Self-Attention Gradients Calculation for Back Pass

Calculating the gradients of the attention mechanism is a crucial part of training transformer models. Below is a detailed explanation of how the gradients are calculated for the scaled dot-product attention mechanism, which is the standard attention mechanism used in transformer architectures.

Scaled Dot-Product Attention

The scaled dot-product attention is defined as:

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

where:

- Q is the query matrix
- K is the key matrix
- V is the value matrix
- d_k is the dimension of the keys

Forward Pass

1. **Compute the attention scores**:

$$S = \frac{QK^T}{\sqrt{d_k}}$$

2. **Apply softmax to get attention weights**:

$$A = \text{softmax}(S)$$

3. **Multiply by values to get the output**:

$$O = AV$$

Backward Pass (Gradient Calculation)

To compute the gradients, we need to consider the chain rule of differentiation. Let's denote the loss function as L.

1. Gradient of the Loss with respect to the Output O

$$\frac{\partial L}{\partial O} = \frac{\partial L}{\partial O}$$

This is typically provided by the subsequent layers or the loss function.

2. Gradient of the Loss with respect to the Attention Weights A

$$\frac{\partial L}{\partial A} = \frac{\partial L}{\partial O} \cdot V^T$$

This comes from the chain rule applied to $O = AV$.

3. Gradient of the Loss with respect to the Softmax Input S

The softmax function is applied row-wise. For each row S_i of S, the derivative of the softmax function is:

$$\frac{\partial A_i}{\partial S_i} = A_i \cdot (I - A_i^T)$$

where I is the identity matrix. This results in a Jacobian matrix for each row. However, in practice, we use the following formula for the gradient:

$$\frac{\partial L}{\partial S} = \left(\frac{\partial L}{\partial A}\right) \odot A - A\left(\frac{\partial L}{\partial A}\right)^T \cdot A$$

where $\odot$ denotes element-wise multiplication. This formula accounts for the fact that the softmax function introduces dependencies between the elements of S.

4. Gradient of the Loss with respect to the Scaled Dot-Product $\frac{QK^T}{\sqrt{d_k}}$

$$\frac{\partial L}{\partial \left(\frac{QK^T}{\sqrt{d_k}}\right)} = \frac{\partial L}{\partial S}$$

5. Gradient of the Loss with respect to Q and K

The scaled dot-product $\frac{QK^T}{\sqrt{d_k}}$ can be seen as a matrix multiplication. The gradients with respect to Q and K are:

$$\frac{\partial L}{\partial Q} = \frac{1}{\sqrt{d_k}} \frac{\partial L}{\partial \left(\frac{QK^T}{\sqrt{d_k}}\right)} \cdot K$$

$$\frac{\partial L}{\partial K} = \frac{1}{\sqrt{d_k}} Q^T \cdot \frac{\partial L}{\partial \left(\frac{QK^T}{\sqrt{d_k}}\right)}$$

6. Gradient of the Loss with respect to V

$$\frac{\partial L}{\partial V} = A^T \cdot \frac{\partial L}{\partial O}$$

Summary of the Gradient Formulas

1. **Gradient with respect to V**:

$$\frac{\partial L}{\partial V} = A^T \cdot \frac{\partial L}{\partial O}$$

2. **Gradient with respect to Q**:

$$\frac{\partial L}{\partial Q} = \frac{1}{\sqrt{d_k}} \frac{\partial L}{\partial S} \cdot K$$

3. **Gradient with respect to K**:

$$\frac{\partial L}{\partial K} = \frac{1}{\sqrt{d_k}} Q^T \cdot \frac{\partial L}{\partial S}$$

4. **Gradient with respect to S**:

$$\frac{\partial L}{\partial S} = \left(\frac{\partial L}{\partial A}\right) \odot A - A\left(\frac{\partial L}{\partial A}\right)^T \cdot A$$

5. **Gradient with respect to A**:

$$\frac{\partial L}{\partial A} = \frac{\partial L}{\partial O} \cdot V^T$$

How These Formulas Are Derived

1. **Chain Rule**: The gradients are derived using the chain rule of calculus, which allows us to compute the derivative of a composite function by multiplying the derivatives of each individual function.
2. **Softmax Derivative**: The derivative of the softmax function is derived using the properties of the softmax function, which introduces dependencies between the elements of the input vector. The derivative of the softmax function with respect to its input is a Jacobian matrix that accounts for these dependencies.
3. **Matrix Calculus**: The gradients with respect to Q, K, and V are derived using matrix calculus, specifically the rules for differentiating matrix products.

