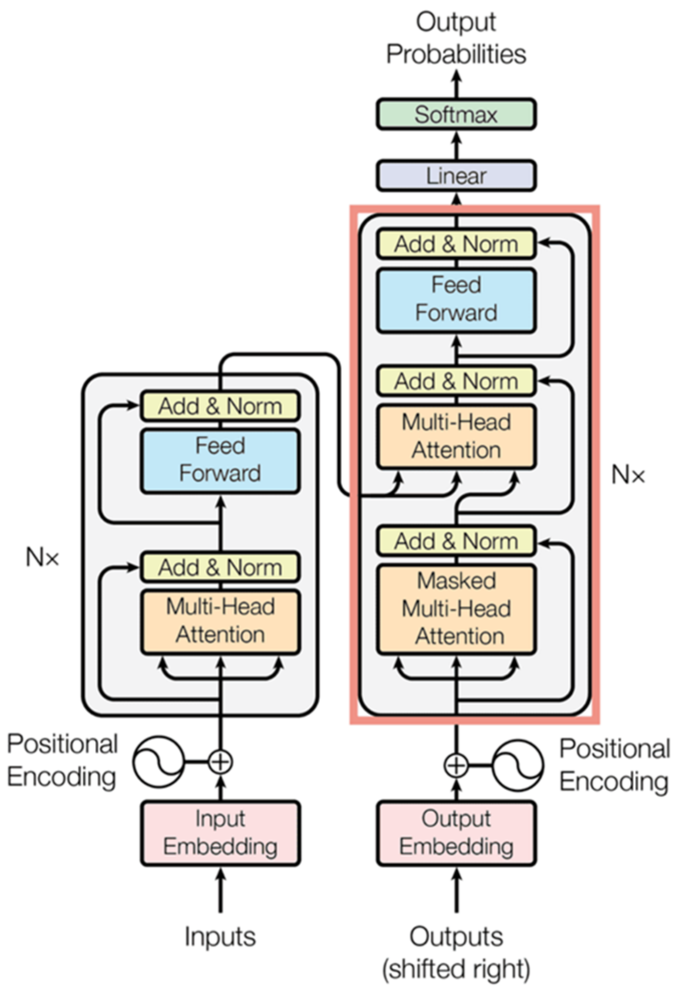

Masked Self-Attention

In the decoder component of a transformer model, masked multi-head attention is used to ensure that the model adheres to the autoregressive property during training and inference. This means that when predicting a token at a specific position in the output sequence, the model should only have access to the tokens that come before it, not the ones that come after. Let's break down why this is important and how the mask works.

Why Masking is Necessary

1. **Autoregressive Generation**:
   - The decoder's goal is to generate output sequences token by token. For each token prediction at position t, the model should only use information from tokens at positions 1 to $t-1$. This ensures that the model learns to generate sequences in a left-to-right manner, similar to how humans process language.
2. **Preventing Information Leakage**:
   - Without masking, the self-attention mechanism in the decoder would allow each token to attend to all other tokens in the sequence, including future tokens. This would give the model an unfair advantage during training, as it could "cheat" by looking at future tokens to predict the current one.

How the Mask Works

1. **Masking Future Tokens**:
   - A mask is applied to the attention scores before computing the softmax. This mask is typically an upper triangular matrix where all values above the diagonal are set to negative infinity (or a very large negative number). When softmax is applied, these masked values become zero, effectively ignoring future tokens.
2. **Mechanics of the Mask**:
   - For a sequence of length n, the mask $M \in \mathbb{R}^{n \times n}$ is defined as:

     $$M_{i,j} = \begin{cases} 0 & \text{if } i \geq j, \\ -\infty & \text{if } i < j. \end{cases}$$

   - The masked attention scores are computed as:

     $$S_{\text{masked}} = S + M$$

   - After adding the mask, the softmax function is applied to S_{masked}, ensuring that the attention weights for future tokens are zero.

Example of Masked Attention Scores

Consider a sequence of length 4. The mask would look like this:

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

When added to the raw attention scores S, the values corresponding to future tokens are set to negative infinity. After applying softmax, the attention weights for future tokens become zero.

How This Helps Sequential Learning

1. **Forcing Sequential Processing**:
   - The mask forces the model to process tokens sequentially. When predicting the token at position t, the model can only attend to tokens from positions 1 to $t - 1$. This mimics the natural way humans process language and ensures that the model learns to generate sequences step by step.
2. **Improving Generalization**:
   - During training, the mask prevents the model from relying on future tokens, which are not available during actual inference. This helps the model generalize better to real-world scenarios where future tokens are unknown.
3. **Maintaining Context**:
   - While future tokens are masked, the model can still attend to previous tokens. This allows it to use contextual information from earlier in the sequence to inform its predictions.

