
```

```
[Embedding] → [Add Positional Encoding]
      ↓
[Encoder Layer(s)]
   ├─ Multi-Head Self Attention
   ├─ Feed Forward
      ↓
Output Encoded Sequence
```

Would you like a diagram or explanation of how each tensor transforms shape during the forward pass?

write viva questions with answers

Here are **viva questions with answers** based on the Transformer Encoder code you provided, organized by topic for clarity:

🔹 General Concepts

Q1: What is a Transformer Encoder?
A: A Transformer Encoder is a neural network architecture that processes input sequences using self-attention mechanisms and feedforward layers, without recurrence or convolution. It captures relationships between all tokens in a sequence simultaneously.

Q2: Why do we need Positional Encoding in Transformers?
A: Because Transformers process the entire sequence in parallel (not sequentially), they don't inherently understand the order of tokens. Positional encodings are added to the input embeddings to inject information about the position of each token.

🔹 Positional Encoding

Q3: How is Positional Encoding implemented in this code?
A: It uses sine and cosine functions of different frequencies for each position and dimension. These values are precomputed and added to the input embeddings.

Q4: Why are sine and cosine functions used?
A: They provide a smooth and continuous way to represent positions, and their periodic nature helps the model generalize to longer sequences than seen during training.

🔹 Attention Mechanism

Q5: What is Scaled Dot-Product Attention?
A: It computes attention weights using the formula:

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

It scales the dot product of queries and keys to prevent large values and stabilize gradients.

Q6: What is the purpose of the mask in attention?
A: The mask is used to ignore specific positions (e.g., padding tokens or future tokens in decoding) by

setting their attention scores to a large negative number before softmax.

🔹 Multi-Head Attention

Q7: Why do we use multiple heads in attention?
A: Multi-head attention allows the model to learn information from different representation subspaces at different positions in the sequence, improving performance.

Q8: How are attention weights visualized in this code?
A: If `visualize=True`, the attention weights are stored during the forward pass and can be visualized using the `plot_attention()` function.

🔹 Encoder Architecture

Q9: What components does each encoder layer contain?
A: Each encoder layer has:

- Multi-head self-attention
- Feedforward neural network
- Layer normalization
- Residual connections
- Dropout

Q10: What is the role of layer normalization?
A: It stabilizes and accelerates training by normalizing the input to each sub-layer (attention or feedforward).

🔹 Code Understanding

Q11: What does `self.embedding(src) * math.sqrt(self.d_model)` do?
A: It embeds the input tokens and scales them by the square root of `d_model` to maintain a suitable variance in the embeddings.

Q12: What does the `forward()` function in `TransformerEncoder` return?
A: It returns the final encoded representation of the input sequence after passing through all encoder layers.

🔹 Visualization

Q13: How are positional encodings visualized?
A: The `plot()` function of the `PositionalEncoding` class plots selected dimensions of the encoding values for the first 100 positions.

Q14: Why is visualization useful in Transformers?
A: Visualization helps interpret and debug the model by showing how attention focuses on different parts of the input and how positions are encoded.

Would you like this in a printable format (e.g., PDF or docx), or with additional questions on training, loss functions, or decoder modules?

give advance questions

Here are **advanced viva questions with answers** related to the Transformer Encoder architecture and the provided code. These are suitable for demonstrating deep understanding in interviews or project evaluations.

🔹 Advanced Attention Mechanisms

Q1: Why is the dot product attention scaled by $\sqrt{d_k}$?
A: Without scaling, the dot product values grow large when d_k increases, which pushes the softmax into regions with very small gradients. Scaling by $\sqrt{d_k}$ keeps the gradients stable and prevents saturation of softmax.

Q2: How does multi-head attention help avoid redundancy in learned features?
A: Each head learns to focus on different aspects of the input by using different projections of Q, K, V. This enables the model to learn richer representations by capturing diverse dependencies.

🔹 Positional Encoding

Q3: Why are even and odd dimensions used differently in sinusoidal positional encoding?
A: Using sine for even indices and cosine for odd indices ensures that each dimension has a unique encoding and introduces phase shifts between dimensions, helping the model distinguish between positions.

Q4: Can positional encodings be learned instead of fixed? What are the trade-offs?
A: Yes, learned positional encodings are trainable parameters. They can adapt to specific tasks but may not generalize well to longer sequences than seen during training, unlike fixed sinusoidal encodings.

🔹 Model Design & Implementation

Q5: Why do we use residual connections in Transformer layers?
A: Residual connections help gradients flow through deep networks during backpropagation, reduce vanishing gradient issues, and allow the model to retain the original input information.

Q6: What is the purpose of using `nn.LayerNorm` instead of batch normalization?
A: `LayerNorm` normalizes across the feature dimension, making it suitable for sequences and batch-independent. BatchNorm depends on batch statistics, which can vary and harm sequence modeling.

🔹 Masking and Padding

Q7: In what scenarios is masking especially important in attention?
A: Masking is crucial during:

- Padding: to ignore padded tokens.
- Autoregressive decoding: to prevent attention to future tokens.
  This ensures valid dependencies and correct output predictions.

Q8: How would you modify the code to handle padding masks in batches with variable-length sequences?
A: You would:

1. Generate a padding mask (where 1 is real token, 0 is padding).
2. Pass it to `` `scaled_dot_product_attention` `` as the `` `mask` `` parameter.
3. Ensure the mask shape aligns with attention score dimensions.

🔹 Computational Considerations

Q9: What is the time complexity of self-attention in Transformers?
A: The time and space complexity is $O(n^2 \cdot d)$, where n is sequence length and d is embedding dimension. This can be a bottleneck for long sequences.

Q10: How can self-attention be optimized for long sequences?
A: Techniques include:

- Sparse attention (Longformer, BigBird)
- Linear attention (Performer, Linformer)
- Memory-compressed attention (Reformer)
  These reduce the quadratic complexity of standard self-attention.

🔹 Theoretical and Practical Insights

Q11: What happens if we remove positional encoding from the Transformer?
A: The model loses the ability to differentiate between token positions, severely degrading performance on tasks where order matters, like language modeling.

Q12: Explain the role of dropout in this Transformer implementation.
A: Dropout is applied after attention and feedforward layers to prevent overfitting by randomly zeroing some outputs during training, encouraging redundancy in learned representations.

Would you like questions focused on **training strategies**, **hyperparameter tuning**, or **real-world applications** of Transformers as well?