These gradient calculations are typically implemented efficiently in deep learning frameworks like PyTorch or TensorFlow, which handle the automatic differentiation. However, understanding these formulas is essential for implementing custom attention mechanisms or debugging training issues.

Implementing the self-attention mechanism from scratch, including both the forward and backward passes, involves several steps. Below is a Python implementation using NumPy. This implementation will include the scaled dot-product attention and its backward pass.

Self-Attention Forward and Backward Pass

```
import numpy as np

class SelfAttention:
    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)
```

```python
        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)

        # Gradients for the weights
        self.dWQ = np.zeros_like(self.WQ)
        self.dWK = np.zeros_like(self.WK)
        self.dWV = np.zeros_like(self.WV)

        # Intermediate values for backward pass
        self.Q = None
        self.K = None
        self.V = None
        self.A = None
        self.S = None
        self.O = None

    def split_heads(self, x, batch_size):
        # Split the embedding into self.num_heads
        x = x.reshape(batch_size, -1, self.num_heads, self.depth)
        # Transpose to (batch_size, num_heads, seq_length, 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)

        # Store intermediate values for backward pass
        self.Q = Q
        self.K = K
        self.V = V
        self.A = A
        self.S = S
        self.O = O

        return O

    def backward(self, dO, query, key, value, learning_rate):
        batch_size = dO.shape[0]
        seq_length = dO.shape[1]
```

```

        # 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)

        # Backward pass through the softmax
        dS = dA * self.A  # Element-wise multiplication
        dS = dS - np.mean(dS, axis=-1, keepdims=True)  # Subtract mean for numerical stability

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

        # Combine heads
        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)

        # Backward pass through the linear projections
        self.dWQ = np.dot(query.T, dQ)
        self.dWK = np.dot(key.T, dK)
        self.dWV = np.dot(value.T, dV)

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

        # Compute gradients for the input
        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

# Example usage
d_model = 128
num_heads = 4
batch_size = 32
seq_length = 10

# Random input data
query = np.random.randn(batch_size, seq_length, d_model)
key = np.random.randn(batch_size, seq_length, d_model)
value = np.random.randn(batch_size, seq_length, d_model)

# Initialize self-attention layer
attention = SelfAttention(d_model, num_heads)

# Forward pass
O = attention.forward(query, key, value)

# Simulate a loss gradient (for demonstration purposes)
```

```python
dO = np.random.randn(batch_size, seq_length, d_model)

# Backward pass
dQ_input, dK_input, dV_input = attention.backward(dO, query, key, value,
learning_rate=0.001)

print("Output shape:", O.shape)
print("Gradient shapes:", dQ_input.shape, dK_input.shape, dV_input.shape)
```

Explanation:

1. **Initialization**:
   - The `SelfAttention` class initializes the weights for query (W_Q), key (W_K), and value (W_V) projections.
   - The depth of each head is calculated as $\text{d\_model}/\text{num\_heads}$.
2. **Forward Pass**:
   - **Linear Projections**: The input sequences are projected into query, key, and value spaces using the learned weights.
   - **Split Heads**: The projected vectors are split into multiple heads to parallelize attention computations.
   - **Scaled Dot-Product Attention**: The attention scores are computed, scaled, and passed through a softmax function to obtain attention weights.
   - **Output Computation**: The attention weights are used to compute the weighted sum of the value vectors.
3. **Backward Pass**:
   - **Gradient Calculation**: The gradients of the loss with respect to the output are propagated backward through the attention mechanism.
   - **Softmax Gradient**: The gradient of the softmax function is computed to update the attention weights.
   - **Matrix Multiplication Gradients**: Gradients are computed for the matrix multiplications involving query, key, and value matrices.
   - **Weight Updates**: The gradients are used to update the weights W_Q, W_K, and W_V.

This implementation provides a basic understanding of how self-attention works and how gradients are computed during backpropagation. In practice, deep learning frameworks like PyTorch or TensorFlow handle these operations more efficiently and provide automatic differentiation.

Optimizing the backward pass of the self-attention mechanism can be achieved through several techniques, including vectorization, memory optimization, and leveraging efficient matrix operations. Below is an optimized implementation of the backward pass for the self-attention mechanism, along with an explanation of the optimization techniques used.