Implementation in Practice

Here's a simplified implementation of masked multi-head attention:

```python
import numpy as np

def create_mask(sequence_length):
    mask = np.tril(np.ones((sequence_length, sequence_length)))
    mask = np.where(mask == 0, -np.inf, 0)
    return mask

class MaskedSelfAttention:
    def __init__(self, d_model, num_heads):
        self.d_model = d_model
        self.num_heads = num_heads
        self.depth = d_model // num_heads
        self.WQ = np.random.randn(d_model, d_model)
        self.WK = np.random.randn(d_model, d_model)
        self.WV = np.random.randn(d_model, d_model)

    def split_heads(self, x, batch_size):
        x = x.reshape(batch_size, -1, self.num_heads, self.depth)
        return x.transpose(0, 2, 1, 3)

    def forward(self, query, key, value, mask):
        batch_size = query.shape[0]
        seq_length = query.shape[1]

        Q = np.dot(query, self.WQ)
        K = np.dot(key, self.WK)
        V = np.dot(value, self.WV)

        Q = self.split_heads(Q, batch_size)
        K = self.split_heads(K, batch_size)
        V = self.split_heads(V, batch_size)

        S = np.matmul(Q, K.transpose(0, 1, 3, 2)) / np.sqrt(self.depth)
        S += mask  # Apply the mask

        A = np.softmax(S, axis=-1)
        O = np.matmul(A, V)

        O = O.transpose(0, 2, 1, 3).reshape(batch_size, seq_length, self.d_model)

        return O
```

Key Takeaway

Masked multi-head attention is essential in the decoder component of transformer models because it ensures that the model adheres to the autoregressive property. By preventing the model from attending to future tokens, the mask forces the model to generate sequences step by step, using only the context from previous tokens. This is crucial for tasks like machine translation, text generation, and any other sequence-to-sequence task where the order of tokens matters.

Back Pass

In the backward pass of masked multi-head attention, the mask itself does not have any parameters to update. The mask is a static tensor used to prevent the model from attending to future tokens during the forward pass. However, the gradients of the loss with respect to the attention scores are affected by the mask because the masked positions contribute nothing to the forward pass and thus should contribute nothing to the gradients.

How the Mask Affects the Backward Pass

1. **Forward Pass Recap**:
   - The mask is added to the raw attention scores S. This effectively sets the scores for future tokens to $-\infty$.
   - After adding the mask, the softmax function is applied, which turns the masked scores into zero probabilities.
2. **Backward Pass**:
   - During backpropagation, the gradients of the loss with respect to the attention scores S are computed.
   - The mask ensures that the gradients for the masked positions (future tokens) are zero because the masked scores do not contribute to the loss.

Implementation Details

Here's how the mask is handled during the backward pass in practice:

```python
import numpy as np

class MaskedSelfAttention:
    def __init__(self, d_model, num_heads):
        self.d_model = d_model
        self.num_heads = num_heads
        self.depth = d_model // num_heads
        self.WQ = np.random.randn(d_model, d_model)
        self.WK = np.random.randn(d_model, d_model)
        self.WV = np.random.randn(d_model, d_model)
        self.dWQ = np.zeros_like(self.WQ)
        self.dWK = np.zeros_like(self.WK)
        self.dWV = np.zeros_like(self.WV)

    def split_heads(self, x, batch_size):
        x = x.reshape(batch_size, -1, self.num_heads, self.depth)
        return x.transpose(0, 2, 1, 3)

    def forward(self, query, key, value, mask):
        self.mask = mask  # Store the mask for the backward pass
        batch_size = query.shape[0]
        seq_length = query.shape[1]

        Q = np.dot(query, self.WQ)
        K = np.dot(key, self.WK)
        V = np.dot(value, self.WV)

        Q = self.split_heads(Q, batch_size)
        K = self.split_heads(K, batch_size)
        V = self.split_heads(V, batch_size)

        S = np.matmul(Q, K.transpose(0, 1, 3, 2)) / np.sqrt(self.depth)
        S += mask  # Apply the mask
        self.S = S  # Store S for backward pass

        A = np.softmax(S, axis=-1)
        self.A = A  # Store A for backward pass
        O = np.matmul(A, V)

        O = O.transpose(0, 2, 1, 3).reshape(batch_size, seq_length, self.d_model)
        return O

    def backward(self, dO, query, key, value, learning_rate):
        batch_size, seq_length, _ = dO.shape

        # Reshape dO to match the dimensions after splitting heads
        dO = dO.reshape(batch_size, seq_length, self.num_heads, self.depth)
        dO = dO.transpose(0, 2, 1, 3)

        # Backward pass through the matrix multiplication with V
        dA = np.matmul(dO, self.V.transpose(0, 1, 3, 2))
        dV = np.matmul(self.A.transpose(0, 1, 2, 3), dO)

        # Compute softmax gradient
        dS = dA * self.A  # Element-wise multiplication
        dS -= np.sum(dS, axis=-1, keepdims=True) * self.A

        # Apply the mask to the gradient (set gradients of masked positions to zero)
        dS += self.mask  # This line is illustrative; actual implementation may vary

        # Backward pass through the scaled dot-product
        dQ = np.matmul(dS, K) / np.sqrt(self.depth)
        dK = np.matmul(dS.transpose(0, 1, 3, 2), Q) / np.sqrt(self.depth)

        # Combine heads and compute gradients for the linear projections
        dQ = dQ.transpose(0, 2, 1, 3).reshape(batch_size, seq_length, self.d_model)
        dK = dK.transpose(0, 2, 1, 3).reshape(batch_size, seq_length, self.d_model)
        dV = dV.transpose(0, 2, 1, 3).reshape(batch_size, seq_length, self.d_model)

        # Update weights
        self.dWQ = np.dot(query.T, dQ)
        self.dWK = np.dot(key.T, dK)
        self.dWV = np.dot(value.T, dV)

        self.WQ -= learning_rate * self.dWQ
        self.WK -= learning_rate * self.dWK
        self.WV -= learning_rate * self.dWV

        # Compute input gradients
        dQ_input = np.dot(dQ, self.WQ.T)
        dK_input = np.dot(dK, self.WK.T)
        dV_input = np.dot(dV, self.WV.T)

        return dQ_input, dK_input, dV_input
```

Key Points:

1. **Mask Application in Forward Pass**:
   - The mask is added to the raw attention scores S during the forward pass to prevent attention to future tokens.
2. **Gradient Calculation in Backward Pass**:
   - The gradients of the loss with respect to S (denoted as dS) are computed.
   - The mask is applied to dS to ensure that gradients for masked positions are zero. This is typically done by adding the mask to dS, similar to how it was applied in the forward pass.
3. **No Parameter Update for Mask**:
   - The mask itself is a static tensor and does not have any parameters to update. It is only used to manipulate the attention scores and their gradients.

Why This Works

By **applying the mask during both the forward and backward passes**, we **<u>ensure that the model does not receive any information from future tokens during training</u>**. This is crucial for maintaining the autoregressive property of the decoder, which is essential for tasks like language generation and translation.

In practice, the mask is typically created once and reused throughout training and inference. The key takeaway is that while the mask affects the forward pass by zeroing out certain attention scores, it also indirectly affects the backward pass by ensuring that gradients for those scores are zero.

Cross Attention

Cross-attention is a crucial component in transformer-based sequence-to-sequence models, such as those used for machine translation, text generation, and other tasks where information from two different sequences needs to interact. It allows the decoder to attend to the encoder's output, thereby incorporating contextual information from the input sequence into the decoding process. Let's break down how cross-attention works in detail.

Key Components

1. **Query (Q)**:
   - Derived from the decoder's input.
   - Represents the current state of the decoder as it processes the output sequence.
   - The decoder generates queries to retrieve relevant information from the encoder's output.
2. **Key (K)** and **Value (V)**:
   - Derived from the encoder's output.
   - The keys and values contain contextual information from the entire input sequence.
   - The keys are used to compute attention scores with the queries, while the values provide the actual content that the decoder should use.

How Cross-Attention Works

1. **Input to Cross-Attention**:
   - The decoder's output from the previous layer (or the decoder's input embeddings) is used to generate queries (Q).
   - The encoder's output (from its final layer) is used to generate keys (K) and values (V).
2. **Scaled Dot-Product Attention**:
   - Compute the attention scores by taking the dot product of the query and key vectors.
   - Scale the scores by the square root of the dimension of the keys to prevent large values from pushing the softmax function into regions where it has extremely small gradients.
   - Apply the softmax function to obtain attention weights.
   - Multiply the attention weights with the value vectors to obtain the final output.
3. **Multi-Head Attention**:
   - Cross-attention is typically implemented with multiple attention heads to capture different aspects of the relationships between the input and output sequences.
   - Each head projects the queries, keys, and values into a lower-dimensional space, performs the attention computation, and then concatenates the results before passing them through a linear layer.