Optimized Backward Pass Implementation

```python
import numpy as np

class OptimizedSelfAttention:
    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)

        # Gradients for the weights
        self.dWQ = np.zeros_like(self.WQ)
        self.dWK = np.zeros_like(self.WK)
        self.dWV = np.zeros_like(self.WV)

        # Intermediate values for backward pass
        self.Q = None
        self.K = None
        self.V = None
        self.A = None
        self.S = None
        self.O = None

    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]

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

        O = np.matmul(A, V)
        O = O.transpose(0, 2, 1, 3).reshape(batch_size, -1, self.d_model)

        self.Q = Q
        self.K = K
        self.V = V
        self.A = A
        self.S = S
        self.O = O

        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)

```

```python
        # Optimize memory usage by reusing intermediate variables
        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 efficiently
        dS = self.A * dA
        dS -= np.mean(dS, axis=-1, keepdims=True)

        # Compute gradients for Q and K efficiently
        dQ = np.matmul(dS, self.K) / np.sqrt(self.depth)
        dK = np.matmul(dS.transpose(0, 1, 3, 2), self.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 using optimized matrix multiplications
        self.dWQ = np.dot(query.T, dQ)
        self.dWK = np.dot(key.T, dK)
        self.dWV = np.dot(value.T, dV)

        # Apply learning rate and update weights
        self.WQ -= learning_rate * self.dWQ
        self.WK -= learning_rate * self.dWK
        self.WV -= learning_rate * self.dWV

        # Compute input gradients efficiently
        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

# Example usage
d_model = 128
num_heads = 4
batch_size = 32
seq_length = 10

query = np.random.randn(batch_size, seq_length, d_model)
key = np.random.randn(batch_size, seq_length, d_model)
value = np.random.randn(batch_size, seq_length, d_model)

attention = OptimizedSelfAttention(d_model, num_heads)
O = attention.forward(query, key, value)
dO = np.random.randn(batch_size, seq_length, d_model)
dQ_input, dK_input, dV_input = attention.backward(dO, query, key, value,
learning_rate=0.001)

print("Output shape:", O.shape)
print("Gradient shapes:", dQ_input.shape, dK_input.shape, dV_input.shape)
```

Optimization Techniques Explained

1. **Vectorization and Batch Operations**:
   - **Matrix Multiplications**: Using `np.matmul` for batch matrix multiplications instead of loops. This leverages highly optimized BLAS (Basic Linear Algebra Subprograms) libraries under the hood.
   - **Reshaping and Transposing**: Efficiently reshaping and transposing arrays to leverage contiguous memory access patterns, which is faster than non-contiguous access.
2. **Memory Reuse**:
   - **Intermediate Variables**: Reusing intermediate variables like `dA` and `dS` to avoid redundant computations. This reduces memory footprint and computation time.
3. **Efficient Softmax Gradient**:
   - **Direct Computation**: Computing the softmax gradient directly using element-wise operations instead of constructing the full Jacobian matrix. This is more memory-efficient and faster.
4. **Parallelization**:
   - **Head Parallelization**: Processing all attention heads in parallel by keeping them in the batch dimension. This allows for efficient utilization of modern hardware (GPUs/TPUs) which excel at parallel computations.
5. **Reduced Precision (Optional)**:
   - **Mixed Precision**: Using mixed precision training (not shown here) where lower precision (e.g., float16) is used for certain operations to reduce memory usage and accelerate computations. This requires hardware support and careful implementation to maintain numerical stability.
6. **Gradient Clipping (Optional)**:
   - **Stabilizing Gradients**: Clipping gradients to prevent exploding gradients, which can destabilize training. This involves scaling gradients if their norm exceeds a certain threshold.
7. **Sparse Updates (Optional)**:
   - **Sparse Activation**: If certain activations are sparse, leveraging sparse matrix operations can significantly reduce computation. However, in self-attention, the attention matrix is typically dense, so this is less applicable.
8. **Kernel Fusion (Advanced)**:
   - **Fused Operations**: Combining multiple operations into a single kernel (not applicable in pure NumPy but relevant in framework-specific optimizations like TensorFlow or PyTorch). This minimizes memory transfers and kernel launch overhead.

Performance Comparison

The optimized implementation reduces computation time primarily through:

- **Efficient Matrix Operations**: Leveraging `np.matmul` which is highly optimized.
- **Memory Access Patterns**: Contiguous memory access during reshaping and transposing.
- **Reduced Redundant Computations**: Direct computation of gradients without constructing full Jacobians.

In practice, frameworks like PyTorch and TensorFlow automatically apply many of these optimizations through their backends (e.g., using cuBLAS on NVIDIA GPUs). Implementing these optimizations manually in NumPy is primarily for educational purposes, as deep learning frameworks handle these details internally.