Why Cross-Attention is Effective

- **Context Incorporation**:
  - Cross-attention allows the decoder to dynamically focus on different parts of the input sequence when generating each token of the output sequence. This is crucial for tasks like machine translation, where the decoder needs to refer back to relevant parts of the source sentence.
- **Parallel Processing**:
  - While the encoder processes the input sequence in parallel and the decoder generates the output sequence autoregressively (one token at a time), cross-attention provides a mechanism for the decoder to access the entire input sequence's context in parallel.

Implementation Example

Here's a more detailed implementation of cross-attention:

```python
import numpy as np

class CrossAttention:
    def __init__(self, d_model, num_heads):
        self.d_model = d_model
        self.num_heads = num_heads
        self.depth = d_model // num_heads

        # Weights for query, key, and value projections
        self.WQ = np.random.randn(d_model, d_model) * np.sqrt(2.0 / d_model)
        self.WK = np.random.randn(d_model, d_model) * np.sqrt(2.0 / d_model)
        self.WV = np.random.randn(d_model, d_model) * np.sqrt(2.0 / d_model)

    def split_heads(self, x, batch_size):
        x = x.reshape(batch_size, -1, self.num_heads, self.depth)
        return x.transpose(0, 2, 1, 3)

    def forward(self, query, key, value):
        batch_size = query.shape[0]

        # Linear projections
        Q = np.dot(query, self.WQ)
        K = np.dot(key, self.WK)
        V = np.dot(value, self.WV)

        # Split into heads
        Q = self.split_heads(Q, batch_size)
        K = self.split_heads(K, batch_size)
        V = self.split_heads(V, batch_size)

        # Scaled dot-product attention
        S = np.matmul(Q, K.transpose(0, 1, 3, 2)) / np.sqrt(self.depth)
        A = np.softmax(S, axis=-1)

        # Compute output
        O = np.matmul(A, V)

        # Combine heads
        O = O.transpose(0, 2, 1, 3).reshape(batch_size, -1, self.d_model)

        return O
```

Step-by-Step Explanation

1. **Linear Projections**:
   - The query, key, and value vectors are projected into a higher-dimensional space using learned weight matrices (WQ, WK, WV). This allows the model to capture complex relationships between the input and output sequences.
2. **Splitting into Heads**:
   - The projected vectors are split into multiple heads to perform parallel attention computations. Each head focuses on different aspects of the relationships between the sequences.
3. **Scaled Dot-Product Attention**:
   - Compute the dot product of queries and keys to get raw attention scores.
   - Scale the scores to prevent large values from pushing the softmax function into regions with very small gradients.
   - Apply softmax to obtain normalized attention weights.
   - Multiply the attention weights with the value vectors to get the final output.

Advantages

- **Rich Contextual Information**:
  - Cross-attention enables the decoder to access the entire input sequence's context, which is essential for generating coherent and contextually relevant outputs.
- **Parallel Computation**:
  - Unlike recurrent neural networks, cross-attention allows for parallel computation of attention scores, making the model more efficient to train and deploy.
- **Flexibility**:
  - Cross-attention can be used in various transformer-based architectures and can be adapted to different sequence-to-sequence tasks.

Cross-attention is a fundamental component of transformer architectures that enables effective information exchange between the encoder and decoder. By allowing the decoder to dynamically focus on relevant parts of the input sequence, cross-attention significantly enhances the model's ability to generate accurate and contextually appropriate outputs.

Back Pass through Cross-Attention

During the backward pass of a transformer model with a decoder and encoder, gradients are propagated through multiple layers, including the encoder. Here's a detailed explanation of how gradients are calculated and propagated back to the encoder:

Key Concepts:

1. **Chain Rule**: Gradients flow backward through the network using the chain rule, starting from the loss and moving through each layer.
2. **Cross-Attention Layers**: In the decoder, cross-attention layers use queries from the decoder and keys/values from the encoder. During the backward pass, gradients from the decoder's cross-attention layers are propagated back to the encoder.
3. **Multiple Stacks**: Transformers typically have multiple layers (stacks) in both the encoder and decoder. Each layer contributes to the gradients that are accumulated and propagated backward.