Derivation of the softmax gradient for the attention weights

Let's clarify the derivation of the softmax gradient for the attention weights in the self-attention mechanism.

Softmax Function and Its Derivative

Given a vector of raw attention scores $S_i = [S_{i,1}, S_{i,2}, \ldots, S_{i,n}]$ for the i-th query position, the softmax function computes the attention weights $A_i = [A_{i,1}, A_{i,2}, \ldots, A_{i,n}]$ as follows:

$$A_{i,j} = \frac{e^{S_{i,j}}}{\sum_{k=1}^{n} e^{S_{i,k}}}$$

To compute the derivative of $A_{i,j}$ with respect to $S_{i,j}$ and $S_{i,k}$ (where $k \neq j$), we use the following results from the softmax derivative:

1. **Derivative of $A_{i,j}$ with respect to $S_{i,j}$:**

$$\frac{\partial A_{i,j}}{\partial S_{i,j}} = A_{i,j}(1 - A_{i,j})$$

2. **Derivative of $A_{i,j}$ with respect to $S_{i,k}$ (where $k \neq j$):**

$$\frac{\partial A_{i,j}}{\partial S_{i,k}} = -A_{i,j}A_{i,k}$$

Jacobian Matrix for a Single Row

The Jacobian matrix J_i for the i-th row of attention weights A_i with respect to the i-th row of raw attention scores S_i is:

$$J_i = \begin{bmatrix} \frac{\partial A_{i,1}}{\partial S_{i,1}} & \frac{\partial A_{i,1}}{\partial S_{i,2}} & \cdots & \frac{\partial A_{i,1}}{\partial S_{i,n}} \\ \frac{\partial A_{i,2}}{\partial S_{i,1}} & \frac{\partial A_{i,2}}{\partial S_{i,2}} & \cdots & \frac{\partial A_{i,2}}{\partial S_{i,n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial A_{i,n}}{\partial S_{i,1}} & \frac{\partial A_{i,n}}{\partial S_{i,2}} & \cdots & \frac{\partial A_{i,n}}{\partial S_{i,n}} \end{bmatrix}$$

Each element of J_i is:

$$J_{i,j,k} = \frac{\partial A_{i,j}}{\partial S_{i,k}} = \begin{cases} A_{i,j}(1 - A_{i,j}) & \text{if } j = k, \\ -A_{i,j}A_{i,k} & \text{if } j \neq k. \end{cases}$$

Gradient of Loss with Respect to S

Given the gradient of the loss L with respect to the attention weights A, denoted as $\frac{\partial L}{\partial A}$, the gradient of L with respect to the raw attention scores S can be computed using the chain rule:

$$\frac{\partial L}{\partial S} = \frac{\partial L}{\partial A} \cdot J$$

Where J is the Jacobian matrix. However, explicitly constructing J is computationally expensive. Instead, we use the following efficient formula:

$$\frac{\partial L}{\partial S} = \left(\frac{\partial L}{\partial A}\right) \odot A - A \cdot \left(\frac{\partial L}{\partial A}\right)^T \cdot A$$

Implementation in Code

Let's break down the code implementation:

```
dS = dA * self.A  # Element-wise multiplication for the diagonal terms
dS -= np.sum(dS, axis=-1, keepdims=True) * self.A  # Subtract the outer product term
```

1. **First Line: `dS = dA * self.A`**
   - This computes the element-wise product of $\frac{\partial L}{\partial A}$ and A, which corresponds to the diagonal terms of the Jacobian matrix.
2. **Second Line: `dS -= np.sum(dS, axis=-1, keepdims=True) * self.A`**
   - This subtracts the outer product term, which accounts for the off-diagonal elements of the Jacobian matrix. The term $\np.sum(dS, axis = -1, keepdims = True)$ computes the sum of the current gradients along the last axis, and multiplying by A gives the contribution from all off-diagonal terms.

Explanation

- **Diagonal Terms**: The first term $\frac{\partial L}{\partial A} \odot A$ captures how changes in $S_{i,j}$ affect $A_{i,j}$.
- **Off-Diagonal Terms**: The second term $-A \cdot \left(\frac{\partial L}{\partial A}\right)^T \cdot A$ captures how changes in $S_{i,k}$ (where $k \neq j$) affect $A_{i,j}$.

This efficient computation avoids constructing the full Jacobian matrix, making it feasible for large sequence lengths. It ensures that the gradients are correctly propagated backward through the softmax operation in the self-attention mechanism.

Further Interpreting of Attention Mechanism

Let's break down the **query (Q)**, **key (K)**, and **value (V)** matrices in the Transformer's attention mechanism step by step.

---

1. Why are W^Q, W^K, W^V 2D matrices?

These are **weight matrices** used to project the input embeddings into **Query (Q)**, **Key (K)**, and **Value (V)** spaces.

- **Shape**: If the input X has shape `(seq_len, d_model)`, then:
  - W^Q, W^K, W^V each have shape `(d_model, d_k)` (or `d_v` for W^V).
  - Typically, $d_k = d_v = d_{model}/h$ (where h = number of attention heads).

Why 2D?

- They perform a **linear transformation** (matrix multiplication) to map input embeddings into different subspaces:
  - $Q = X \cdot W^Q \rightarrow$ Projects input into "query" space (what the token is looking for).
  - $K = X \cdot W^K \rightarrow$ Projects input into "key" space (what the token can offer).
  - $V = X \cdot W^V \rightarrow$ Projects input into "value" space (actual content to be retrieved).

---

2. Why is $Q = X \cdot W^Q$ (Dot Product)?

The dot product computes how much one vector "aligns" with another. Here:

- X = Input sequence (each row is a token's embedding).
- W^Q = Learned weights that transform X into queries.

Interpretation:

- The **query (Q)** represents "what a token is interested in."
- The **key (K)** represents "what a token contains."
- The **value (V)** represents "the actual information to retrieve."

Attention Calculation:

1. Compute attention scores:

$$\text{Scores} = Q \cdot K^T$$

   - Measures similarity between queries and keys (higher score = more relevant).
2. Apply softmax to get attention weights (probabilities).
3. Retrieve weighted values:

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

---

3. Intuitive Meaning of the Calculation

- **Query (Q)**: "What am I looking for?"
  - Example: In the sentence *"The cat sat on the mat"*, if the current word is *"sat"*, its query might look for *"what action?"* or *"where?"*.
- **Key (K)**: "What information do I have?"
  - The key for *"mat"* might represent *"a place to sit."*
- **Value (V)**: "What should I return if selected?"
  - The actual embedding of *"mat"* (the object itself).

Why Dot Product?

- The dot product $Q \cdot K^T$ measures **compatibility** between queries and keys:
  - High score → The key matches the query well (e.g., *"sat"* and *"mat"* relate).
  - Low score → Irrelevant (e.g., *"sat"* and *"cat"* are less related).

Why Divide by $\sqrt{d_k}$?

- To prevent dot products from growing too large (which could make softmax gradients too small).

---

4. Summary

Component	Role	Calculation
W^Q, W^K, W^V	Project input into query/key/value spaces	Learned during training
$Q = XW^Q$	What the token is searching for	Dot product = alignment measure
$K = XW^K$	What the token can provide	Used to compute attention scores
$V = XW^V$	Actual content to retrieve	Weighted sum based on attention

This mechanism allows each token to **dynamically focus** on the most relevant parts of the input sequence, enabling context-aware representations.

Here are the **shapes of key variables** in the attention mechanism and their corresponding **gradients during backpropagation**. We'll consider the multi-head self-attention case used in Transformers.

---

1. Variable Shapes in Self-Attention

Assume:

- Input: $X \in \mathbb{R}^{n \times d_{\text{model}}}$
  - n = sequence length, d_{model} = embedding dimension.
- Number of heads: h, and $d_k = d_v = d_{\text{model}}/h$.

Variable	Shape	Description
X	$(n \times d_{\text{model}})$	Input embeddings
W^Q	$(d_{\text{model}} \times d_k)$	Query weights
W^K	$(d_{\text{model}} \times d_k)$	Key weights

Variable	Shape	Description
W^V	$(d_{\text{model}} \times d_v)$	Value weights
$Q = XW^Q$	$(n \times d_k)$	Query matrix
$K = XW^K$	$(n \times d_k)$	Key matrix
$V = XW^V$	$(n \times d_v)$	Value matrix
QK^T	$(n \times n)$	Raw attention scores
$A = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$	$(n \times n)$	Attention weights
$\text{Output} = AV$	$(n \times d_v)$	Contextual embeddings

2. Gradients During Backpropagation

During backpropagation, gradients flow from the loss $\mathcal{L}$ backward through the attention mechanism. Below are key gradients:

(1) Gradient w.r.t. Attention Output $O = AV$

$$\frac{\partial \mathcal{L}}{\partial O} \in \mathbb{R}^{n \times d_v}$$

- This comes from the next layer (e.g., feed-forward network).