Step-by-Step Explanation:

1. **Loss Calculation**:
   - The loss is calculated based on the model's output and the target sequence.
2. **Decoder Backward Pass**:
   - The loss gradient is propagated backward through the decoder's layers. This includes the masked self-attention layers and feed-forward layers.
   - For each cross-attention layer in the decoder:
     - Gradients are computed for the queries (from the decoder) and keys/values (from the encoder).

    - The gradients for the keys and values are propagated back to the encoder's output.

3. **Gradient Propagation to Encoder**:
    - The gradients from the decoder's cross-attention layers are accumulated and passed to the encoder. This happens because the keys and values used in cross-attention are derived from the encoder's output.
    - The encoder's output is used as input to the cross-attention layers in the decoder. Therefore, the gradients from the decoder's cross-attention layers are propagated back through these keys and values to the encoder's output.
4. **Encoder Backward Pass**:
    - The accumulated gradients from the decoder are used to compute gradients for the encoder's layers. This includes the encoder's self-attention layers and feed-forward layers.
    - The gradients are propagated backward through each encoder layer, updating the model's parameters.

Implementation Details:

Here's a simplified illustration of how gradients flow from the decoder to the encoder:

```
# Forward pass
encoder_output = encoder(input_seq)
decoder_output = decoder(target_seq, encoder_output)

# Loss calculation
loss = compute_loss(decoder_output, target_labels)

# Backward pass
loss.backward()

# The gradients for the encoder are accumulated through the cross-attention layers
encoder_gradients = encoder.get_gradients()
decoder_gradients = decoder.get_gradients()
```

Key Points:

- **Cross-Attention Layers**: These layers in the decoder are crucial for propagating gradients back to the encoder. The keys and values from the encoder's output are used in these layers, so their gradients are computed and passed backward.
- **Accumulation of Gradients**: Gradients from multiple layers (both in the encoder and decoder) are accumulated to update the model's parameters.
- **Parameter Updates**: After gradients are computed for all layers, the model's parameters (weights) are updated using an optimization algorithm like SGD or Adam.

Summary:

The gradients of the encoder are collected during the backward pass by propagating the loss gradient through the decoder's cross-attention layers. These layers use the encoder's output as keys and values, so their gradients naturally flow back to the encoder. This process ensures that the entire model, including the encoder, is trained based on the task's loss.

Inference Process

The inference process in a transformer model involves generating output sequences based on input sequences. Below is a step-by-step explanation of this process:

Step 1: Input Processing

The input sequence is processed by the encoder. Each token in the input sequence is converted into an embedding and then augmented with positional information using positional encoding. The encoder consists of multiple layers, each comprising a multi-head self-attention mechanism and a position-wise feed-forward network. The self-attention mechanism allows the model to weigh the importance of different words in the input sequence relative to each other.

Step 2: Encoder Processing

The encoder processes the input embeddings through multiple layers. In each layer:

- **Self-Attention Mechanism**: Computes attention scores between all words in the input sequence. This helps the model understand the relationships and dependencies between words.
- **Position-wise Feed-Forward Network**: Applies a feed-forward neural network to each position individually, allowing the model to learn complex patterns and transformations of the input features.

Step 3: Decoder Initialization

The decoder is initialized with a start-of-sequence (SOS) token. This token signals the start of the output sequence generation.

Step 4: Iterative Output Generation

The decoder generates the output sequence token by token. For each token generation step:

- **Masked Self-Attention**: The decoder uses masked self-attention to ensure that each token is generated based only on the previous tokens in the output sequence. This prevents the model from cheating by looking at future tokens during generation.

- **Cross-Attention**: The decoder attends to the encoder's output through cross-attention. This allows the decoder to focus on relevant parts of the input sequence when generating each token.
- **Position-wise Feed-Forward Network**: Similar to the encoder, the decoder uses a feed-forward network to transform the features at each position.

Step 5: Output Generation

After processing through the decoder layers, the final output is passed through a linear layer followed by a softmax function to produce a probability distribution over the vocabulary. The token with the highest probability is selected as the next token in the output sequence.

Step 6: Termination

The generation process continues until an end-of-sequence (EOS) token is generated or a predefined maximum length is reached.

Summary

The transformer model's inference process involves the encoder processing the entire input sequence and the decoder generating the output sequence token by token. The decoder uses masked self-attention to ensure causal generation and cross-attention to incorporate information from the input sequence. This process continues iteratively until the output sequence is fully generated.