(2) Gradients w.r.t. A and V

$$\frac{\partial \mathcal{L}}{\partial A} = \frac{\partial \mathcal{L}}{\partial O} V^T \quad \in \mathbb{R}^{n \times n}$$

$$\frac{\partial \mathcal{L}}{\partial V} = A^T \frac{\partial \mathcal{L}}{\partial O} \quad \in \mathbb{R}^{n \times d_v}$$

(3) Gradients w.r.t. Softmax Scores $S = \frac{QK^T}{\sqrt{d_k}}$

Let $A = \text{softmax}(S)$. The gradient is:

$$\frac{\partial \mathcal{L}}{\partial S} = A \circ \left(\frac{\partial \mathcal{L}}{\partial A} - \sum_{i=1}^{n} A_i \frac{\partial \mathcal{L}}{\partial A_i} \right) \quad \in \mathbb{R}^{n \times n}$$

(Where ∘ = Hadamard product.)

(4) Gradients w.r.t. Q, K, and V

$$\frac{\partial \mathcal{L}}{\partial Q} = \frac{1}{\sqrt{d_k}} \frac{\partial \mathcal{L}}{\partial S} K \quad \in \mathbb{R}^{n \times d_k}$$

$$\frac{\partial \mathcal{L}}{\partial K} = \frac{1}{\sqrt{d_k}} \frac{\partial \mathcal{L}}{\partial S}^T Q \quad \in \mathbb{R}^{n \times d_k}$$

(5) Gradients w.r.t. Weight Matrices W^Q, W^K, W^V

$$\frac{\partial \mathcal{L}}{\partial W^Q} = X^T \frac{\partial \mathcal{L}}{\partial Q} \quad \in \mathbb{R}^{d_{\text{model}} \times d_k}$$

$$\frac{\partial \mathcal{L}}{\partial W^K} = X^T \frac{\partial \mathcal{L}}{\partial K} \quad \in \mathbb{R}^{d_{\text{model}} \times d_k}$$

$$\frac{\partial \mathcal{L}}{\partial W^V} = X^T \frac{\partial \mathcal{L}}{\partial V} \quad \in \mathbb{R}^{d_{\text{model}} \times d_v}$$

(6) Gradient w.r.t. Input X

$$\frac{\partial \mathcal{L}}{\partial X} = \frac{\partial \mathcal{L}}{\partial Q} W^{Q^T} + \frac{\partial \mathcal{L}}{\partial K} W^{K^T} + \frac{\partial \mathcal{L}}{\partial V} W^{V^T} \quad \in \mathbb{R}^{n \times d_{\text{model}}}$$

---

3. Summary of Gradients

Gradient	Shape	Description
$\frac{\partial \mathcal{L}}{\partial O}$	$(n \times d_v)$	From downstream layer
$\frac{\partial \mathcal{L}}{\partial A}$	$(n \times n)$	Gradient through softmax
$\frac{\partial \mathcal{L}}{\partial V}$	$(n \times d_v)$	Gradient to values
$\frac{\partial \mathcal{L}}{\partial S}$	$(n \times n)$	Gradient to pre-softmax scores
$\frac{\partial \mathcal{L}}{\partial Q}$	$(n \times d_k)$	Gradient to queries
$\frac{\partial \mathcal{L}}{\partial K}$	$(n \times d_k)$	Gradient to keys
$\frac{\partial \mathcal{L}}{\partial W^Q}$	$(d_{\text{model}} \times d_k)$	Update for W^Q
$\frac{\partial \mathcal{L}}{\partial W^K}$	$(d_{\text{model}} \times d_k)$	Update for W^K
$\frac{\partial \mathcal{L}}{\partial W^V}$	$(d_{\text{model}} \times d_v)$	Update for W^V
$\frac{\partial \mathcal{L}}{\partial X}$	$(n \times d_{\text{model}})$	Backprop to input

---

4. Key Takeaways

1. **Weight Matrices (W^Q, W^K, W^V)** are **learned** via gradient descent using the above updates.
2. **Dot products (QK^T)** compute token-to-token relevance.
3. **Softmax gradients** adjust attention weights based on downstream errors.
4. **Backpropagation** flows through:
   - Attention scores → Queries/Keys → Weights → Input.

Suggestion: Build a **numerical example** to see how these variables and gradients are computed in practice.